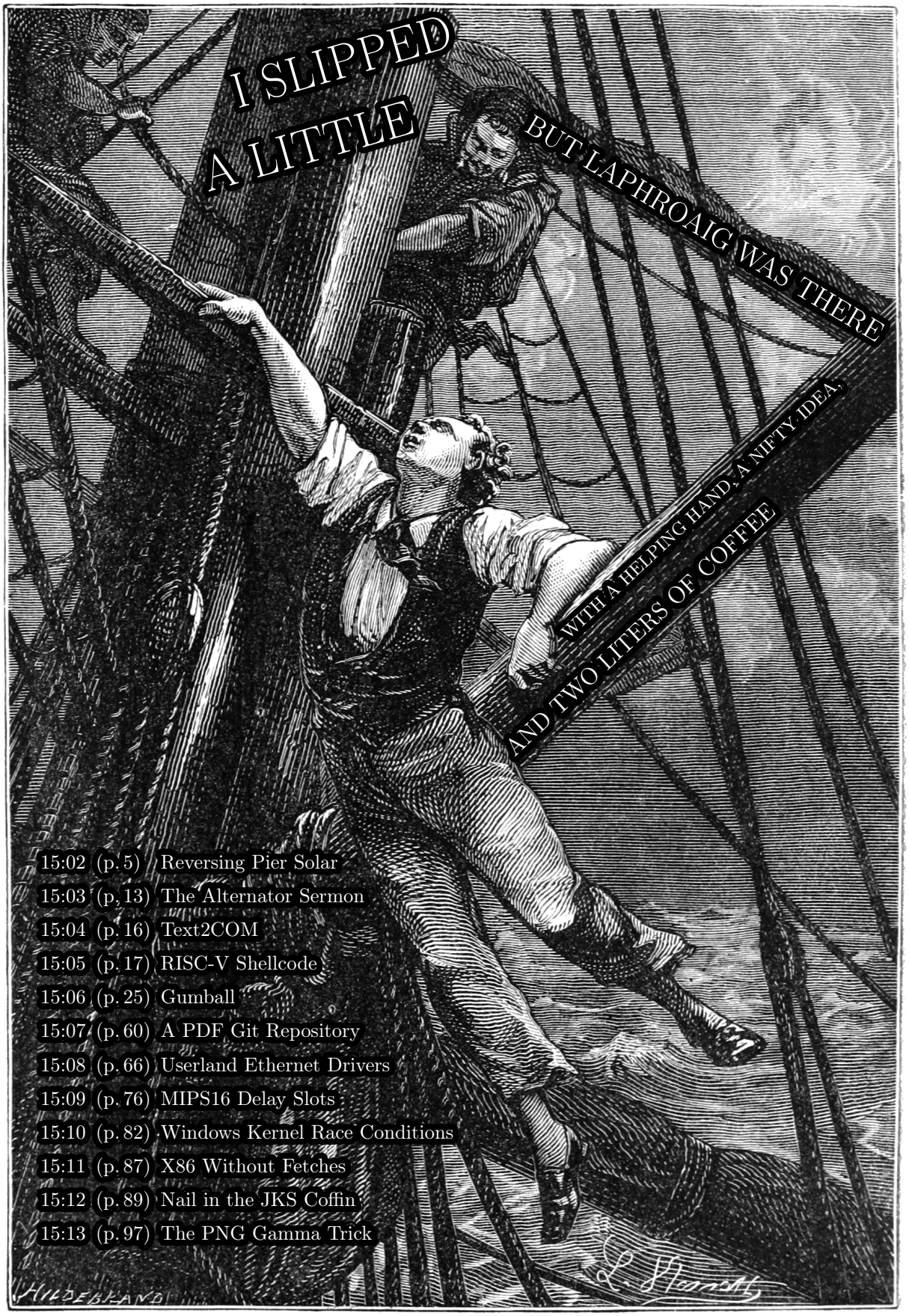
PoC||GTFO



Aide-toi et le ciel t'aidera ; это самиздат.
Compiled on June 17, 2017. Free Radare2 license included with each and every copy!
€ 0, $0 USD, $0 AUD, 10s 6d GBP, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő).

Legal Note: If you learn something from this magazine, even just one nifty little idea, you are politely requested to share that with a neighbor over a good cup of coffee.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo15.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/`

Technical Note: This file, `pocorgtfo15.pdf`, is valid as PDF document and as a ZIP file of the relevant source code. Those of you who have laser projection equipment supporting the ILDA standard will find that this issue can be handily projected by your laser beams.

Cover Art: The cover illustration from this issue is a Hildebrand engraving of a painting by Léon Benett that was first published in *Le tour du monde en quatre-vingts jours* by Jules Verne in 1873. In George M. Towle's English translation of the same year, you will find this illustration on page 137.

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 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 pocorgtfo15.pdf -o pocorgtfo15-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

and sundry others

15:01 There's no excuse for not knowing.

Neighbors, please join me in reading this sixteenth 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 Montréal and Las Vegas.

If you are missing the first fifteen 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, or the fifteenth release in Canberra, Heidelberg, or Miami.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo15.pdf`. It is a valid PDF document and a ZIP file of the relevant source code. Those of you who have laser projection equipment supporting the ILDA standard will find that this issue can be handily projected by your laser beams.

At BSides Knoxville in 2015, Brandon Wilson gave one hell of a talk on how he dumped the cartridge of Pier Solar, a modern game for the Sega Genesis; the lost lecture was not recorded and the slides were never published. After others failed with traditional cartridge dumping techniques, Brandon jumped in to find that the cartridge only provides the first 32 kB until an unlock sequence is executed, and that it will revert to the first 32 KB if it ever detects that the CPU is not executing from ROM. On page 5, Brandon will explain his nifty tricks for avoiding these protection mechanisms, armed with only the right revision of Sega CD, a serial cable, and a few cheat codes for the Game Genie.

Pastor Laphroaig is back on page 13 with a sermon on alternators, Studebakers, and bug hunting in general. This allegory of a broken Ford might teach you a thing or two about debugging, and why all the book learning in the world won't match the experience of repairing your own car.

Page 16 by Saumil Shah reminds us of those fine days when magazines would include type-in code. This particular example is one that Saumil authored twenty-five years ago, a stub that produces a self-printing COM file for DOS.

Don A. Bailey presents on page 17 an introduction to writing shellcode for the new RISC-V architecture, a modern RISC design which might not yet have the popularity of ARM but has much finer prospects than MIPS.

Our longest article for this issue, page 25 presents the monumental task of cracking Gumball for the Apple][. Neighbors 4am and Peter Ferrie spent untold hours investigating every nook and cranny of this game, and their documentation might help you to preserve a protected Apple game of your own, or to craft some deviously clever 6502 code to stump the finest of reverse engineers.

Evan Sultanik has been playing around with the internals of Git, and on page 60 he presents a PDF which is also a Git repository containing its own source code.

Rob Graham is our most elusive author, having promised an article for PoC||GTFO 0x04 that finally arrived this week. On page 66 he will teach you how to write Ethernet card drivers in userland that never switch back to the kernel when sending or receiving packets. This allows for incredible improvements to speed and drastically reduced memory requirements, allowing him to portscan all of `/0` in a single sweep.

Ryan Speers and Travis Goodspeed have been toying around with MIPS anti-emulation techniques, which this journal last covered in PoC||GTFO 6:6 by Craig Heffner. This new technique, found on page 76, involves abusing the real behavior of a branch-delay slot, which is a bit more complicated than what you might remember from your Hennessy and Patterson textbook.

Page 82 describes how BSDaemon and NadavCH reproduced the results of the Gynvael Coldwind's and jur00's Pwnie-winning 2013 paper on race conditions, using Intel's SAE tracer to not just verify the results, but also to provide new insights into how they might be applied to other problems.

Chris Domas, who the clever among you remember from his Movfuscator, returns on page 87 to demonstrate that X86 is Turing-complete without data fetches.

Tobias Ospelt shares with us a nifty little tale on page 89 about the Java Key Store (JKS) file format, which is the default key storage method for both Java and Android. Not content with a simple proof of concept, Tobias includes a fully functional patch against Hashcat to properly crack these files in a jiffy.

There's a trick that you might have fallen prey to: sometimes there's a perfectly innocent thumbnail of an image, but when you click on it to view the full image, you are hit with different graphics entirely. On page 97, Hector Martin presents one technique for generating these false thumbnail images with `gAMA` chunks of a PNG file.

On page 100, the last page, we pass around the collection plate. Our church has no interest in cash or wooden nickels, but we'd love your donation of a nifty reverse engineering story. Please send one our way.

15:02 Pier Solar and the Great Reverser

by Brandon L. Wilson

Hello everyone!

I'm here to talk about dumping the ROM from one of the most secure Sega Genesis game ever created.

This is a story about the unusual, or even crazy techniques used in reverse engineering a strange target. It demonstrates that if you want to do something, you don't have to be the best or the most qualified person to do it—you should do what you know how to do, whatever that is, and keep at it until it works, and eventually it will pay off.

First, a little background on the environment we're talking about here. For those who don't know, the Sega Genesis is a cartridge-based, 16-bit game console made by Sega and released in the US in 1989. In Europe and Japan, it was known as the Sega Mega Drive.

As you may or may not know, there were three different versions of the Genesis. The Model 1 Genesis is on the left of Figure 1. Some versions of this model have an extension port, which is actually just a third controller port. It was originally intended for a modem add-on, which was later scrapped.

pierSolar
and the great ~~architects~~ reverser

SEGA GENESIS

Some versions of the Model 1 (and all of the Model 2 devices) started to include a cartridge protection mechanism called the TMSS, or TradeMark Security System. Basically this was just some extra logic to lock up some of the internal Genesis hardware if the word "SEGA" didn't appear at a certain location in the ROM and if the ASCII bytes representing "`S`", "`E`", "`G`", "`A`" weren't written to a certain hardware register. Theoretically only people with official Sega documentation would know to put this code in their games, thereby preventing unlicensed games, but that of course didn't last long.

And then there's the Model 3 of my childhood living room, which generally sucked. It doesn't support the Sega CD, Game Genie, or any other interesting accessories.

There was also a not-as-well-known CD add-on for the Genesis called the Sega CD, or the Mega CD in Europe and Japan, released in 1992. It allowed for slightly-nicer-looking CD-based games as an attempt to extend the Genesis' life, but like many other attempts to do so, that didn't really work out.

Sega CD has its own BIOS and Motorola 68k processor, which gets executed if you don't have a cartridge in the main slot on top. That way you can still play all your old Genesis games, but if you didn't have one of those games inserted, it would boot off the Sega CD BIOS and then whatever CD you inserted.

There were two versions of it, the first one was shaped to fit the Model 1 Genesis, and while the second was modeled for the shape of the Model 2 Genesis, although either would work on the other Genesis. The Model 1 is rare and prone to failure, so it's much more difficult to find. I have the Model 2.

So finally we get to the game itself, a game called Pier Solar. It was released in 2010 and is a "homebrew" game, which means it was programmed by a bunch of fans of the Genesis, not in any way licensed by Sega. Rather than just playing it in an emulator, they took the time to produce an actual plastic cartridge just like real games, make the plastic case for it, nice printed manual, everything just as if it

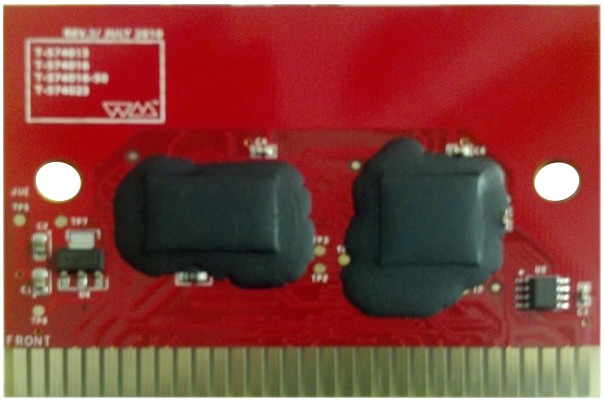
were a real game.

It's unique in that it is the only game ever to use the Sega CD add-on for an enhanced soundtrack while you're playing the game, and it has what they refer to as a "high-density" cartridge, which means it has an 8MB ROM, larger than any Genesis game ever made.

It's also unique in that its ROM had never been successfully dumped by anyone, preventing folks from playing it on an emulator. The lack of a ROM dump was not from lack of trying, however.

Taking apart the cartridge, you can see that they're very, very protective of something. They put some sort of black epoxy over the most interesting parts of the board, to prevent analysis or direct dumping of what is almost certainly flash memory.

Since they want to protect this, it's our obligation to try and understand what it is and, if necessary, defeat it. I can't help it; I see something that someone put a lot of effort into protecting, and I just *have* to un-do it.



I have no idea how to get that crud off, and I have to assume that since they put it on there, it's not easy to remove. We have to keep in mind, this game and protection were created by people with a long history of disassembling Genesis ROMs, writing Genesis emulators, and bypassing older forms of copy protection that were used on clones and pirate cartridges. They know what people are likely to try in order to dump it and what would keep it secure for a long time.

So we're going to have to get creative to dump this ROM.

There are two methods of dumping Sega Genesis ROMs. The first would be to use a device dedicated to that purpose, such as the Retrode. Essentially it pretends to be a Sega Genesis and retrieves each byte of the ROM in order until it has it all.

Unfortunately, when other people applied this to the 8MB Pier Solar, they reported that it just produces the same 32KB over and over again. That's obviously not right, so they must have some hardware under that black crud that ensures it's actually running in a Sega Genesis.

So, we turn to the other main method of dumping Genesis ROMs, which involves running a program on the Genesis itself to read the inserted cartridge's data and output it through one of the controller ports, which as I mentioned before is actually just a serial port. The people with the ability to do this also reported the same 32KB mirrored over and over again, so that doesn't work either.

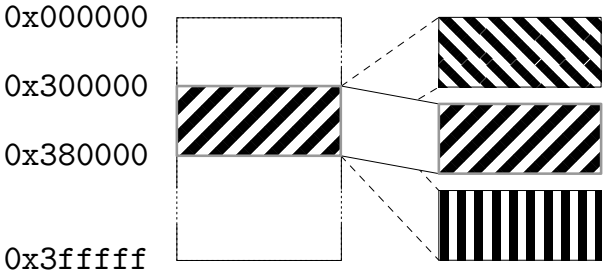
Where's the rest of the ROM data? Well, let's take a step back and think about how this works. When we do a little Googling, we find that "large" ROMs are not a new thing on the Genesis. Plenty of games would resort to tricks to access more data than the Genesis could normally.

Figure 1. From left to right, Sega Genesis models 1, 2, and 3.

The system only maps four megabytes of cartridge memory, probably because Sega figured, "-Four megs is enough ROM for anybody!" So it's impossible for it to directly reference memory beyond this region. However some games, such as Super Street Fighter 2, are actually larger than that. That game in particular is five megabytes.

They get access to the rest of the ROM by using a really old trick called bank switching. Since they know they can only address 4MB, they just change which 4MB is visible at any one time, using external hardware in the cartridge. That external hardware is called a memory mapper, because it "maps" various sections of the ROM into the addressable area. It's a poor man's MMU.

So the game itself can communicate with the cartridge and tell the mapper "Hey, I need access to part of that last megabyte. Put it at address `0x300000` for me." When you access the data at `0x300000`, you're really accessing the data at, say, `0x400000`, which would normally be just outside of the addressable range.



[1] `unzip pocorgtfo15.pdf comcable11.zip`

All this is documented online, of course. I found it by Googling about Genesis homebrew and programming your own games.

So where does this memory mapper live? It's in the game cartridge itself. Since the game runs from the Genesis CPU, it needs a way to communicate with the cartridge to tell it what memory to map and where.

All Genesis I/O is memory-mapped, meaning that when you read from or write to a specific memory address, something happens externally. When you write to addresses `0xA130F3` through `0xA130FF`, the cartridge hardware can detect that and take some kind of action. So for Super Street Fighter 2, those addresses are tied to the memory mapper hardware, which swaps in blocks of memory as needed by the game.

Pier Solar does the same thing, right? Not exactly; loading up the first 32KB in IDA Pro reveals no reads or writes here, nor to anywhere else in the `0xA130xx` range for that matter. So now what?

Well, and this is something important that we have to keep in mind, if the game's code can access all the ROM data, then so can our code. Right? If they can do it, we can do it.

– — — – — — — – — – —

So the question becomes, how do we run code on a Sega Genesis? The same way others tried dumping the ROM—through what's called the Sega CD transfer cable. This is an easy-to-make cable linking a PC's parallel port with one of the Genesis' controller ports, which as I said before is just a serial port. There are no resistors, capacitors, or anything like that. It's literally just the parallel port connector, a cut-up controller cable, and the wire between them. The cable pinout and related software are publicly available online.[1]

As I mentioned before, while the Sega CD is attached, the Genesis boots from the top cartridge slot *only if* a game is inserted. Otherwise, it uses the BIOS to boot from the CD.

Since they weren't too concerned with CD piracy way back in 1992, there is no protection at all against simply burning a CD and booting it. We burn a CD with a publicly-available ISO of a Sega CD program that waits to receive a payload of code to execute from a PC via the transfer cable. That gives us a way of writing code on a PC, transferring it to a Sega Genesis + Sega CD, running it, and communicating back and forth with a PC. We now

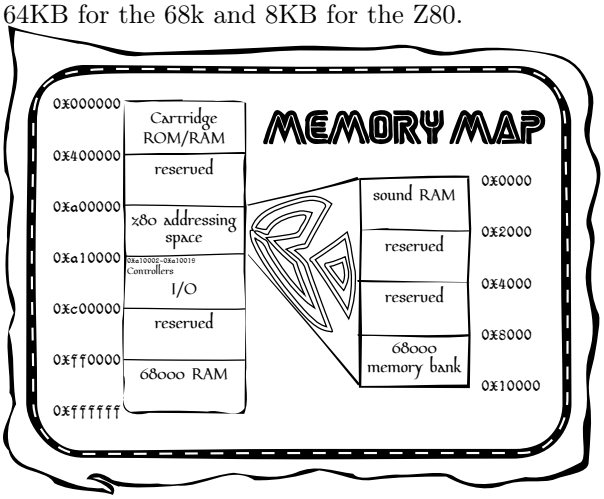
have ourselves a framework for dumping the ROM.

Great, we found some documentation online about how to send code to a Genesis and execute it, now what?

Well, let's start with trying to understand what code for this thing would even look like. Wikipedia tells us that it has two processors. The main processor is a Motorola 68000 CPU running at 7.6MHz, and which can directly access the other CPU's RAM.

The second CPU is a Zilog Z80 running at 4MHz, whose sole purpose is to drive the Yamaha YM2612 FM sound chip. The Z80 has its own RAM, which can be reset or controlled by the main Motorola 68000. It also has the ability to access cartridge ROM—so typically a game would play sound by transferring over to the Z80's RAM a small program that reads sound data from the cartridge and dumps it to the Yamaha sound chip. So when the game wanted to play a sound, the Motorola 68k would reset the Z80 CPU, which would start executing the Z80 program and playing the sound.

So anyway, combined that's 72KB of RAM: 64KB for the 68k and 8KB for the Z80.



Documentation also tells us the memory map of the Genesis. The first part we've already covered, that we can access up to `0x400000`, or 4MB, of the cartridge memory. The next useful area starts at `0xA00000`, which is where you would read from or write to the Z80's RAM.

After that is the most important area, starting at `0xA10000`, which is where all the Genesis hardware is controlled. Here we find the registers for manipulating the two controller ports, and the area I mentioned earlier about communicating directly with the hardware in the cartridge.

We also have 64KB of Motorola 68k RAM, starting at address `0xFF0000`. This should give you an idea of what code would look like, essentially reading from and writing to a series of memory mapped I/O registers.

Reports online are that the standard Sega CD transfer cable ROM dumping method doesn't work, but since we have the source code to it, let's go ahead and try it ourselves. To do that, I needed an older Genesis and Sega CD. I went to a flea market and picked up a Model 1 Sega Genesis and Model 2 Sega CD for a few dollars, then soldered together a transfer cable.

We now have the Sega Genesis attached to the Sega CD and our boot CD inserted, we then cover up the "cartridge detect" pin with tape, so that it won't detect an inserted cartridge. It will boot to the Sega CD.

As the system turns on, the Sega CD and then our burned boot CD starts up. Then the ROM dumping program is transferred over from the PC and executed on the Genesis.

The dump is transferred back to the PC via the transfer cable. We take a look at it in a hex editor, but the infernal thing is *still* mirrored.

Why is this happening? Well, we're reading the data off the cartridge using the Genesis CPU, the same way the game runs, so maybe the cartridge hardware requires a certain series of instructions to execute first? I mean, a certain set of values might need to be written to a certain address, or a certain address might need to be read.

If that's the case, maybe we should let the game boot as much as possible before we try the dump. But, if the game has booted, we're going to need to steal control away from it, which means we need to change how it runs.

Enter the Game Genie, which you might remember from when you were a kid. You'd plug your game into the cartridge slot on top of the Game Genie, then put that in your Genesis, turn it on, flip through a code book and enter your cheat codes, then hit START and cheat to your heart's content.

As it turns out, this thing is actually very useful. What it really does is patch the game by intercepting attempts to read cartridge ROM, changing them before they make it to the console for execution. The codes are actually address/value pairs. For example, if there's a check in a game to jump to a "you're dead" subroutine when your health is at zero, you could simply NOP out that Motorola 68k assembly instruction. It will never take that jump, and your character will never die.

Those of you who grow up with this thing might remember that some games had a "master" code that was required before any other codes. That code was for defeating the ROM checksum check that the game does to make sure it hasn't been tampered with. So once you entered the master code, you could make all the changes you wanted.

Since the code format is documented,[2] we can easily make a Game Genie code that will change the value at a certain address to whatever we specify. We can make minor changes to the game's code while it runs.

Due to the way the Motorola 68k works, we can only change one 16-bit word at a time, never just a single byte. No big deal, but keep it in mind because it limits the changes that we can make.

Well, that's nice in theory, but can it really work with this game? First we fire up the game with the Game Genie plugged in, but *don't* enter any codes, just to see if the cartridge works while it's attached.

Yes, it does, so next we fire up the game, again with the Game Genie plugged in, but *this* time we enter a code that, say, locks up hard. Now, that's not the best test in the world, since the code could be doing something we don't understand, but if the game suddenly won't boot, we know at least we've made an impact.

Now, according to online documentation, the format of a Genesis ROM begins with a 256-byte interrupt vector table of the Motorola 68k,followed by a 256-byte area holding all sorts of information about the ROM, such as the name of the game, the author, the ROM checksum, etc. Then finally the game's machine code begins at address `0x0200`.

If we make a couple of Game Genie codes that place the Motorola 68k instruction "`jmp 0x0200`" at `0x200`, the game will begin with an infinite loop. I tried it, and that's exactly what happened. We can lock the game up, and that's a pretty strong indication that this technique might work.

Getting back to our theory: if the game needs to execute a special set of instructions to make the 32KB mirroring stop, we need to let it run and then take back control and dump the ROM. How do we know when and where to do that? We fire up a disassembler and take a look.

```
1  0x0ec6   2079000015de   movea.l 0x15de.l, a0
   0x0ecc   317c0001000a   move.w 0x1, 0xa(a0)
3  0x0ed2   588f           addq.l 0x4, a7
   0x0ed4   600c           bra.b 0xee2
5  0x0ed6   2079000015de   movea.l 0x15de.l, a0
   0x0edc   317c0001000a   move.w 0x1, 0xa(a0)
7  0x0ee2   0839000000c0   btst.b 0x0, 0xc00005.l
   0x0eea   670e           beq.b 0xefa
9  0x0eec   2079000015de   movea.l 0x15de.l, a0
   0x0ef2   317c0bb80004   move.w 0xbb8, 0x4(a0)
11 0x0ef8   600c           bra.b 0xf06
   0x0efa   2079000015de   movea.l 0x15de.l, a0
13 0x0f00   317c0e100004   move.w 0xe10, 0x4(a0)
   0x0f06   2079000015de   movea.l 0x15de.l, a0
15 0x0f0c   0c680001000a   cmpi.w 0x1, 0xa(a0)
   0x0f12   6608           bne.b 0xf1c
17 0x0f14   4ef90000e000   jmp 0xe000.l
```

[2] `unzip pocorgtfo15.pdf MakingGenesisGGcodes.txt AdvancedGenGGtips.txt`

It is at `0x000F14` that the code takes its first jump outside of the first 32KB, to address `0x00E000`. So assuming this code executes properly, we know that at the moment the game takes that jump, the mirroring is no longer occurring. That's the safest moment to take control. We don't yet have any idea what happens once it jumps there, as this first 32KB is all we have to study and work with.

So we can make 16-bit changes to the game's code as it runs via the Game Genie, and separately, we can run code on the Genesis and access at least part of the cartridge's ROM via the Sega CD. What we really need is a way to combine the two techniques.

So then I had an idea: What if we booted the Sega CD and wrote some 68k code to embed a ROM dumper at the end of 68k RAM, then insert the Game Genie and game while the system is on, then hit the RESET button on the console, which just resets the main 68k CPU, which means our ROM dumper at the end of 68k RAM is still there It should then go to boot the Game Genie this time instead of the Sega CD, since there's now a cartridge in the slot, then enter Game Genie codes to make the game jump straight into 68k RAM, then boot the game, giving us control?

That's quite a mouthful, so let's go over it one more time.

- We write some 68k shellcode to read the ROM data and push it out the controller port back to the PC.
- To run this code, we boot the Sega CD, which receives and executes a payload from the PC.
- This payload copies our ROM dumping code to the end of 68k RAM, which the 32KB dump doesn't seem to use.
- We insert our Game Genie and game into the Genesis. This makes the system lock up, but that's not necessarily a bad thing, as we're about to reset anyway.
- We hit the RESET button on the console. The Genesis starts to boot, detects the Game Genie and game cartridge so it boots from those instead of the CD.
- We enter our Game Genie codes for the game to jump into 68k RAM and hit START to start the game, aaaand...
- Attempting this technique, the system locks up just as we should be jumping into the payload left in RAM. But why?

I went over this over and over and over in my head, trying to figure out what's wrong. Can you see what's wrong with this logic?

Yeah, so, I failed to take into account anything the Game Genie might be doing to mess with our embedded ROM dumping code in the 68K's RAM. When you disassemble the Game Genie's ROM, you find that one of the first things it does is wipe out all of the 68K's RAM.

```
1 0x0294   41f900ff0000    lea.l 0xff0000.l, a0
  0x029a   323c7fff        move.w 0x7fff, d1
3 0x029e   7000            moveq 0x0, d0
  0x02a0   30c0            move.w d0, (a0)+
5 0x02a2   51c9fffc        dbra d1, 0x2a0
```

We can't leave code in main CPU RAM across a reboot because of the very same Game Genie that lets us patch the ROM to jump into our shellcode. So what do we do?

We know we can't rely on our code still being in 68k RAM by the time the game boots, but we need something, anything to persist after we reset the console. Well, what about Z80's RAM?

Studying the Game Genie ROM reveals that it puts a small Z80 sound program in Z80 RAM, for playing the code entry sound effects, like when you're selecting or deleting a character. This program is rather small, and the Game Genie doesn't wipe out all of Z80 RAM first. It just copies this little program, leaving the rest alone.

So instead of putting our code at the end of 68K RAM, we can instead put it at the end of Z80 RAM, along with a little Z80 code to copy it back into 68k RAM. We can make a sequence of Game Genie codes that patches Pier Solar's Z80 program to jump right to the end of Z80 RAM, where our Z80 code will be waiting. We'll then be free to copy our 68k code back into 68k RAM, hopefully before the Game Genie makes the 68k jump there.

```
ROM:0000083A                movem.l d0-d2/a0-a1,-(sp)
ROM:0000083E                move.w  #$100,($A11100).l
ROM:00000846                move.w  #$14,d2
ROM:0000084A
ROM:0000084A loc_84A:                               ; CODE XREF: sub_83A+20↓j
ROM:0000084A                subq.w  #1,d2
ROM:0000084C                beq.w   loc_88A
ROM:00000850                move.w  ($A11100).l,d1
ROM:00000856                btst    #8,d1
ROM:0000085A                bne.s   loc_84A
ROM:0000085C                lea     (unk_19BC).w,a0
ROM:00000860                lea     ($A00000).l,a1
ROM:00000866                move.w  (word_1B1A).w,d0
ROM:0000086A
ROM:0000086A loc_86A:                               ; CODE XREF: sub_83A+32↓j
ROM:0000086A                move.b  (a0)+,(a1)+
ROM:0000086C                dbf     d0,loc_86A
ROM:00000870                move.w  #0,($A11200).l
ROM:00000878                move.w  #0,($A11100).l
ROM:00000880                mulu.w  d1,d0
ROM:00000882                move.w  #$100,($A11200).l
ROM:0000088A
ROM:0000088A loc_88A:                               ; CODE XREF: sub_83A+12↑j
ROM:0000088A                movem.l (sp)+,d0-d2/a0-a1
ROM:0000088E                rts
ROM:0000088E ; End of function sub_83A
```

With this new arrangement, we get control of the 68K CPU *after* the game has booted! But the extracted data is still mirrored, even though we are executing the same way the real game runs.

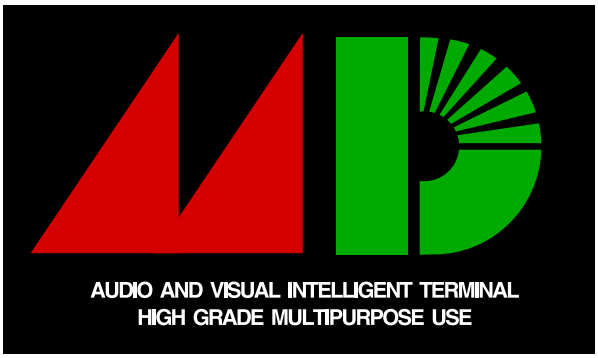
Okay, so what are the differences between the game's code and our code?

We're using a Game Genie, maybe the game detects that? This is unlikely, as the game boots fine with it attached. If it had a problem with the Game Genie, you'd think it wouldn't work at all.

Well, we're running from RAM, and the game is running from ROM. Perhaps the cartridge can distinguish between instruction fetches of code running from ROM and the data fetches that occur when code is running from RAM?

Our only ability to change the code in ROM comes from the Game Genie, which is limited to five codes. A dumper just needs to write bytes in order to `0xA1000F`, the Controller 2 UART Transmit Buffer, but code to do that won't fit in five codes.

Luckily there is a cheat device called the Pro Action Replay 2 which supports 99 codes. These are extremely rare and were never sold in the States, but I was able to buy one through eBay. Unfortunately, the game doesn't boot with it at all, even with no codes. It just sits at a black screen, even though the Action Replay works fine with other cartridges.



So now what? Well, we think that the CPU must be actively running from ROM, but except for minor patches with the Game Genie, we know our code can only run from RAM. Is there any way we can do both? Well, as it turns out, we already have the answer.

We have two processors, and we were already using both of them! We can use the Game Genie to make the 68k spin its wheels in an infinite loop in ROM, just like the very first thing we tried with it, while we use the other processor to dump it.

We were overthinking the first (and second) attempts to get control away from the game, as there's no reason the 68K *has* to be the one doing the dumping. In fact, having the Z80 do it might be the *only* way to make this work.

So the Z80 dumper does its thing, dumping cartridge data through the Sega CD's transfer cable while the 68K stays locked in an infinite loop, still fetching instructions from cartridge hardware! As far as the cartridge is concerned, the game is running normally.

And *YES*, finally, it works! We study the first 4MB in IDA Pro to see how the bank switching works. As luck would have it, Pier Solar's bank switching is almost exactly the same as Super Street Fighter 2.

Armed with that knowledge, we can modify the dumper to extract the remaining 4MB via bank switching, which I dumped out in sixteen pieces very slowly, through lots and lots and lots of triggering this crazy boot procedure. I mean, I can't tell you how excited I was that this crazy mess actually worked. It was like four o'clock in the morning, and I felt like I was on top of the world. That's why I do this stuff; really, that payoff is so worth it. It's just indescribable.

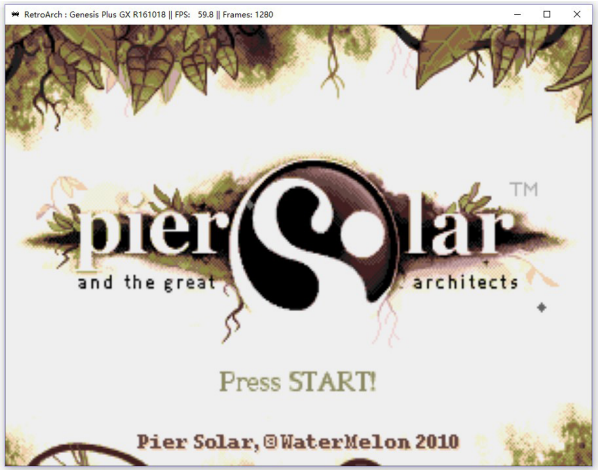
Now that I had a complete dump, I looked for the ROM checksum calculation code and implemented it PC-side, and it actually matched the checksum in the ROM header. Then I knew it was dumped correctly.

Now starts the long process of studying the disassembly to understand all the extra hardware. For example, the save-state hardware is just a serial EEPROM accessed by reads and writes to a couple of registers.

So now that we have all of it, what exactly can we say was the protection? Well, I couldn't tell you how it works at a hardware level other than that it appears to be an FPGA, but, disassembly reveals these secrets from the software side.

The first 32KB is mirrored over and over until specific accesses to `0x18010` occur. The mirroring is automatically re-enabled by hardware if the system isn't executing from ROM for more than some unknown amount of time.

[3]VDP is the display hardware in the Genesis.



The serial EEPROM, while it doesn't require a battery to hold its data, does prevent the game from running in emulators that don't explicitly support it. It also breaks compatibility with those flash cartridges that people use for playing downloaded ROMs on real consoles.

Once I got the ROM dumped, I couldn't help but try to get it working in some kind of emulator, and at the time DGen was the easiest to understand and modify, so I did the bare minimum to get that working. It boots and works for the most part, but it has a few graphical glitches here and there, probably related to VDP internals I don't and will never understand.[3]

Eventually somebody else came along and did it better, with a port to MESS.

Don't think anything is beyond your abilities: use the skills you have, whatever they may be. Me, I do TI graphing calculator programming and reverse engineering as a hobby. The two main processors those calculators use are the Motorola 68K and Zilog Z80, so this project was tailor-made for me. But as far as the hardware behind it, I had no clue; I just had to make some guesses and hope for the best.

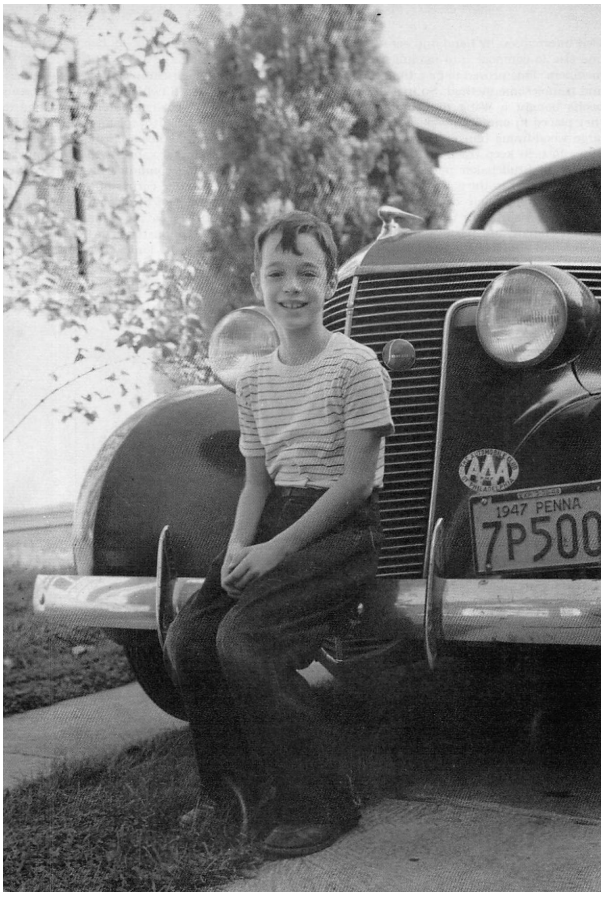
"This isn't the most efficient method" and "Nobody else would try this method." are *not* reasons to not work on something. If anything, they're actually reasons *to* do it, because that means nobody else bothered to try it, and you're more likely to be first. Crazy methods work, and I hope this little endeavor has proven that.

15:03 That car by the bear ain't got no fire; or, A Sermon on Alternators, Voltmeters, and Debugging

by Pastor Manul Laphroaig,
who is not certified by ASE.

Dear neighbors, I have a story to tell, and it's not a very flattering one.

A few years back, when I was having a bad day, I bought a five hundred dollar Mercedes and took to the open road. It had some issues, of course, so a hundred miles down the road, I stopped in rural Virginia and bought a new stereo. This was how I learned that installing a stereo in a Walmart parking lot looks a lot like stealing a stereo from a Walmart parking lot.[4]



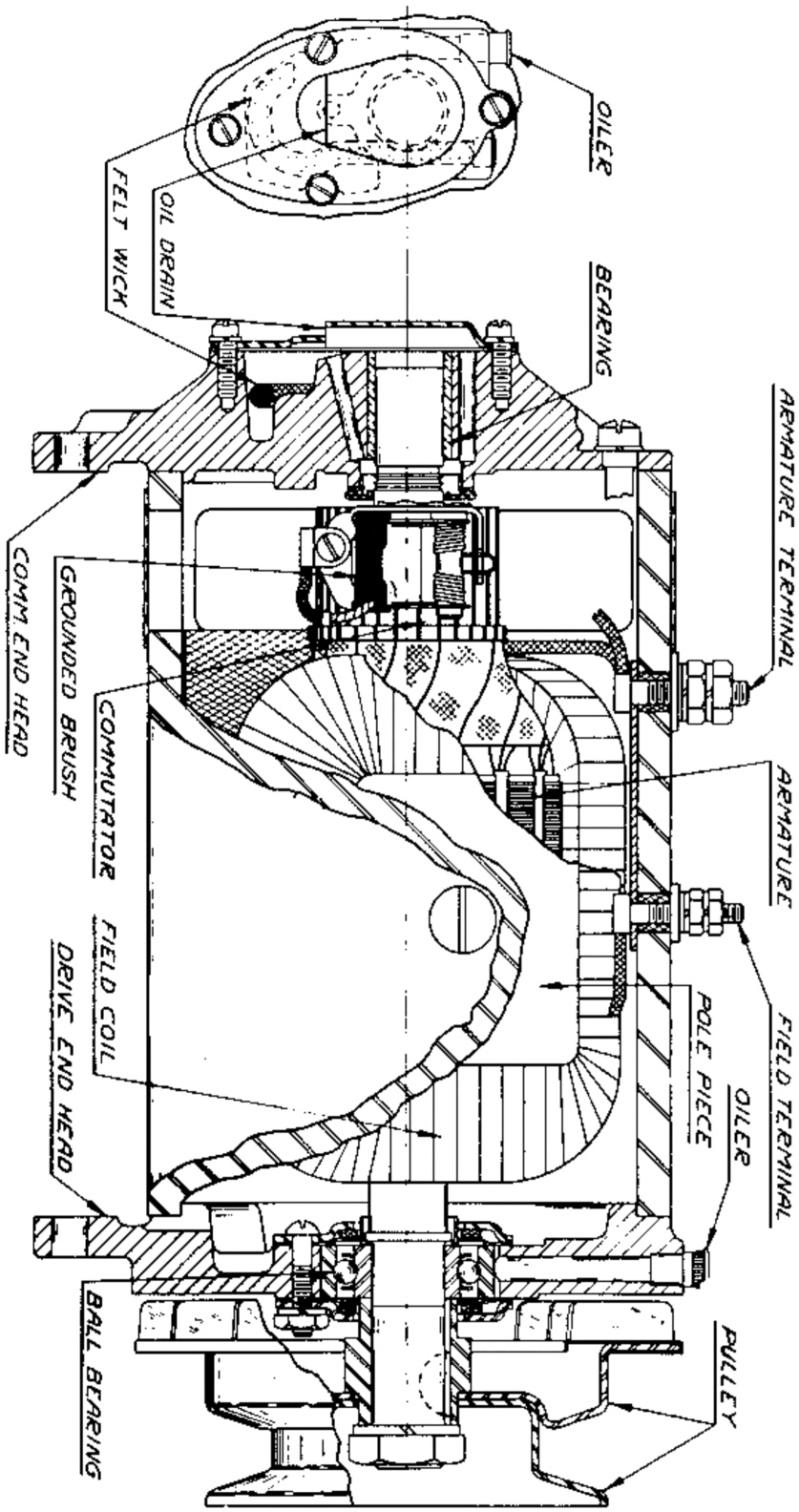
I also learned rather quickly that my four courses of auto-shop in high school amounted to a lot of book knowledge and not that much practical knowledge. My buddies who bought old cars and fixed them first-hand learned—and still know—a hell of a lot more about their machines that I ever will about mine. When squirrels chewed through the wiring harness, when metal flakes made the windshield wiper activate on its own, when the fuel line was cut by rubbish in the street as I was tearing down the Interstate at Autobahn speeds, I often took the lazy way out and paid for a professional to repair it.

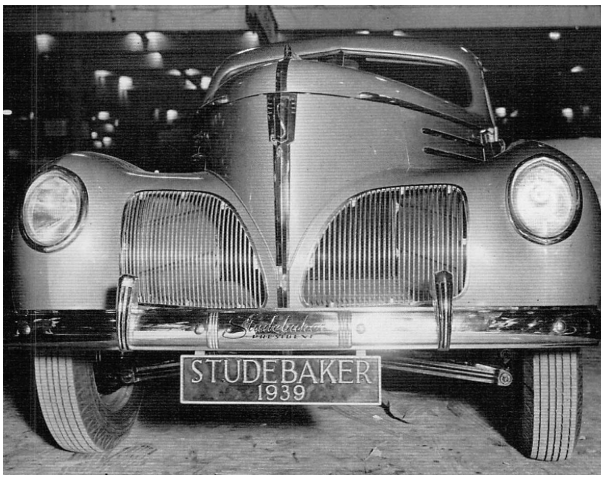
But while it's true that you learn more by building your own birdfeeder, that's not the purpose of this sermon. Today I'd like to tell you about some alternator trouble. Somehow, someway, by some mechanism unknown to gods and men, this car seemed to be killing every perfectly good alternator that was placed inside of it, and no mechanic could figure out why.

It went like this: I'd be off having adventures, then drop into town to pick up my wheels. Having been away for so long, the battery would be dead. "No big deal," I'd say and jump-start the engine. After the engine caught, I'd remove the cables, and soon enough the battery would be dead again, the engine with it. So I'd switch to driving my Ford[5] and send my car to the shop.

[4]The fastest way to clear up such a misunderstanding, when confronted by a local, is to ask to borrow some tools.

[5]In auto-shop class we learned that FORD stands for "Found On Road Dead," "Fix Or Repair Daily," or "Job Security." Coach Crigger never mentioned what Mercedes stood for, but I expect it depends upon your credit, current lease terms, and willingness to take a balloon payment!


OILER
FELT WICK
OIL DRAIN
BEARING
ARMATURE TERMINAL
COMM END HEAD
GROUNDED BRUSH
COMMUTATOR
ARMATURE
FIELD TERMINAL
FIELD COIL
POLE PIECE
OILER
DRIVE END HEAD
BALL BEARING
PULLEY



The mechanics at the shop would test the alternator, and it'd look good. They'd test the battery, and it'd look good. Then they'd start the car, and the alternator's voltage would be low, so they'd replace it out of caution. No one knew the root cause, but the part's under warranty, and the labor is cheap, so who cares?

What actually happened is this: The alternator doesn't engage until the engine revs beyond natural idling or starting. The designers must have done this to reduce the load on the starter motor, but it has the annoying side effect of letting the battery run to nothing after a jump start. The only indication to the driver is that the lights are a little dim until the gas is first pressed.

I learned this by accident after installing a voltmeter. Setting aside for the moment how absurd it is that a car ships without one, let's consider how the mechanics were fooled. In software terms, we'd say that they were confronted with a poorly reproducible test case; they were bug-hunting from anecdotes, from hand-picked artisanal data. This always ends in disaster, whether it's a frustrated software maintainer or a mechanic who becomes an unknowing accomplice to four counts of warranty fraud.

So what mistakes did I make? First, I outsourced my understanding to a shop rather than fixing my own birdfeeder. The mechanic at the shop would see my car once every six months, and he'd forget the little things. He never noticed that the lights were slightly dimmer before revving the engine, because he never started the car at night. To really understand something, you ought to have a deep familiarity with it; a passing view is bound to give you a quick little fix, or an exploit that doesn't always achieve continuation on its target.

Further, he never noticed that the battery only died after a jumpstart, but never in normal use, because all of the cars that he sees have already exhibited one problem or another and most of them were daily drivers. Whenever you are hunting a rare bug, consider the pre-existing conditions that brought that crash to your attention.[6]

Getting back to the bastard who designed a car with a single idiot light and no voltmeter, the single handiest tool to avoid these unnecessary repairs would have been to reproduce the problem when the car wasn't failing. Rather than spending months between the car failing to start, a voltmeter would have shown me that the voltage was low *only before the engine was first revved up*! In the same way, we should use every debugging tool at our disposal to make a problem reproducible in the shortest time possible, even if that visibility doesn't end in the problem that was first reported.

Paying attention to the voltage during a few drives would have revealed the real problem, even when the battery is sufficiently charged that the engine doesn't die. For this reason, we should be looking for the root cause of *EVERYTHING*, never settling for the visible effects.

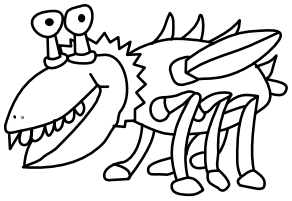
We who play with computers have debugging tools that the best mechanics can only dream of. We have checkpoint-restart debuggers which can take a snapshot just before a failure, then repeatedly execute a crash until the cause is known. We have `strace` and `dtrace` and `ftrace`, we have disassemblers and decompilers, we have `tcpdump` and `tcpreplay`, we have more hooks than Muad'Dib's Fedaykin! We can deluge the machine with a thousand core dumps, then merge them into a single test case that reproduces a crash with crystal clarity; or, if we prefer, a proof of concept that escapes from the deepest sandbox to the outer limits!

Yet the humble alternator still has important lessons to teach us.

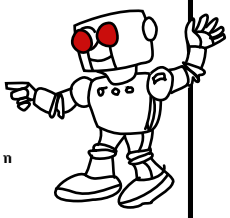
[6]Some of you may recall the story of World War II statisticians who were called in to decide where to add armor based on surveys of damage to returned Allied bombers. The right answer was to armor not where there were the most bullet holes, but where there were none. Planes hit in those areas didn't make it home to be surveyed.

15:04 Text2COM

Silver Jubilee Edition, specially re-mastered for PoC||GTFO
by Saumil Shah (@therealsaumil),
with special help from Mr. Udayan Shah



```
START:
    MOV     SI,FILE     ; Start of Text File
    PUSH    CS
    POP     DS          ; Set Data Segment = Code Segment

CLEAR:
    MOV     AH,06       ; Scroll Up Window
    XOR     AL,AL       ; 0 = Clear Screen
    MOV     BH,07       ; White over Black
    XOR     CX,CX       ; Start at 0,0
    MOV     DH,18       ; row 22
    MOV     DL,4F       ; column 79
    INT     10          ; Video Services

    MOV     AH,02       ; Set Cursor Position
    XOR     DX,DX       ; 0,0
    XOR     BH,BH       ; Page number 0
    INT     10          ; Video Services

WRITECHAR:
    LODSB               ; AL = [DS:SI]
    MOV     DL,AL       ; DL = character to write
    NOT     AL          ; 1's Complement
    XOR     AL,E5       ; E5 = 1's C (EOF)
    JZ      END         ; If EOF character, jump to END
    MOV     AH,02       ; Write Character
    INT     21          ; DOS Services

    MOV     AH,03       ; Get Cursor Position
    XOR     BH,BH       ; Page 0
    INT     10          ; Video Services. DH,DL = Row,Col

    CMP     DH,16       ; Is row 22?
    JLE     WRITECHAR   ; Jump if < 22 to WRITECHAR

    MOV     AH,09       ; Write $-Terminated String
    MOV     DX,PAGER    ; Address of Pager String
    INT     21          ; DOS Services

    MOV     AH,08       ; Read Single Character
    INT     21          ; DOS Services
    JMP     CLEAR       ; Jump to CLEAR

END:
    INT     20

PAGER:
    DB      '[Text2COM by Saumil Shah (c)1992] '
    DB      'Press Any Key... $'

FILE:
    ; Text content goes here.
```



Text2COM generates self-displaying `README.COM` files by prefixing a short sequence of DOS Assembly instructions before a text file. The resultant file is an MS-DOS `.COM` program which can be executed directly from the command prompt.

The Text2COM code displays the contents of the appended file page by page.

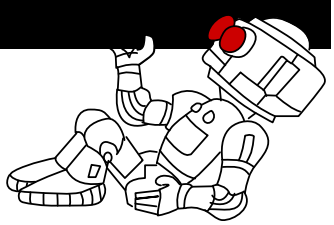
Text2COM's executable code is created by MS-DOS's `DEBUG` program.

Then take any text file and concatenate it with `README.BIN` and store the resultant file as `README.COM`:

```
C:\>copy README.BIN+TEXT2COM.TXT README.COM
```

You now have a self-displaying `README.COM` file!

```
C:\>debug
-n README.BIN
-e 100 BE 78 01 0E 1F B4 06 30 C0 B7 07 31 C9 B6 18 B2
-e 110 4F CD 10 B4 02 31 D2 30 FF CD 10 AC 88 C2 F6 D0
-e 120 34 E5 74 1C B4 02 CD 21 B4 03 30 FF CD 10 80 FE
-e 130 16 7E E8 B4 09 BA 42 01 CD 21 B4 08 CD 21 EB C5
-e 140 CD 20 5B 54 65 78 74 32 43 4F 4D 20 62 79 20 53
-e 150 61 75 6D 69 6C 20 53 68 61 68 20 28 63 29 20 31
-e 160 39 39 32 5D 20 50 72 65 73 73 20 41 6E 79 20 4B
-e 170 65 79 2E 2E 2E 20 24 0A
-rcx
CX 0000
:78
-w
Writing 00078 bytes
-q
```



15:05 RISC-V Shellcode

by Don A. Bailey

RISC-V is a new and exciting open source architecture developed by the RISC-V Foundation. The Foundation has released the Instruction Set Architecture open to the public, and a Privilege Architecture Model that defines how general purpose operating systems can be implemented. Even more exciting than a modern open source processing architecture is the fact that implementations of the RISC-V are available that are fully open source, such as the Berkeley Rocket Chip[7] and the PULPino.[8]

To facilitate silicon development, a new language developed at Berkeley, Chisel,[9] was developed. Chisel is an open-source hardware language built from Scala, and synthesizes Verilog. This allows fast, efficient, effective development of hardware solutions in far less time. Much of the Rocket Chip implementation was written in Chisel.

Furthermore, and perhaps most exciting of all, the RISC-V architecture is 128-bit processor ready. Its ISA already defines methodologies for implementing a 128-bit core. While there are some aspects of the design that still require definition, enough of the 128-bit architecture has been specified that Fabrice Bellard has successfully implemented a demo emulator.[10] The code he has written as a demo of the emulator is, perhaps, the first 128-bit code ever executed.

Binary Exploitation

To compromise a RISC-V application or kernel in the traditional memory corruption manner, one must understand both the ISA and the calling convention for the architecture. In RISC-V, the term XLEN is used to denote the native integer size of the base architecture, e.g. XLEN=32 in RV32G. Each register in the processor is of XLEN length, meaning that when a register is defined in the specification, its format will persist throughout any definition of the RISC-V architecture, except for the length, which will always equate to the native integer length.

General Registers

In general, RISC-V has 32 general (or `x`) registers: `x0` through `x31`.[11] These registers are all of length XLEN, where bit zero is the least-significant-bit and the most-significant-bit is XLEN-1. These registers have no specific meaning without the definition of the Application Binary Interface (ABI).

The ABI defines the following naming conventions to contextualize the general registers, shown in Figure 2.[12]

[7] `git clone https://github.com/freechipsproject/rocket-chip`
[8] `http://www.pulp-platform.org/`
[9] `https://chisel.eecs.berkeley.edu/`
[10] `https://bellard.org/riscvemu/`
[11] RISC-V ISA Specification v2.1, Page 10, Figure 2.1.
[12] RISC-V ISA Specification v2.1, Page 109, Table 20.2

Register	ABI Name	Description	Saver
x0	zero	Hard-wired to zero	–
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Figure 2. Naming conventions for general registers according to the current ABI.

Floating-Point Registers

RISC-V also has 32 floating point registers `fp0` through `fp31`, shown in Figure 3. The bit size of these registers is not XLEN, but FLEN. FLEN refers to the native floating point size, which is defined by which floating point extensions are supported by the implementation. If the 'F' extension is supported, only 32-bit floating point is implemented, making FLEN=32.[13] If the 'D' extension is supported, 64-bit floating point numbers are supported, making FLEN=64.[14] If the 'Q' extension is supported, quad-word floating point numbers are supported, and FLEN extends to 128.[15]

Calling Convention

Like any Instruction Set Architecture (ISA), RISC-V has a standard calling convention. But, because of the RISC-V's definition across multiple architectural subclasses, there are actually three standardized calling conventions: `RVG`, Soft Floating Point, and `RV32E`.

Naming Conventions RISC-V's architecture is somewhat reminiscent of the Plan 9 architecture naming style, where each architecture is assigned a specific alphanumeric `A` through `Z` or `0` through `9`. RISC-V supports 24 architectural extensions, one for each letter of the English alphabet. The two exceptions are `G` and `X`. The `G` extension is actually a mnemonic that represents the RISC-V architecture extension set `IMAFD`, where `I` represents the base integer instruction set, `M` represents multiply/divide, `A` represents atomic instructions, `F` represents single-precision floating point, and `D` represents double-precision floating point. Thus, when one refers to `RVG`, they are indicating the RISC-V (RV) set of architecture extensions `G`, actually referring to the combination `IMAFD`.[16]

This colloquialism also implies that there is no specific architectural bit-space being singled out: all three of the 32-bit, 64-bit, and 128-bit architectures are being referenced. This is common in description of the architectural standard, software relevant to all architectures (a kernel port), or discussion about the ISA. It is more common, in development, to see the architecture described with the bit-space included in the name, e.g. RV32G, RV64G, or RV128G.

It is also worth noting here that it is defined in the specification and core register set that an implementation of RISC-V can support all three bit-spaces in a single processor, and that the state of the processor can be switched at run-time by setting the appropriate bit in the Machine ISA Register misa.[17]

Thus, in this context, the `RVG` calling convention denotes the model for linking one function to another function in any of the three RISC-V bit-spaces.

[13]RISC-V ISA Specification v2.1, Section 7.1, Page 39
[14]RISC-V ISA Specification v2.1, Section 8.1
[15]RISC-V ISA Specification v2.1, Chapter 12, Paragraph 1
[16]RISC-V Privileged Architecture Manual v1.9.1, Section 3.1.1, Page 18
[17]Ibid.
[18]RISC-V ISA Specification v2.1, Page 6, Paragraph 1

Register	ABI Name	Description	Saver
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Figure 3. Floating point register naming convention according to the current ABI.

RVG RISC-V is little-endian by definition and big or bi-endian systems are considered non-standard.[18] Thus, it should be presumed that all RISC-V implementations are little-endian unless specifically stated otherwise.

To call any given function there are two instructions: Jump and Link and Jump and Link Register. These instructions take a target address and branch to it unconditionally, saving the return address in a specific register. To call a function whose address is within 1MB of the caller's address, the `jal` instruction can be used:

```
1 20400060:   661000ef   jal 20400ec0 <printk>
```

To call a function whose address is either generated dynamically, or is outside of the 1MB target range, the `jalr` instruction must be used:

```
1 204001ac:   0087a783   lw    a5,8(a5)
  204001b0:   000780e7   jalr  a5
```

In both of the above examples, bits 7 through 11 of the encoded opcode equate to `0b00001`. These bits indicate the destination register where the return address is stored. In this case, 1 is equivalent to register `x1`, also known as the return address register: `ra`. In this fashion, the callee can simply perform their specific functionality and return by using the contents of the register `ra`.

Returning from a function is even simpler. In the RISC-V ABI, we learned earlier that the return address is presumed to be stored in `ra`, or, general register `x1`. To return control to the address stored in `ra`, we simply use the Jump and Link Register instruction, with one slight caveat. When returning from a function, the return address can be discarded. So, the encoded destination register for `jalr` is `x0`. We learned earlier that `x0` is hardwired to the value zero. This means that despite the return address being written to `x0`, the register will always read as the value zero, effectively discarding the return address.

Thus, a return instruction is colloquially:

```
204002a8:   00008067   ret
```

Which actually equates to the instruction:

```
1 204002a8:   00008067   jalr ra, zero
```

Local stack space can be allocated in a similar fashion to any modern processing environment. RISC-V's stack grows downward from higher addresses, as is common convention. Thus, to allocate space for automatics, a function simply decrements the stack pointer by whatever stack size is required.

```
1  20402188 <arch_main>:
   20402188:   fe010113   addi sp,sp,-32
3  2040218c:   80000537   lui  a0,0x80000
   20402190:   80000637   lui  a2,0x80000
5  20402194:   00112e23   sw   ra,28(sp)

7  20402220:   01c12083   lw   ra,28(sp)
   20402224:   02010113   addi sp,sp,32
9  20402228:   00008067   ret
```

In the above example, a standard `addi` instruction (highlighted in red) is used to both create and destroy a stack frame of 32 bytes. Four of these bytes are used to store the value of `ra`. This implies that this function, `arch_main`, will make calls to other functions and will require the use of `ra`. The lines highlighted in green depict the saving and retrieval of the return address value.

This fairly standard calling convention implies that binary exploitation can be achieved, but has several caveats. Like most architectures, the return address can be overwritten in stack memory, meaning that standard stack buffer overflows can result in the control of execution. However, the return address is only stored in the stack for functions that make calls to other functions.

Leaf functions, functions that make no calls to other functions, do not store their return address on the stack. These functions, similar to other RISC architectures, must be attacked by

- Overwriting the previous function's stack frame or stored return address
- Overwriting the return address value in register ra
- Manipulating application flow by attacking a function-specific feature such as a function pointer

Soft-Float Calling Convention With regard to the threat of exploitation, the RISC-V soft-float calling convention has little effect on an attacker strategy. The `jal`/`jalr` and stack conventions from `RVG` persist. The only difference is that the floating point arguments are passed in argument registers according to their size. But, this typically has little effect on general exploitation theory and will only be abused in the event that there is an application-specific issue.

It is notable, however, that implementations with hard-float extensions may be vulnerable to memory corruption attacks. While hard-float implementations use the same `RVG` calling conventions as defined above, they use floating point registers that are used to save and restore state within the floating point ecosystem. This may provide an attacker an opportunity to affect an application in an unexpected manner if they are able to manipulate saved registers (either in the register file or on the stack).

While this is application specific and does not apply to general exploitation theory, it is interesting in that the RISC-V ABI does implement saved and temporary registers specifically for floating point functionality.

RV32E Calling Convention It's important to note the RV32E calling convention, which is slightly different from `RVG`. The `E` extension in RISC-V denotes changes in the architecture that are beneficial for 32-bit Embedded systems. One could liken this model to ARM's Cortex-M as a variant of the Cortex-A/R, except that `RVG` and `RV32E` are more tightly bound.

`RV32E` only uses 16 general registers rather than 32, and never has a hard-floating point extension. As a result, exploit developers can expect the call and local stack to vary. This is because, with the reduced number of general registers, there are less argument registers, save registers, and temporaries.

- 6 argument registers, `x10` to `x15`.
- 2 save registers, `x8` and `x9`.
- 3 temporary registers, `x5` to `x7`.

As is described earlier in this document, the general `RVG` model is

- 8 argument registers.
- 12 save registers.
- 7 temporary registers.

Functions defined with numbers of arguments exceeding the argument register count will pass excess arguments via the stack. In RV32E this will obviously occur two arguments sooner, requiring an adjustment to stack or frame corruption attacks. Save and temporary registers saved to stack frames may also require adjustments. This is especially true when targeting kernels.

The 'C' Extension Effect

The RISC-V `C` (compression) extension can be considered similar to the Thumb variant of the ARM ISA. Compression reduces instructions from 32 to 16 bits in size. For exploits where shellcode is used, or Return Oriented Programming (ROP) is required, the availability (or lack) of `C` will have a significant effect on the effects of an implant.

An interesting side effect of the `C` extension is that not all instructions are compressed. In fact, in the Harvest OS kernel (a Lab Mouse Security proprietary operating system), the compression extension currently only results in approximately 60% of instructions compressed to 16 bits.

Because the processor must evaluate the type of an instruction at every fetch (compressed or not) when compression is available, there is a CISC-like effect for exploitation. Valid compressed instructions may be encoded in the lower 16 bits of an existing 32-bit instruction. This means that someone, for example, implementing a ROP attack against a target may be able to find useful 16 bit opcodes embedded in intentional 32-bit opcodes. This is similar to a paper I wrote in 2002 that demonstrated that ROP on CISC architectures (then called return-to-text) could abuse long multi-byte opcodes to target useful bytes that represented beneficial opcodes not intended to be used by the compiler.[19]

```
20400032 <lock_unlock>:
20400032: 0a05202f amoswap.w.rl zero,zero,(a0)
20400036: 4505     li       a0,1
20400038: 8082
```

Since the `C` extension is not a part of the `RVG IMAFD` extension set, it is currently unknown whether `C` will become a commonly implemented extension. Until RISC-V is more predominant and a key player arises in chip manufacturing, exploit developers should either target their payloads for specific machines, or should focus on the uncompressed instruction set.

Observations

Exploitation really isn't so different from other RISC targets, such as ARM. Just like ARM, the compression extension isn't necessary for ROP, but it can be handy for unintentionally encoded gadgets. While mitigations like `-fstack-protection[-all]` are supported, they require `__stack_chk_{guard-,fail}`, which might be lacking on your target platform. For Linux targets, be sure to enable `PIE, now, relro` for ASLR and GOT hardening.

Building Shellcode

Building shellcode for any given architecture generally only requires understanding how to satisfy the following abstractions:

- Allocating memory.
- Locating static data.
- Calling routines.
- Returning from routines.

Allocating Memory

Allocating memory in RISC-V environments is similar to almost any other processing environment for conventional operating systems. Since there is a stack pointer register (`sp`/`x2`), the programmer can simply take a chance and allocate memory via the stack. This presumes that there is enough available memory in the system, and that a fault won't occur. If the exploitation target is a userland application in a typical operating system, this is always a reasonable gamble as even if allocating stack would fault, the underlying OS will generally allocate another page for the userland application. So, since the stack grows down, the programmer only needs to decrement the `sp` (round up to a multiple of 4 bytes) to create more space using system stack.

[19] Sendmail Prescan Exploitation and CISCO Encodings (127 Research & Development, 2002)

Some environments may allocate thread-specific storage, accessible through a structure stored in the thread pointer (`tp`/`x4`). In this case, simply dereference the structure pointed to by `x4`, and find the pointer that references thread-local storage (TLS). It's best to store the pointer to TLS in a temporary register (or even `sp`), to make it easier to abuse.

As with most programming environments, dynamic memory is typically also available, but must be acquired through normal calling conventions. The underlying mechanism is usually `malloc`, `mmap`, or an analog of these functions.

Locating Static Data

Data stored within shellcode must be referenced as an offset to the shellcode payload. This is another normal shellcode construct. Again, RISC-V is similar to any other processing environment in this context. The easiest way to identify the address of data in a payload is to find the address in memory of the payload, or to write assembly code that references data at position independent offsets. The latter is my preferred method of writing shellcode, as it makes the most engineering sense. But, if you prefer to build address offsets within executable images, the usual shellcode self-calling convention works fine:

```
0000000000000000 <lol>:
  0:   0100006f    j       10 <bounce>
0000000000000004 <lol2>:
  4:   00000513    li      a0,0
  8:   0000a583    lw      a1,0(ra)
  c:   00000073    ecall
0000000000000010 <bounce>:
 10:   ff5ff0ef    jal     ra,4 <lol2>
0000000000000014 <data>:
 14:   0304        addi    s1,sp,384
 16:   0102        slli    sp,sp,0x0
```

As you can see in the above code example, the first instruction performs a jump to the last instruction prior to static data. The last instruction is a jump-and-link instruction, which places the return address in `ra`. The return address, being the next instruction after jump-and-link, is the exact address in memory of the static data. This means that we can now reference chunks of that data as an offset of the `ra` register, as seen in the load-word instruction above at address `0x08`, which loads the value `0x01020304` into register a1.

It's notable, at this point, to make a comment about shellcode development in general. Artists generally write raw assembly code to build payloads, because it's more elegant and it results in a much more efficient application. This is my personal preference, because it's a demonstration of one's connection to the code, itself. However, it's largely unnecessary. In modern environments, many targets are 64-bit and contain enough RAM to inject large payloads containing encrypted blobs. As a result, one can even write position independent code (PIC) applications in C (and even C++, if one dares). The resultant binary image can be injected as its own complete payload, and it runs perfectly well.

But, for constrained targets with little usable scratch memory, primary loaders, or adversaries with an artistic temperament, assembly will always be the favorite tool of trade.

Calling Routines

Earlier in this document, I described the general RISC-V calling convention. Arguments are placed in the `aN` registers, with the first argument at a0, second at a1, and so-forth. Branching to another routine can be done with the jump-and-link (`jal`) instruction, or with the jump-and-link register (`jalr`) instruction. The latter instruction has the absolute address of the target routine stored in the register encoded into the instruction, which is a normal RISC convention. This will be the case for any application routine called by your shellcode.

The Linux syscall convention, in the context of RISC-V, is likely similar to other general purpose operating systems running on RISC-V processors. The Linux model deviates from the generic calling convention by using the `ecall` instruction. This instruction, when executed from userland, initiates a trap into a higher level of privilege. This trap is processed as, of course, a system call, which allows the kernel running at the higher layer of privilege to process the request appropriately.

System call numbers are encoded into register `a7`. Other arguments are encoded in the standard fashion, in registers `a0` through `a6`. System calls exceeding seven arguments are stored on the stack prior to the call. This convention is also true of general routine calls whose argument totals exceed available argument registers.

Returning from Routines

Passing arguments back from a routine is simple, and is, again, similar to any other conventional processing environment. Arguments are passed back in the argument register `a0`. Or, in the argument pair `a0` and `a1`, depending on the context.

This is also true of system calls triggered by the `ecall` instruction. Values passed back from a higher layer of privilege will be encoded into the `a0` register (or `a0` and `a1`). The caller should retrieve values from this register (or pair) and treat the value properly, depending on the routine's context.

One notable feature of RISC-V is its compare-and-branch methodology. Branching can be accomplished by encoding a comparison of registers, like other RISC architectures. However, in RISC-V, two specific registers can be compared along with a target in the event that the comparison is equivalent. This allows very streamlined evaluation of values. For example, when the standard system call `mmap` returns a value to its caller, the caller can check for `mmap` failure by comparing `a0` to the `zero` register and using the branch-less-than instruction. Thus, the programmer doesn't actually need multiple instructions to effect the correct comparison and branch code block; a single instruction is all that is required.

Putting it Together

The following example performs all actions described in previous sections. It allocates 80 bytes of memory on the stack, room for ten 64-bit words. It then uses the aforementioned bounce method to acquire the address of the static data stored in the payload. The system call for socket is then called by loading the arguments appropriately.

After the system call is issued, the return value is evaluated. If the socket call failed, and a negative value was returned, the `_open_a_socket` function is looped over.

If the socket call does succeed, which it likely will, the application will crash itself by calling a (presumably) non-existent function at virtual address `0x00000000`.

As an example, the byte stored in static memory is loaded as part of the system call, only to demonstrate the ability to load code at specific offsets.

```
0000000000000000 <lol>:
   0:   fb010113    addi   sp,sp,-80
   4:   00113023    sd     ra,0(sp)
   8:   00813423    sd     s0,8(sp)
   c:   0200006f    j      2c <bounce>
0000000000000010 <_open_a_socket>:
  10:   00200513    li     a0,2
  14:   00100593    li     a1,1
  18:   00600613    li     a2,6
  1c:   00008883    lb     a7,0(ra)
  20:   00000073    ecall
0000000000000024 <_crash_or_loop>:
  24:   fe0546e3    bltz   a0,10 <_open_a_socket>
0000000000000028 <_crash>:
  28:   00000067    jr     zero
000000000000002c <bounce>:
  2c:   fe5ff0ef    jal    ra,10 <_open_a_socket>
0000000000000030 <data>:
  30:   00c6        slli   ra,ra,0x11
```

– – – – – – – – – – –

Big shout out to `#plan9` for still existing after 17 years, TheNewSh for always rocking the mic, Travis Goodspeed for leading the modern zine revolution, RMinnich for being an excellent resource over the past decade, RPike for being an excellent role model, and my baby Pierce, for being my inspiration.

Source code and shellcode for this article are available attached to this PDF and through Github.[20]

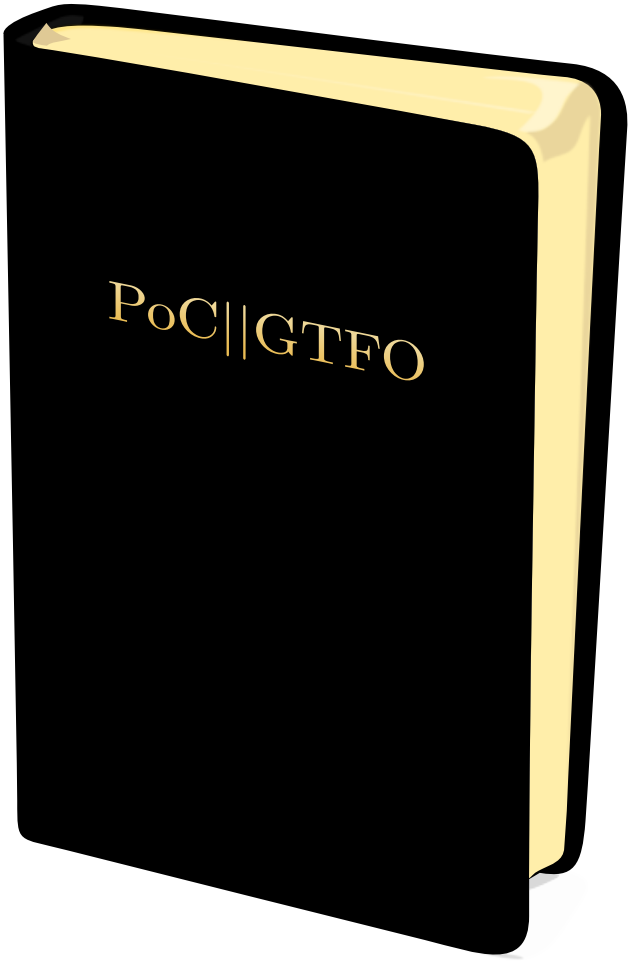
[20] `git clone https://github.com/donbmouse/riscv-security || unzip pocorgtfo15.pdf riscv-security.zip`

Dearest neighbors,

In 19th century America, there were these books made just for the frontiersman who couldn't carry a library. The idea was that if you were setting out to homestead in the wild blue yonder, one properly assembled book could teach you everything you needed to know that wasn't told in the family bible. How to make ink from the green husks around walnuts, how to grow food from wild seeds, and how to build a shelter from scruffy little trees when there's not yet time to fell hardwood. You might even learn to make medicines, though I'd caution against any recipes involving nightshade or mercury.

Now that the 21st century and its newfangled ways are upon, the fine folks at No Starch Press have seen fit to print the collected works of PoC||GTFO, our first nine releases in one classy tome, bound in the finest faux leather on nearly eight hundred pages of thin paper with a ribbon to keep your place while studying. You will see practical examples of how to write exploits for ancient and modern architectures, how to patch emulators to prototype hardware backdoors that would be beyond a hobbyist's budget, and how to break bad cryptography. You will learn more about file formats than you every believed possible, and a little about how to photograph microchips and circuit boards for reverse engineering.

This fine collection was carefully indexed and cross-referenced, with twenty-four full color pages of Ange Albertini's file format illustrations to help understand our polyglots. It's available for just $30 plus shipping, with the option of a free pickup at Defcon.



https://nostarch.com/gtfo

Your neighbor,

Pastor Manul Laphroaig

15:06 Gumball

by 4am and Peter Ferrie (qkumba, san inc)

Name Gumball

Genre arcade

Year 1983

Credits by Robert Cook, concept by Doug Carlston

Publisher Broderbund Software

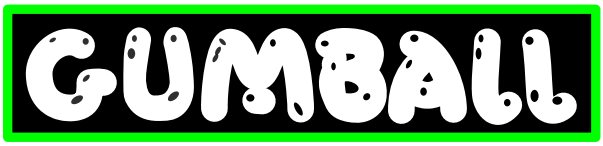
Platform Apple][+ or later (48K)

Media single-sided 5.25-inch floppy

OS custom

Other versions

- Mr. Krac-Man & The Disk Jockey
- several uncredited cracks



In Which Various Automated Tools Fail In Interesting Ways

COPYA immediate disk read error

Locksmith Fast Disk Backup unable to read any track

EDD 4 bit copy (no sync, no count) Disk seeks off track 0, then hangs with the drive motor on

Copy II+ nibble editor

- T00 has a modified address prologue (D5 AA B5) and modified epilogues
- T01+ appears to be 4-4 encoded data (2 nibbles on disk = 1 byte in memory) with a custom prologue/ delimiter. In any case, it's neither 13 nor 16 sectors.

Disk Fixer not much help

Why didn't COPYA work? not a 16-sector disk

Why didn't Locksmith FDB work? ditto

Why didn't my EDD copy work? I don't know. Early Broderbund games loved using half tracks and quarter tracks, not to mention the runtime protection checks, so it could be literally anything. Or, more likely, any combination of things.

This is decidedly not a single-load game. There is a classic crack that is a single binary, but it cuts out a lot of the introduction and some cut scenes later. All other cracks are whole-disk, multi-loaders.

Combined with the early indications of a custom bootloader and 4-4 encoded sectors, this is not going to be a straightforward crack by any definition of "straight" or "forward."

Let's start at the beginning.

In Which We Brag About Our Humble Beginnings

I have two floppy drives, one in slot 6 and the other in slot 5. My "work disk" (in slot 5) runs DiversiDOS 64K, which is compatible with Apple DOS 3.3 but relocates most of DOS to the language card on boot. This frees up most of main memory (only using a single page at `$BF00..$BFFF`), which is useful for loading large files or examining code that lives in areas typically reserved for DOS.

```
[S6,D1=original disk]
[S5,D1=my work disk]
```

The floppy drive firmware code at `$C600` is responsible for aligning the drive head and reading sector 0 of track 0 into main memory at `$0800`. Because the drive can be connected to any slot, the firmware code can't assume it's loaded at `$C600`. If the floppy drive card were removed from slot 6 and reinstalled in slot 5, the firmware code would load at `$C500` instead.

To accommodate this, the firmware does some fancy stack manipulation to detect where it is in memory (which is a neat trick, since the 6502 program counter is not generally accessible). However, due to space constraints, the detection code only cares about the lower 4 bits of the high byte of its own address.

Stay with me, this is all about to come together and go boom.

`$C600` (or `$C500`, or anywhere in `$Cx00`) is read-only memory. I can't change it, which means I can't stop it from transferring control to the boot sector of the disk once it's in memory. BUT! The disk firmware code works unmodified at any address. Any address that ends with `$x600` will boot slot 6, including `$B600`, `$A600`, `$9600`, &c.

```
*9600<C600.C6FFM                copy drive firmware to $9600

*9600G                          and execute it
```

. . .reboots slot 6, loads game. . .

Now then:

```
]PR#5 . . .
]CALL -151
*9600<C600.C6FFM
*96F8L
96F8  4C 01 08   JMP $0801
```

That's where the disk controller ROM code ends and the on-disk code begins. But $9600 is part of read/write memory. I can change it at will. So I can interrupt the boot process after the drive firmware loads the boot sector from the disk but before it transfers control to the disk's bootloader.

```
96F8     A0 00   LDY #$00      instead of jumping to on-disk
96FA  B9 00 08   LDA $0800,Y   code, copy boot sector to
96FD  99 00 28   STA $2800,Y   higher memory so it survives
9700        C8   INY           a reboot
9701     D0 F7   BNE $96FA

9703  AD E8 C0   LDA $C0E8     turn off slot 6 drive motor

9706  4C 00 C5   JMP $C500     reboot to my work disk in slot
*9600G                         5
. . .reboots slot 6. . .
. . .reboots slot 5. . .
]BSAVE BOOT0,A$2800,L$100
```

Now we get to[21] trace the boot process one sector, one page, one instruction at a time.

In Which We Get To Dip Our Toes Into An Ocean Of Raw Sewage

```
]CALL -151

*800<2800.28FFM                copy code back to $0800
801L                           where it was originally loaded,
                               to make it easier to follow

0801     A2 00   LDX #$00      immediately move this code
0803  BD 00 08   LDA $0800,X   to the input buffer at $0200
0806  9D 00 02   STA $0200,X
0809        E8   INX
080A     D0 F7   BNE $0803
080C  4C 0F 02   JMP $020F
```

OK, I can do that too. Well, mostly. The page at $0200 is the text input buffer, used by both Applesoft BASIC and the built-in monitor (which I'm in right now). But I can copy enough of it to examine this code in situ.

```
*20F<80F.8FFM
*20FL
```

```
020F     A0 AB   LDY #$AB      set up a nibble translation
0211        98   TYA           table at $0800
0212     85 3C   STA $3C
0214        4A   LSR
0215     05 3C   ORA $3C
0217     C9 FF   CMP #$FF
0219     D0 09   BNE $0224
021B     C0 D5   CPY #$D5
021D     F0 05   BEQ $0224
021F        8A   TXA
0220  99 00 08   STA $0800,Y
0223        E8   INX
0224        C8   INY
0225     D0 EA   BNE $0211
0227     84 3D   STY $3D

0229     84 26   STY $26       #$00 into zero page $26 and
022B     A9 03   LDA #$03      #$03 into $27 means we're
022D     85 27   STA $27       probably going to be loading
                               data into $0300..$03FF later,
                               because ($26) points to $0300.

022F     A6 2B   LDX $2B       zero page $2B holds the boot
0231  20 5D 02   JSR $025D     slot x16

*25DL

025D        18   CLC           read a sector from track $00
025E        08   PHP           (this is actually derived from
025F  BD 8C C0   LDA $C08C,X   the code in the disk controller
0262     10 FB   BPL $025F     ROM routine at $C65C, but
0264     49 D5   EOR #$D5      looking for an address
0266     D0 F7   BNE $025F     prologue of "D5 AA B5" instead
0268  BD 8C C0   LDA $C08C,X   of "D5 AA 96") and using the
026B     10 FB   BPL $0268     nibble translation table we set
026D     C9 AA   CMP #$AA      up earlier at $0800
026F     D0 F3   BNE $0264
0271        EA   NOP
0272  BD 8C C0   LDA $C08C,X
0275     10 FB   BPL $0272

0277     C9 B5   CMP #$B5      #$B5 for third prologue
0279     F0 09   BEQ $0284     nibble
027B        28   PLP
027C     90 DF   BCC $025D
027E     49 AD   EOR #$AD
0280     F0 1F   BEQ $02A1
0282     D0 D9   BNE $025D
0284     A0 03   LDY #$03
0286     84 2A   STY $2A
0288  BD 8C C0   LDA $C08C,X
028B     10 FB   BPL $0288
028D        2A   ROL
028E     85 3C   STA $3C
0290  BD 8C C0   LDA $C08C,X
0293     10 FB   BPL $0290
0295     25 3C   AND $3C
0297        88   DEY
0298     D0 EE   BNE $0288
029A        28   PLP
029B     C5 3D   CMP $3D
029D     D0 BE   BNE $025D
029F     B0 BD   BCS $025E
02A1     A0 9A   LDY #$9A
02A3     84 3C   STY $3C
02A5  BC 8C C0   LDY $C08C,X
02A8     10 FB   BPL $02A5
```

[21] If you replace the words "need to" with the words "get to," life becomes amazing.

```
02AA  59 00 08   EOR $0800,Y   use the nibble translation
02AD     A4 3C   LDY $3C       table we set up earlier to
02AF        88   DEY           convert nibbles on disk into
02B0  99 00 08   STA $0800,Y   bytes in memory
02B3     D0 EE   BNE $02A3
02B5     84 3C   STY $3C
02B7  BC 8C C0   LDY $C08C,X
02BA     10 FB   BPL $02B7
02BC  59 00 08   EOR $0800,Y
02BF     A4 3C   LDY $3C

02C1     91 26   STA ($26),Y   store the converted bytes at
02C3        C8   INY           $0300
02C4     D0 EF   BNE $02B5

02C6  BC 8C C0   LDY $C08C,X   verify the data with a
02C9     10 FB   BPL $02C6     one-nibble checksum
02CB  59 00 08   EOR $0800,Y
02CE     D0 8D   BNE $025D
02D0        60   RTS
```

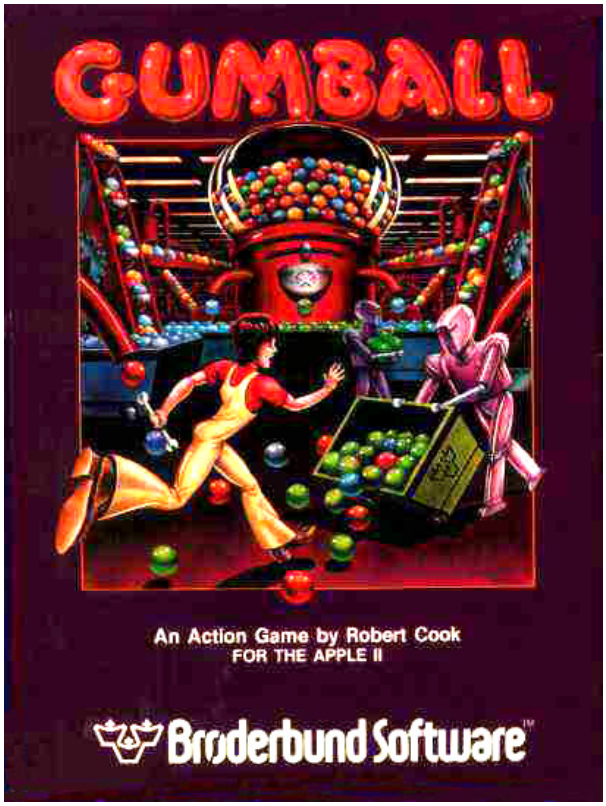
Continuing from $0234...

```
*234L
0234  20 D1 02   JSR $02D1
*2D1L
02D1        A8   TAY           finish decoding nibbles
02D2     A2 00   LDX #$00
02D4  B9 00 08   LDA $0800,Y
02D7        4A   LSR
02D8  3E CC 03   ROL $03CC,X
02DB        4A   LSR
02DC  3E 99 03   ROL $0399,X
02DF     85 3C   STA $3C
02E1     B1 26   LDA ($26),Y
02E3        0A   ASL
02E4        0A   ASL
02E5        0A   ASL
02E6     05 3C   ORA $3C
02E8     91 26   STA ($26),Y
02EA        C8   INY
02EB        E8   INX
02EC     E0 33   CPX #$33
02EE     D0 E4   BNE $02D4
02F0     C6 2A   DEC $2A
02F2     D0 DE   BNE $02D2

02F4  CC 00 03   CPY $0300     verify final checksum
02F7     D0 03   BNE $02FC

02F9        60   RTS           checksum passed, return to
                               caller and continue with the
                               boot process

02FC  4C 2D FF   JMP $FF2D     checksum failed, print "ERR"
                               and exit
```

Continuing from $0237...

```
0237  4C 01 03   JMP $0301     jump into the code we just
                               read
```

This is where I get to interrupt the boot, before it jumps to $0301.



In Which We Do A Bellyflop Into A Decrypted Stack And Discover That I Am Very Bad At Metaphors

```
*9600<C600.C6FFM

96F8     A9 05   LDA #$05      patch boot0 so it calls my
96FA  8D 38 08   STA $0838     routine instead of jumping to
96FD     A9 97   LDA #$97      $0301
96FF  8D 39 08   STA $0839

9702  4C 01 08   JMP $0801     start the boot

9705     A0 00   LDY #$00      (callback is here) copy the
9707  B9 00 03   LDA $0300,Y   code at $0300 to higher
970A  99 00 23   STA $2300,Y   memory so it survives a
970D        C8   INY           reboot
970E     D0 F7   BNE $9707

9710  AD E8 C0   LDA $C0E8     turn off slot 6 drive motor
9713  4C 00 C5   JMP $C500     and reboot to my work disk
*BSAVE TRACE,A$9600,L$116      in slot 5
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0300-03FF,A$2300,L$100
]CALL -151
*2301L
2301     84 48   STY $48
```

```
2303     A0 00   LDY #$00      clear hi-res graphics screen 2
2305        98   TYA
2306     A2 20   LDX #$20
2308  99 00 40   STA $4000,Y
230B        C8   INY
230C     D0 FA   BNE $2308
230E  EE 0A 03   INC $030A
2311        CA   DEX
2312     D0 F4   BNE $2308

2314  AD 57 C0   LDA $C057     and show it (appears blank)
2317  AD 52 C0   LDA $C052
231A  AD 55 C0   LDA $C055
231D  AD 50 C0   LDA $C050

2320  B9 00 03   LDA $0300,Y   decrypt the rest of this page
2323     45 48   EOR $48       to the stack page at $0100
2325  99 00 01   STA $0100,Y
2328        C8   INY
2329     D0 F5   BNE $2320

232B     A2 CF   LDX #$CF      set the stack pointer
232D        9A   TXS

232E        60   RTS           and exit via RTS
```

```
*9600<C600.C6FFM
96F8     A9 05   LDA #$05      patch boot0 so it calls my
96FA  8D 38 08   STA $0838     routine instead of jumping to
96FD     A9 97   LDA #$97      $0301
96FF  8D 39 08   STA $0839

9702  4C 01 08   JMP $0801     start the boot

9705     A0 00   LDY #$00      (callback is here) copy the
9707  B9 00 03   LDA $0300,Y   code at $0300 to higher
970A  99 00 23   STA $2300,Y   memory so it survives a
970D        C8   INY           reboot
970E     D0 F7   BNE $9707

9710  AD E8 C0   LDA $C0E8     turn off slot 6 drive motor
9713  4C 00 C5   JMP $C500     and reboot to my work disk
                               in slot 5
*BSAVE TRACE,A$9600,L$116
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0300-03FF,A$2300,L$100
]CALL -151
*2301L
2301     84 48   STY $48

2303     A0 00   LDY #$00      clear hi-res graphics screen 2
2305        98   TYA
2306     A2 20   LDX #$20
2308  99 00 40   STA $4000,Y
230B        C8   INY
230C     D0 FA   BNE $2308
230E  EE 0A 03   INC $030A
2311        CA   DEX
2312     D0 F4   BNE $2308

2314  AD 57 C0   LDA $C057     and show it (appears blank)
2317  AD 52 C0   LDA $C052
231A  AD 55 C0   LDA $C055
231D  AD 50 C0   LDA $C050
```

```
2320  B9 00 03   LDA $0300,Y   decrypt the rest of this page
2323     45 48   EOR $48       to the stack page at $0100
2325  99 00 01   STA $0100,Y
2328        C8   INY
2329     D0 F5   BNE $2320

232B     A2 CF   LDX #$CF      set the stack pointer
232D        9A   TXS

232E        60   RTS           and exit via RTS
```

Oh joy, stack manipulation. The stack on an Apple II is just $100 bytes in main memory ($0100..$01FF) and a single byte register that serves as an index into that page. This allows for all manner of mischief—overwriting the stack page (as we're doing here), manually changing the stack pointer (also doing that here), or even putting executable code directly on the stack.

The upshot is that I have no idea where execution continues next, because I don't know what ends up on the stack page. I get to interrupt the boot again to see the decrypted data that ends up at $0100.

Mischief Managed

```
*BLOAD TRACE
[first part is the same as the
 previous trace]
9705     84 48   STY $48       reproduce the decryption
9707     A0 00   LDY #$00      loop, but store the result at
9709  B9 00 03   LDA $0300,Y   $2100 so it survives a reboot
970C     45 48   EOR $48
970E  99 00 21   STA $2100,Y
9711        C8   INY
9712     D0 F5   BNE $9709

9714  AD E8 C0   LDA $C0E8     turn off drive motor and
9717  4C 00 C5   JMP $C500     reboot to my work disk

*BSAVE TRACE2,A$9600,L$11A
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0100-01FF,A$2100,L$100
]CALL -151
```

The original code at $0300 manually reset the stack pointer to #$CF and exited via RTS. The Apple II will increment the stack pointer before using it as an index into $0100 to get the next address. (For reasons I won't get into here, it also increments the address before passing execution to it.)

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
     next return address
```

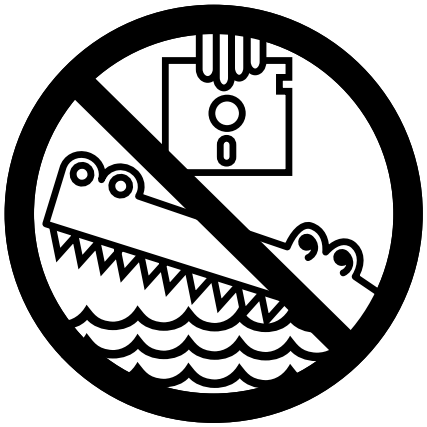
$012F + 1 = $0130, which is already in memory at $2130.

Oh joy. Code on the stack. (Remember, the "stack" is just a page in main memory. If you want to use that page for something else, it's up to you to ensure that it doesn't conflict with the stack functioning as a stack.)

```
*2130L
2130     A2 04   LDX #$04
2132     86 86   STX $86
2134     A0 00   LDY #$00
2136     84 83   STY $83
2138     86 84   STX $84
```

Now ($83) points to $0400.

```
213A     A6 2B   LDX $2B        get slot number (x16)

213C  BD 8C C0   LDA $C08C,X    find a 3-nibble prologue ("BF
213F     10 FB   BPL $213C      D7 D5")
2141     C9 BF   CMP #$BF
2143     D0 F7   BNE $213C
2145  BD 8C C0   LDA $C08C,X
2148     10 FB   BPL $2145
214A     C9 D7   CMP #$D7
214C     D0 F3   BNE $2141
214E  BD 8C C0   LDA $C08C,X
2151     10 FB   BPL $214E
2153     C9 D5   CMP #$D5
2155     D0 F3   BNE $214A

2157  BD 8C C0   LDA $C08C,X   read 4-4-encoded data
215A     10 FB   BPL $2157
215C        2A   ROL
215D     85 85   STA $85
215F  BD 8C C0   LDA $C08C,X
2162     10 FB   BPL $215F
2164     25 85   AND $85

2166     91 83   STA ($83),Y   store in $0400 (text page, but
2168        C8   INY           it's hidden right now because
2169     D0 EC   BNE $2157     we switched to hi-res graphics
                               screen 2 at $0314)
216B  0E 00 C0   ASL $C000     find a 1-nibble epilogue ("D4")
216E  BD 8C C0   LDA $C08C,X
2171     10 FB   BPL $216E
2173     C9 D4   CMP #$D4
2175     D0 B9   BNE $2130

2177     E6 84   INC $84       increment target memory
                               page
2179     C6 86   DEC $86       decrement sector count
217B     D0 DA   BNE $2157     (initialized at $0132)

217D        60   RTS           exit via RTS
```



Wait, what? Ah, we're using the same trick we used to call this routine—the stack has been prefilled with a series of "return" addresses. It's time to "return" to the next one.

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
     next return address
```

$03FF + 1 = $0400, and that's where I get to interrupt the boot.

Seek And Ye Shall Find

```
*BLOAD TRACE2
.
.  [same as previous trace]
.
9705     84 48   STY $48        reproduce the decryption loop
9707     A0 00   LDY #$00       that was originally at $0320
9709  B9 00 03   LDA $0300,Y
970C     45 48   EOR $48
970E  99 00 01   STA $0100,Y
9711        C8   INY
9712     D0 F5   BNE $9709

9714     A9 21   LDA #$21       now that the stack is in place
9716  8D D2 01   STA $01D2       at $0100, change the first
9719     A9 97   LDA #$97       return address so it points to
971B  8D D3 01   STA $01D3       a callback under my control
                                (instead of continuing to
                                $0400)
971E     A2 CF   LDX #$CF       continue the boot
9720        9A   TXS
9721        60   RTS

9722     A2 04   LDX #$04       (callback is here) copy the
9724     A0 00   LDY #$00       contents of the text page to
9726  B9 00 04   LDA $0400,Y    higher memory
9729  99 00 24   STA $2400,Y
972C        C8   INY
972D     D0 F7   BNE $9726
972F  EE 28 97   INC $9728
9732  EE 2B 97   INC $972B
9735        CA   DEX
9736     D0 EE   BNE $9726
```

```
9738  AD E8 C0   LDA $C0E8     turn off the drive and reboot
973B  4C 00 C5   JMP $C500     to my work disk

*BSAVE TRACE3,A$9600,L$13E
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0400-07FF,A$2400,L$400
]CALL -151
```

I'm going to leave this code at $2400, since I can't put it on the text page and examine it at the same time. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2400L
2400     A0 00   LDY #$00      copy three pages to the top of
2402  B9 00 05   LDA $0500,Y   main memory
2405  99 00 BD   STA $BD00,Y
2408  B9 00 06   LDA $0600,Y
240B  99 00 BE   STA $BE00,Y
240E  B9 00 07   LDA $0700,Y
2411  99 00 BF   STA $BF00,Y
2414     C8      INY
2415     D0 EB   BNE $2402
```

I can replicate that.

```
*FE89G FE93G ; disconnect DOS
*BD00<2500.27FFM ; simulate
copy loop
2417     A6 2B   LDX $2B
2419  8E 66 BF   STX $BF66
241C  20 48 BF   JSR $BF48

*BF48L
BF48  AD 81 C0   LDA $C081     zap contents of language card
BF4B  AD 81 C0   LDA $C081
BF4E     A0 00   LDY #$00
BF50     A9 D0   LDA #$D0
BF52     84 A0   STY $A0
BF54     85 A1   STA $A1
BF56     B1 A0   LDA ($A0),Y
BF58     91 A0   STA ($A0),Y
BF5A     C8      INY
BF5B     D0 F9   BNE $BF56
BF5D     E6 A1   INC $A1
BF5F     D0 F5   BNE $BF56
BF61  2C 80 C0   BIT $C080
BF64     60      RTS
```

Continuing from $041F...

```
241F  AD 83 C0   LDA $C083     set low-level reset vectors and
2422  AD 83 C0   LDA $C083     page 3 vectors to point to
2425     A0 00   LDY #$00      $BF00—presumably The
2427     A9 BF   LDA #$BF      Badlands (from which there is
2429  8C FC FF   STY $FFFC     no return)
242C  8D FD FF   STA $FFFD
242F  8C F2 03   STY $03F2
2432  8D F3 03   STA $03F3
2435     A0 03   LDY #$03
2437  8C F0 03   STY $03F0
243A  8D F1 03   STA $03F1
243D     84 38   STY $38
243F     85 39   STA $39
2441     49 A5   EOR #$A5
2443  8D F4 03   STA $03F4

*BF00L
```

```
BF00     A9 D2   LDA #$D2      There are multiple entry
BF02  2C A9 D0   BIT $D0A9     points here: $BF00, $BF03,
BF05  2C A9 CC   BIT $CCA9     $BF06, and $BF09 (hidden in
BF08  2C A9 A1   BIT $A1A9     this listing by the "BIT"
BF0B     48      PHA           opcodes).

BF0C  20 48 BF   JSR $BF48     zap the language card again

BF0F  20 2F FB   JSR $FB2F     TEXT/HOME/NORMAL
BF12  20 58 FC   JSR $FC58
BF15  20 84 FE   JSR $FE84

BF18     68      PLA           Depending on the initial entry
BF19  8D 00 04   STA $0400     point, this displays a different
                               character in the top left
                               corner of the screen

BF1C     A0 00   LDY #$00      now wipe all of main memory
BF1E     98      TYA
BF1F  99 00 BE   STA $BE00,Y
BF22     C8      INY
BF23     D0 FA   BNE $BF1F
BF25  CE 21 BF   DEC $BF21

BF28  2C 30 C0   BIT $C030     while playing a sound
BF2B  AD 21 BF   LDA $BF21
BF2E     C9 08   CMP #$08
BF30     B0 EA   BCS $BF1C

BF32  8D F3 03   STA $03F3     munge the reset vector
BF35  8D F4 03   STA $03F4

BF38  AD 66 BF   LDA $BF66     and reboot from whence we
BF3B     4A      LSR           came
BF3C     4A      LSR
BF3D     4A      LSR
BF3E     4A      LSR
BF3F     09 C0   ORA #$C0
BF41     E9 00   SBC #$00
BF43     48      PHA
BF44     A9 FF   LDA #$FF
BF46     48      PHA
BF47     60      RTS
```

Yeah, let's try not to end up there.

Continuing from $0446...

```
2446     A9 07   LDA #$07
2448  20 00 BE   JSR $BE00

*BE00L

BE00     A2 13   LDX #$13      entry point #1

BE02  2C A2 0A   BIT $0AA2     entry point #2 (hidden
                               behind a BIT opcode, but it's
                               "LDX #$0A")

BE05  8E 6E BE   STX $BE6E     (!) modify the code later
                               based on which entry point
                               we called
```

```
BE08  8D 90 BE   STA $BE90     The rest of this routine is a
BE0B  CD 65 BF   CMP $BF65     garden variety drive seek. The
BE0E     F0 59   BEQ $BE69     target phase (track x 2) is in
BE10     A9 00   LDA #$00      the accumulator on entry.
BE12  8D 91 BE   STA $BE91
BE15  AD 65 BF   LDA $BF65
BE18  8D 92 BE   STA $BE92
BE1B        38   SEC
BE1C  ED 90 BE   SBC $BE90
BE1F     F0 37   BEQ $BE58
BE21     B0 07   BCS $BE2A
BE23     49 FF   EOR #$FF
BE25  EE 65 BF   INC $BF65
BE28     90 05   BCC $BE2F
BE2A     69 FE   ADC #$FE
BE2C  CE 65 BF   DEC $BF65
BE2F  CD 91 BE   CMP $BE91
BE32     90 03   BCC $BE37
BE34  AD 91 BE   LDA $BE91
BE37     C9 0C   CMP #$0C
BE39     B0 01   BCS $BE3C
BE3B        A8   TAY
BE3C        38   SEC
BE3D  20 5C BE   JSR $BE5C
BE40  B9 78 BE   LDA $BE78,Y
BE43  20 6D BE   JSR $BE6D
BE46  AD 92 BE   LDA $BE92
BE49        18   CLC
BE4A  20 5F BE   JSR $BE5F
BE4D  B9 84 BE   LDA $BE84,Y
BE50  20 6D BE   JSR $BE6D
BE53  EE 91 BE   INC $BE91
BE56     D0 BD   BNE $BE15
BE58  20 6D BE   JSR $BE6D
BE5B        18   CLC
BE5C  AD 65 BF   LDA $BF65
BE5F     29 03   AND #$03
BE61        2A   ROL
BE62  0D 66 BF   ORA $BF66
BE65        AA   TAX
BE66  BD 80 C0   LDA $C080,X
BE69  AE 66 BF   LDX $BF66
BE6C        60   RTS


BE6D     A2 13   LDX #$13      (value of X may be modified
BE6F        CA   DEX           depending on which entry
BE70     D0 FD   BNE $BE6F     point was called)
BE72        38   SEC
BE73     E9 01   SBC #$01
BE75     D0 F6   BNE $BE6D
BE77        60   RTS
BE78 [01 30 28 24 20 1E 1D 1C]
BE80 [1C 1C 1C 1C 70 2C 26 22]
BE88 [1F 1E 1D 1C 1C 1C 1C 1C]
```

The fact that there are two entry points is interesting. Calling `$BE00` will set X to `#$13`, which will end up in `$BE6E`, so the wait routine at `$BE6D` will wait long enough to go to the next phase (a.k.a. half a track). Nothing unusual there; that's how all drive seek routines work. But calling `$BE03` instead of `$BE00` will set `X` to `#$0A`, which will make the wait routine burn fewer CPU cycles while the drive head is moving, so it will only move half a phase (a.k.a. a quarter track). That is potentially very interesting.

Continuing from `$044B`...

```
244B     A9 05   LDA #$05
244D     85 33   STA $33
244F     A2 03   LDX #$03
2451     86 36   STX $36
2453     A0 00   LDY #$00
2455     A5 33   LDA $33
2457     84 34   STY $34
2459     85 35   STA $35
```

Now (`$34`) points to `$0500`.

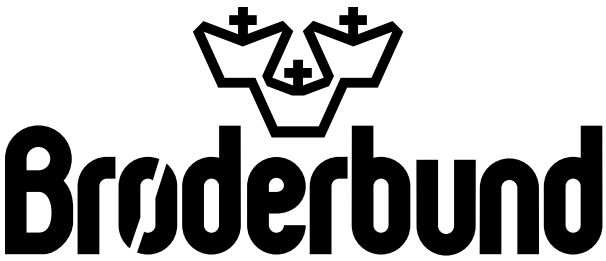
```
245B  AE 66 BF   LDX $BF66     find a 3-nibble prologue ("B5
245E  BD 8C C0   LDA $C08C,X   DE F7")
2461     10 FB   BPL $245E
2463     C9 B5   CMP #$B5
2465     D0 F7   BNE $245E
2467  BD 8C C0   LDA $C08C,X
246A     10 FB   BPL $2467
246C     C9 DE   CMP #$DE
246E     D0 F3   BNE $2463
2470  BD 8C C0   LDA $C08C,X
2473     10 FB   BPL $2470
2475     C9 F7   CMP #$F7
2477     D0 F3   BNE $246C


2479  BD 8C C0   LDA $C08C,X   read 4-4-encoded data into
247C     10 FB   BPL $2479     $0500+
247E        2A   ROL
247F     85 37   STA $37
2481  BD 8C C0   LDA $C08C,X
2484     10 FB   BPL $2481
2486     25 37   AND $37
2488     91 34   STA ($34),Y
248A        C8   INY
248B     D0 EC   BNE $2479
248B     D0 EC   BNE $2479
248D  0E FF FF   ASL $FFFF


2490  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue ("D5")
2493     10 FB   BPL $2490
2495     C9 D5   CMP #$D5
2497     D0 B6   BNE $244F
2499     E6 35   INC $35


249B     C6 36   DEC $36       3 sectors (initialized at $0451)
249D     D0 DA   BNE $2479


249F        60   RTS           and exit via RTS
```

We've read 3 more sectors into `$0500+`, overwriting the code we read earlier (but moved to `$BD00+`), and once again we simply exit and let the stack tell us where we're going next.

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
              next return address
```

`$04FF + 1 = $0500`, the code we just read.

And that's where I get to interrupt the boot.



Return of the Jedi

```
*C500G                          reboot because I disconnected
...                             and overwrote DOS to
]CALL -151                      examine the previous code
*BLOAD TRACE3                   chunk at $BD00+
.
.  [same as previous trace]
.
9714     A9 21    LDA #$21      Patch the stack again, but
9716  8D D4 01    STA $01D4     slightly later, at $01D4. (The
9719     A9 97    LDA #$97      previous trace patched it at
971B  8D D5 01    STA $01D5     $01D2.)

971E     A2 CF    LDX #$CF      continue the boot
9720        9A    TXS
9721        60    RTS

9722     A2 04    LDX #$03      (callback is here) We just
9724     A0 00    LDY #$00      executed all the code up to
9726  B9 00 05    LDA $0500,Y   and including the "RTS" at
9729  99 00 25    STA $2500,Y   $049F, so now let's copy the
972C        C8    INY           latest code at $0500..$07FF to
972D     D0 F7    BNE $9726     higher memory so it survives
972F  EE 28 97    INC $9728     a reboot.
9732  EE 2B 97    INC $972B
9735        CA    DEX
9736     D0 EE    BNE $9726

9738  AD E8 C0    LDA $C0E8     reboot to my work disk
973B  4C 00 C5    JMP $C500

*BSAVE TRACE4,A$9600,L$13E
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT2
0500-07FF,A$2500,L$300
]CALL -151
```

Again, I'm going to leave this at $2500 because I can't examine code on the text page. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2500L
2500     A9 02    LDA #$02      seek to track 1
2502  20 00 BE    JSR $BE00

2505  AE 66 BF    LDX $BF66     get slot number x16 (set a
2508     A0 00    LDY #$00      long time ago, at $0419)
250A     A9 20    LDA #$20
250C     85 30    STA $30
250E        88    DEY
250F     D0 04    BNE $2515
2511     C6 30    DEC $30
2513     F0 3C    BEQ $2551

2515  BD 8C C0    LDA $C08C,X   find a 3-nibble prologue ("D5
2518     10 FB    BPL $2515     FF DD")
251A     C9 D5    CMP #$D5
251C     D0 F0    BNE $250E
251E  BD 8C C0    LDA $C08C,X
2521     10 FB    BPL $251E
2523     C9 FF    CMP #$FF
2525     D0 F3    BNE $251A
2527  BD 8C C0    LDA $C08C,X
252A     10 FB    BPL $2527
252C     C9 DD    CMP #$DD
252E     D0 F3    BNE $2523

2530     A0 00    LDY #$00      read 4-4-encoded data
2532  BD 8C C0    LDA $C08C,X
2535     10 FB    BPL $2532
2537        38    SEC
2538        2A    ROL
2539     85 30    STA $30
253B  BD 8C C0    LDA $C08C,X
253E     10 FB    BPL $253B
2540     25 30    AND $30

2542  99 00 B0    STA $B000,Y   into $B000 (hard-coded here,
2545        C8    INY           was not modified earlier
2546     D0 EA    BNE $2532     unless I missed something)

2548  BD 8C C0    LDA $C08C,X   find a 1-nibble epilogue ("D5")
254B     10 FB    BPL $2548
254D     C9 D5    CMP #$D5
254F     F0 0B    BEQ $255C

2551     A0 00    LDY #$00      This is odd. If the epilogue
2553  B9 00 07    LDA $0700,Y   doesn't match, it's not an
2556  99 00 B0    STA $B000,Y   error. Instead, it appears that
2559        C8    INY           we simply copy a page of data
255A     D0 F7    BNE $2553     that we read earlier (at
                                $0700).

255C  20 F0 05    JSR $05F0     execution continues here
                                regardless
*25F0L
25F0     A0 56    LDY #$56      Weird, but OK. This ends up
25F2     A9 BD    LDA #$BD      calling $BE00 with A=$07,
25F4        48    PHA           which will seek to track 3.5.
25F5     A9 FF    LDA #$FF
25F7        48    PHA
25F8     A9 07    LDA #$07
25FA        60    RTS
```

And now we're on half tracks.

Continuing from $055F...

```
255F  BD 8C C0    LDA $C08C,X   find a 3-nibble prologue ("DD
2562     10 FB    BPL $255F     EF AD")
2564     C9 DD    CMP #$DD
2566     D0 F7    BNE $255F
2568  BD 8C C0    LDA $C08C,X
256B     10 FB    BPL $2568
256D     C9 EF    CMP #$EF
256F     D0 F3    BNE $2564
2571  BD 8C C0    LDA $C08C,X
2574     10 FB    BPL $2571
2576     C9 AD    CMP #$AD
2578     D0 F3    BNE $256D
```

```
257A     A0 00   LDY #$00      read a 4-4 encoded byte (two
257C  BD 8C C0   LDA $C08C,X   nibbles on disk = 1 byte in
257F     10 FB   BPL $257C     memory)
2581        38   SEC
2582        2A   ROL
2583     85 00   STA $00
2585  BD 8C C0   LDA $C08C,X
2588     10 FB   BPL $2585
258A     25 00   AND $00

258C        48   PHA           push the byte to the stack
                               (WTF?)
258D        88   DEY           repeat for $100 bytes
258E     D0 EC   BNE $257C

2590  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue
2593     10 FB   BPL $2590     ("D5")
2595     C9 D5   CMP #$D5
2597     D0 C3   BNE $255C

2599  CE 9C 05   DEC $059C (!)
259C     61 00   ADC ($00,X)
```

(!) Self-modifying code alert! WOO WOO. I'll use this symbol whenever one instruction modifies the next instruction. When this happens, the disassembly listing is misleading because the opcode will be changed by the time the second instruction is executed.

In this case, the DEC at $0599 modifies the opcode at $059C, so that's not really an "ADC." By the time we execute the instruction at $059C, it will have been decremented to #$60, a.k.a. "RTS."

One other thing: we've read $100 bytes and pushed all of them to the stack. The stack is only $100 bytes ($0100..$01FF), so this completely obliterates any previous values.

We haven't changed the stack pointer, though. That means the "RTS" at $059C will still look at $01D6 to find the next "return" address. That used to be ''4F 04'', but now it's been overwritten with new values, along with the rest of the stack. That's some serious Jedi mind trick stuff.

"These aren't the return addresses you're looking for."

"These aren't the return addresses we're looking for."

"He can go about his bootloader."

"You can go about your bootloader."

"Move along."

"Move along... move along."

In Which We Move Along

Luckily, there's plenty of room at $0599. I can insert a JMP to call back to code under my control, where I can save a copy of the stack. (And $B000 as well, whatever that is.) I get to ensure I don't disturb the stack before I save it, so no JSR, PHA, PHP, or TXS. I think I can manage that. JMP doesn't disturb the stack, so that's safe for the callback.

```
*BLOAD TRACE4
.
.  [same as previous trace]
.
9722     A9 4C   LDA #$4C      set up a JMP $9734 at $0599
9724  8D 99 05   STA $0599
9727     A9 34   LDA #$34
9729  8D 9A 05   STA $059A
972C     A9 97   LDA #$97
972E  8D 9B 05   STA $059B

9731  4C 00 05   JMP $0500     continue the boot

9734     A0 00   LDY #$00      (callback is here) Copy $B000
9736  B9 00 B0   LDA $B000,Y   and $0100 to higher memory
9739  99 00 20   STA $2000,Y   so they survive a reboot
973C  B9 00 01   LDA $0100,Y
973F  99 00 21   STA $2100,Y
9742        C8   INY
9743     D0 F1   BNE $9736

9745  AD E8 C0   LDA $C0E8     reboot to my work disk
9748  4C 00 C5   JMP $C500

*BSAVE TRACE5,A$9600,L$14B
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT2
B000-B0FF,A$2000,L$100
]BSAVE BOOT2
0100-01FF,A$2100,L$100
]CALL -151
```

Remember, the stack *pointer* hasn't changed. Now that I have the new stack *data*, I can just look at the right index in the captured stack page to see where the bootloader continues once it issues the "RTS" at $059C.

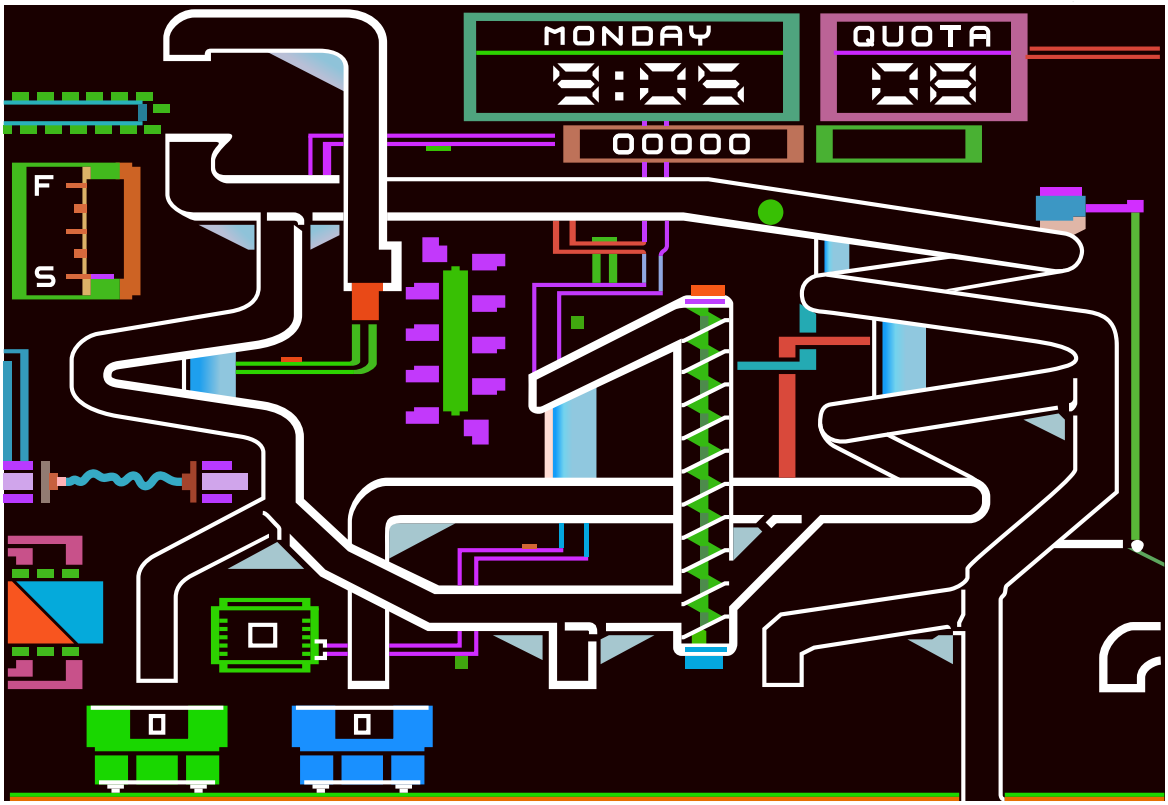
```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
                       ^^^^^
                       next return address
```

That's part of the stack page I just captured, so it's already in memory.

```
*2126L
```

Another disk read routine! The fourth? Fifth? I've truly lost count.

```
2126  BD 8C C0   LDA $C08C,X   find a 3-nibble prologue ("BF
2129     10 FB   BPL $2126     BE D4")
212B     C9 BF   CMP #$BF
212D     D0 F7   BNE $2126
212F  BD 8C C0   LDA $C08C,X
2132     10 FB   BPL $212F
2134     C9 BE   CMP #$BE
2136     D0 F3   BNE $212B
2138  BD 8C C0   LDA $C08C,X
213B     10 FB   BPL $2138
213D     C9 D4   CMP #$D4
213F     D0 F3   BNE $2134
```



```
2141     A0 00   LDY #$00      read 4-4-encoded data
2143  BD 8C C0   LDA $C08C,X
2146     10 FB   BPL $2143
2148        38   SEC
2149        2A   ROL
214A  8D 00 02   STA $0200
214D  BD 8C C0   LDA $C08C,X
2150     10 FB   BPL $214D
2152  2D 00 02   AND $0200

2155  59 00 01   EOR $0100,Y   decrypt the data from disk by
                               using this entire page of code
                               (in the stack page) as the
                               decryption key (more on this
                               later)
2158  99 00 00   STA $0000,Y   and store it in zero page
215B        C8   INY
215C     D0 E5   BNE $2143

215E  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue
2161     10 FB   BPL $215E     ("D5")
2163     C9 D5   CMP #$D5
2165     D0 BF   BNE $2126

2167        60   RTS           and exit via RTS
```

And we're back on the stack again.

```
*21D0.
21D0 F0 78 AD D8 02 85 25 01
21D8 57 FF 57 FF 57 FF 57 FF
21E0 57 FF 22 01 FF 05 B1 4C
```

The six `57 FF` words and the following `22 01` word are the next return addresses.

`$FF57` + 1 = `$FF58`, which is a well-known address in ROM that is always an "RTS" instruction. So this will burn through several return addresses on the stack in short order, then finally arrive at `$0123`, in memory at `$2123`.

```
*2123L
2123  6C 28 00   JMP ($0028)
```

...which is in the new zero page that was just read from disk.

And to think, we've loaded basically nothing of consequence yet. The screen is still black. We have 3 pages of code at `$BD00..$BFFF`. There's still some code on the text screen, but who knows if we'll ever call it again. Now we're off to zero page for some reason.

Un. Be. Lievable.

By Perseverance The Snail Reached The Ark

I can't touch the code on the stack, because it's used as a decryption key. I mean, I could theoretically change a few bytes of it, then calculate the proper decrypted bytes on zero page by hand. But no.

Instead, I'm just going to copy this latest disk routine wholesale. It's short and has no external dependencies, so why not? Then I can capture the decrypted zero page and see where that `JMP ($0028)` is headed.

```
*BLOAD TRACE5
*9734<2126.2166M
```

Here's the entire disassembly listing of boot trace #6:

```
96F8     A9 05   LDA #$05      patch boot0 so it calls my
96FA  8D 38 08   STA $0838     routine instead of jumping to
96FD     A9 97   LDA #$97      $0301
96FF  8D 39 08   STA $0839

9702  4C 01 08   JMP $0801     start the boot

9705     84 48   STY $48       (callback #1 is here)
9707     A0 00   LDY #$00      reproduce the decryption loop
9709  B9 00 03   LDA $0300,Y   that was originally at $0320
970C     45 48   EOR $48
970E  99 00 01   STA $0100,Y
9711        C8   INY
9712     D0 F5   BNE $9709

9714     A9 21   LDA #$21      patch the stack so it jumps to
9716  8D D4 01   STA $01D4     my callback #2 instead of
9719     A9 97   LDA #$97      continuing to $0500
971B  8D D5 01   STA $01D5

971E     A2 CF   LDX #$CF      continue the boot
9720        9A   TXS
9721        60   RTS

9722     A9 4C   LDA #$4C      (callback #2) set up callback
9724  8D 99 05   STA $0599     #3 instead of passing control
9727     A9 34   LDA #$34      to the disk read routine at
9729  8D 9A 05   STA $059A     $0126
972C     A9 97   LDA #$97
972E  8D 9B 05   STA $059B

9731  4C 00 05   JMP $0500     continue the boot

9734  BD 8C C0   LDA $C08C,X   (callback #3) disk read
9737     10 FB   BPL $9734     routine copied wholesale from
9739     C9 BF   CMP #$BF      $0126..$0166 that reads a
973B     D0 F7   BNE $9734     sector and decrypts it into
973D  BD 8C C0   LDA $C08C,X   zero page
9740     10 FB   BPL $973D
9742     C9 BE   CMP #$BE
9744     D0 F3   BNE $9739
9746  BD 8C C0   LDA $C08C,X
9749     10 FB   BPL $9746
974B     C9 D4   CMP #$D4
974D     D0 F3   BNE $9742
974F     A0 00   LDY #$00
9751  BD 8C C0   LDA $C08C,X
9754     10 FB   BPL $9751
9756        38   SEC
9757        2A   ROL
9758  8D 00 02   STA $0200
975B  BD 8C C0   LDA $C08C,X
975E     10 FB   BPL $975B
9760  2D 00 02   AND $0200
9763  59 00 01   EOR $0100,Y
9766  99 00 00   STA $0000,Y
9769        C8   INY
976A     D0 E5   BNE $9751
976C  BD 8C C0   LDA $C08C,X
976F     10 FB   BPL $976C
9771     C9 D5   CMP #$D5
9773     D0 BF   BNE $9734
```

```
 execution falls through here
9775     A0 00    LDY #$00      now capture the decrypted
9777  B9 00 00    LDA $0000,Y   zero page
977A  99 00 20    STA $2000,Y
977D        C8    INY
977E     D0 F7    BNE $9777

9780  AD E8 C0    LDA $C0E8     turn off the slot 6 drive motor

9783  4C 00 C5    JMP $C500     reboot to my work disk

*BSAVE TRACE6,A$9600,L$186

                                Whew. Let's do it.
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT3
0000-00FF,A$2000,L$100
]CALL -151
*2028.2029
2028 D0 06
```

OK, the `JMP ($0028)` points to `$06D0`, which I captured earlier. It's part of the second chunk we read into the text page. (Not the first chunk—that was copied to `$BD00+` then overwritten.) So it's in the "`BOOT2 0500-07FF`" file, not the "`BOOT1 0400-07FF`" file.

```
*BLOAD BOOT2 0500-07FF,A$2500
*26D0L
26D0     A2 00    LDX #$00
26D2  EE D5 06    INC $06D5 (!)
26D5     C9 EE    CMP #$EE
```

Oh joy, more self-modifying code.

```
*26D5:CA
*26D5L
26D5        CA    DEX
26D6  EE D9 06    INC $06D9 (!)
26D9        0F    ???

*26D9:10
*26D9L
26D9     10 FB    BPL $26D6     branch is never taken,
26DB  CE DE 06    DEC $06DE (!) because we just DEX'd from
26DE     61 A0    ADC ($A0,X)   #$00 to #$FF

*26DE:60
*26DEL
26DE        60    RTS
```

And now we're back on the stack.

```
*BLOAD BOOT2 0100-01FF,A$2100
*21E0.
*21E0. 57 FF 22 01 FF 05 B1 4C
                 next return address
```

`$05FF + 1 = $0600`, which is already in memory at `$2600`.

```
*2600L
2600     A0 00    LDY #$00      destroy stack by pushing the
2602        48    PHA           same value $100 times
2603        88    DEY
2604     D0 FC    BNE $2602
```

I guess we're done with all that code on the stack page. I mean, I hope we're done with it, since it all just disappeared.

```
2606     A2 FF    LDX #$FF      reset the stack pointer
2608        9A    TXS

2609  EE 0C 06    INC $060C (!)
260C        A8    TAY
```

Oh joy.

```
*260C:A9
*260CL
260C     A9 27    LDA #$27
260E  EE 11 06    INC $0611 (!)
2611        17    ???

*2611:18
*2611L
2611        18    CLC
2612  EE 15 06    INC $0615 (!)
2615        68    PLA

*2615:69
*2615L
2615     69 D9    ADC #$D9
2617  EE 1A 06    INC $061A (!)
261A        4B    ???

*261A:4C
*261AL
261A  4C 90 FD    JMP $FD90
```

Wait, what?

```
*FD90L
FD90     D0 5B    BNE $FDED
```

Despite the fact that the accumulator is `#$00` (because `#$27 + #$D9 = #$00`), the `INC` at `$0617` affects the Z register and causes this branch to be taken, because the final value of `$061A` was not zero.

```
*FDEDL
FDED  6C 36 00    JMP ($0036)
```

Of course, this is the standard output character routine, which routes through the output vector at `($0036)`. And we just set that vector, along with the rest of zero page. So what is it?

```
*2036.2037
2036 6F BF
```

Oh joy. Let's see, `$BD00..$BFFF` was copied earlier from `$0500..$07FF`, but from the first time we read into the text page, not the second time we read into text page. So it's in the "BOOT1 0400-07FF" file, not the "BOOT2 0500-07FF" file.

```
*BLOAD BOOT1 0400-07FF,A$2400
*FE89G FE93G                  disconnect DOS
```

```
*BD00<2500.27FFM                move code into place
*BF6FL
BF6F     C9 07   CMP #$07
BF71     90 03   BCC $BF76
BF73  6C 3A 00   JMP ($003A)

*203A.203B
203A F0 FD
BF76     85 5F   STA $5F        save input value

BF78        A8   TAY           use value as an index into an
BF79  B9 68 BF   LDA $BF68,Y    array

BF7C  8D 82 BF   STA $BF82      (!)self-modifying code
BF7F     A9 00   LDA #$00       alert—this changes the
BF81  20 D0 BE   JSR $BED0      upcoming JSR at $BF81
```

Amazing. So this "output" vector does actually print characters through the standard `$FDF0` text print routine, but only if the character to be printed is at least `#$07`. If it's less than `#$07`, the "character" is treated as a command. Each command gets routed to a different routine somewhere in `$BExx`. The low byte of each routine is stored in the array at `$BF68`, and the "STA" at `$BF7C` modifies the "JSR" at `$BF81` to call the appropriate address.

```
*BF68.
BF68 D0 DF D0 D0 FD FD D0
```

Since `A = #$00` this time, the call is unchanged and we `JSR $BED0`. Other input values may call `$BEDF` or `$BEFD` instead.

```
*BED0L
BED0     A5 60   LDA $60        use the "value" of $C050 to
BED2  4D 50 C0   EOR $C050      produce a pseudo-random
BED5     85 60   STA $60        number between #$01 and
BED7     29 0F   AND #$0F       #$0E

BED9     F0 F5   BEQ $BED0      not #$00

BEDB     C9 0F   CMP #$0F       not #$0F
BEDD     F0 F1   BEQ $BED0

BEDF  20 66 F8   JSR $F866      set the lo-res plotting color
                                (in zero page $30) to the
                                random-ish value we just
                                produced
BEE2     A9 17   LDA #$17       fill the lo-res graphics screen
BEE4        48   PHA            with blocks of that color

BEE5  20 47 F8   JSR $F847      calculates the base address for
BEE8     A0 27   LDY #$27       this line in memory and puts
BEEA     A5 30   LDA $30        it in $26/$27
BEEC     91 26   STA ($26),Y
BEEE        88   DEY
BEEF     10 FB   BPL $BEEC
BEF1        68   PLA

BEF2        38   SEC            do it for all 24 ($17) rows of
BEF3     E9 01   SBC #$01       the screen
BEF5     10 ED   BPL $BEE4

BEF7  AD 56 C0   LDA $C056      and switch to lo-res graphics
BEFA  AD 54 C0   LDA $C054      mode
BEFD        60   RTS
```

This explains why the original disk fills the screen with a different color every time it boots.

But wait, these commands do so much more than just fill the screen.

Continuing from `$BF84`...

```
BF84     A5 5F   LDA $5F
BF86     C9 04   CMP #$04
BF88     D0 03   BNE $BF8D
BF8A  4C 00 BD   JMP $BD00
```

If `A = #$04`, we exit via `$BD00`, which I'll investigate later.

```
BF8D     C9 05   CMP #$05
BF8F     D0 03   BNE $BF94
BF91  6C 82 BF   JMP ($BF82)
```

If `A = #$05`, we exit via `($BF82)`, which is the same thing we just called via the self-modified `JSR` at `$BF81`.

For all other values of A, we do this:

```
BF94  20 B0 BE   JSR $BEB0

*BEB0L
BEB0     A2 60   LDX #$60       another layer of encryption!
BEB2  BD 9F BF   LDA $BF9F,X
BEB5  5D 00 BE   EOR $BE00,X

BEB8  9D 9F BF   STA $BF9F,X    and it's decrypting the code
BEBB        CA   DEX            that we're about to run
BEBC     10 F4   BPL $BEB2
BEBE  AE 66 BF   LDX $BF66
BEC1        60   RTS
```

This is self-contained, so I can just run it right now and see what ends up at `$BF9F`.

```
*BEB0G
```

Continuing from `$BF97`...

```
BF97     A0 00   LDY #$00
BF99     A9 B2   LDA #$B2
BF9B     84 44   STY $44
BF9D     85 45   STA $45

BF9F  BD 89 C0   LDA $C089,X    everything beyond this point
                                was encrypted, but we just
                                decrypted it in $BEB0

BFA2  BD 8C C0   LDA $C08C,X    find a 3-nibble prologue
BFA5     10 FB   BPL $BFA2      (varies, based on whatever
BFA7     C5 40   CMP $40        the hell is in zero page
BFA9     D0 F7   BNE $BFA2      $40/$41/$42 at this point)
BFAB  BD 8C C0   LDA $C08C,X
BFAE     10 FB   BPL $BFAB
BFB0     C5 41   CMP $41
BFB2     D0 F3   BNE $BFA7
BFB4  BD 8C C0   LDA $C08C,X
BFB7     10 FB   BPL $BFB4
BFB9     C5 42   CMP $42
BFBB     D0 F3   BNE $BFB0
```

```
BFBD  BD 8C C0   LDA $C08C,X   read 4-4-encoded data
BFC0     10 FB   BPL $BFBD
BFC2        38   SEC
BFC3        2A   ROL
BFC4     85 46   STA $46
BFC6  BD 8C C0   LDA $C08C,X
BFC9     10 FB   BPL $BFC6
BFCB     25 46   AND $46

BFCD     91 44   STA ($44),Y  store in memory starting at
BFCF        C8   INY           $B200 (set at $BF9B)
BFD0     D0 EB   BNE $BFBD
BFD2     E6 45   INC $45
BFD4  BD 8C C0   LDA $C08C,X
BFD7     10 FB   BPL $BFD4
BFD9     C5 43   CMP $43
BFDB     D0 BA   BNE $BF97

BFDD     A5 45   LDA $45       read into $B200, $B300, and
BFDF     49 B5   EOR #$B5      $B400, then stop
BFE1     D0 DA   BNE $BFBD
BFE3        48   PHA ; A=00
BFE4     A5 45   LDA $45 ;
A=B5
BFE6     49 8E   EOR #$8E ;
A=3B
BFE8        48   PHA
BFE9        60   RTS
```

So we push `#$00` and `#$3B` to the stack, then exit via RTS. That will "return" to `$003C`, which is in memory at `$203C`.

```
*203CL
203C  4C 00 B2   JMP $B200
```

And that's the code we just read from disk, which means I get to set up another boot trace to capture it.

In Which We Flutter For A Day And Think It Is Forever

I'll reboot my work disk again, since I disconnected DOS to examine the code at `$BD00..$BFFF`.

```
*C500G
...
]CALL -151
*BLOAD TRACE6
.
.  [same as previous trace, up
to and
.  including the inline disk
read
.  routine copied from $0126
that
.  decrypts a sector into zero
page]
.
9775     A9 80   LDA #$80      change the JMP address at
9777     85 3D   STA $3D       $003C so it points to my
9779     A9 97   LDA #$97      callback instead of continuing
977B     85 3E   STA $3E       to $B200

977D  4C 00 06   JMP $0600     continue the boot
```

```
9780     A2 03   LDX #$03      (callback is here) copy the
9782  B9 00 B2   LDA $B200,Y  new code to the graphics page
9785  99 00 22   STA $2200,Y  so it survives a reboot
9788        C8   INY
9789     D0 F7   BNE $9782
978B  EE 84 97   INC $9784
978E  EE 87 97   INC $9787
9791        CA   DEX
9792     D0 EE   BNE $9782

9794  AD E8 C0   LDA $C0E8     reboot to my work disk
9797  4C 00 C5   JMP $C500

*BSAVE TRACE7,A$9600,L$19A
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE
OBJ.B200-B4FF,A$2200,L$300
]CALL -151
*B200<2200.24FFM
*B200L
B200     A9 04   LDA #$04
B202  20 00 B4   JSR $B400
B205     A9 00   LDA #$00
B207     85 5A   STA $5A
B209  20 00 B3   JSR $B300
B20C  4C 00 B5   JMP $B500
```

$B400 is a disk seek routine, identical to the one at $BE00. (It even has the same dual entry points for seeking by half track and quarter track, at $B400 and $B403.) There's nothing at $B500 yet, so the routine at $B300 must be another disk read.

```
*B300L

B300     A0 00   LDY #$00      some zero page initialization
B302     A9 B5   LDA #$B5
B304     84 59   STY $59
B306        48   PHA
B307  20 30 B3   JSR $B330

*B330L

B330        48   PHA           more zero page initialization
B331     A5 5A   LDA $5A
B333     29 07   AND #$07
B335        A8   TAY
B336  B9 50 B3   LDA $B350,Y
B339     85 50   STA $50
B33B     A5 5A   LDA $5A
B33D        4A   LSR
B33E     09 AA   ORA #$AA
B340     85 51   STA $51
B342     A5 5A   LDA $5A
B344     09 AA   ORA #$AA
B346     85 52   STA $52
B348        68   PLA
B349     E6 5A   INC $5A
B34B  4C 60 B3   JMP $B360

*B350.
B350 D5 B5 B7 BC DF D4 B4 DB
```

That could be an array of nibbles. Maybe a rotating prologue? Or a decryption key?

Oh joy. Another disk read routine.

```
*B360L
B360     85 54   STA $54
B362     A2 02   LDX #$02
B364     86 57   STX $57
B366     A0 00   LDY #$00
B368     A5 54   LDA $54
B36A     84 55   STY $55
B36C     85 56   STA $56

B36E  AE 66 BF   LDX $BF66     find a 3-nibble prologue
B371  BD 8C C0   LDA $C08C,X   (varies, based on the zero
B374     10 FB   BPL $B371     page locations that were
B376     C5 50   CMP $50       initialized at $B330 based on
B378     D0 F7   BNE $B371     the array at $B350)
B37A  BD 8C C0   LDA $C08C,X
B37D     10 FB   BPL $B37A
B37F     C5 51   CMP $51
B381     D0 F3   BNE $B376
B383  BD 8C C0   LDA $C08C,X
B386     10 FB   BPL $B383
B388     C5 52   CMP $52
B38A     D0 F3   BNE $B37F

B38C  BD 8C C0   LDA $C08C,X   read a 4-4-encoded sector
B38F     10 FB   BPL $B38C
B391        2A   ROL
B392     85 58   STA $58
B394  BD 8C C0   LDA $C08C,X
B397     10 FB   BPL $B394
B399     25 58   AND $58

B39B     91 55   STA ($55),Y   store the data into ($55)
B39D        C8   INY
B39E     D0 EC   BNE $B38C

B3A0  0E FF FF   ASL $FFFF     find a 1-nibble epilogue
B3A3  BD 8C C0   LDA $C08C,X   ("D4")
B3A6     10 FB   BPL $B3A3
B3A8     C9 D4   CMP #$D4
B3AA     D0 B6   BNE $B362
B3AC     E6 56   INC $56
B3AE     C6 57   DEC $57
B3B0     D0 DA   BNE $B38C
B3B2        60   RTS
```

Let's see:

$57 is the sector count. Initially #$02 (set at $B364), decremented at $B3AE.

$56 is the target page in memory. Set at $B36C to the accumulator, which is set at $B368 to the value of address $54, which is set at $B360 to the accumulator, which is set at $B348 by the PLA, which was pushed to the stack at $B330, which was originally set at $B302 to a constant value of #$B5. Then $56 is incremented (at $B3AC) after reading and decoding $100 bytes worth of data from disk.

$55 is #$00, as set at $B36A.

So this reads two sectors into $B500..$B6FF and returns to the caller.

Backtracking to $B30A...

```
B30A     A4 59   LDY $59       $59 is initially #$00 (set at
B30C        18   CLC           $B304)

B30D  AD 65 BF   LDA $BF65     current phase (track x 2)
B310  79 28 B3   ADC $B328,Y   new phase

B313  20 03 B4   JSR $B403     move the drive head to the
                               new phase, but using the
                               second entry point, which
                               uses a reduced timing loop (!)
B316        68   PLA           this pulls the value that was
                               pushed to the stack at $B306,
                               which was the target memory
                               page to store the data being
                               read from disk by the routine
                               at $B360
B317        18   CLC           page += 2
B318     69 02   ADC #$02

B31A     A4 59   LDY $59       counter += 1
B31C        C8   INY

B31D     C0 04   CPY #$04      loop for 4 iterations
B31F     90 E3   BCC $B304
B321        60   RTS
```

So we're reading two sectors at a time, four times, into $B500+. 2 x 4 = 8, so we're loading into $B500..$BCFF. That completely fills the gap in memory between the code at $B200..$B4FF (this chunk) and the code at $BD00..$BFFF (copied much earlier), which strongly suggests that my analysis is correct.

But what's going on with the weird drive seeking?

There is some definite weirdness here, and it's centered around the array at $B328. At $B200, we called the main entry point for the drive seek routine at $B400 to seek to track 2. Now, after reading two sectors, we're calling the secondary entry point (at $B403) to seek... where exactly?

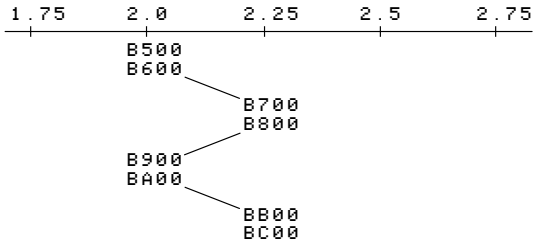
```
*B328.
B328 01 FF 01 00 00 00 00 00
```

Aha! This array is the differential to get the drive to seek forward or back. At $B200, we seeked to track 2. The first time through this loop at $B304, we read two sectors into $B500..$B6FF, then add 1 to the current phase, because $B328 = #$01. Normally this would seek forward a half track, to track 2.5, but because we're using the reduced timing loop, we only seek forward by a quarter track, to track 2.25.

The second time through the loop, we read two sectors into $B700..$B8FF, then subtract 1 from the phase (because $B329 = #$FF) and seek backwards by a quarter track. Now we're back on track 2.0.

The third time, we read two sectors from track 2.25 into $B900..$BAFF, then seek forward by a quarter track, because $B32A = #$01.

The fourth and final time, we read the final two sectors from track 2.25 into $BB00..$BCFF.



This explains the little "fluttering" noise the original disk makes during this phase of the boot. It's flipping back and forth between adjacent quarter tracks, reading two sectors from each.

Boy am I glad I'm not trying to copy this disk with a generic bit copier. That would be nearly impossible, even if I knew exactly which tracks were split like this.

In Which The Floodgates Burst Open

```
*BLOAD TRACE7
.
.  [same as previous trace]
.
9780     A9 8D   LDA #$8D      interrupt the boot at $B20C
9782  8D 0D B2   STA $B20D      after it calls $B300 but before
9785     A9 97   LDA #$97      it jumps to the new code at
9787  8D 0E B2   STA $B20E     $B500

978A  4C 00 B2   JMP $B200     continue the boot

978D     A2 08   LDX #$08      (callback is here) capture the
978F     A0 00   LDY #$00      code at $B500..$BCFF so it
9791  B9 00 B5   LDA $B500,Y   survives a reboot
9794  99 00 25   STA $2500,Y
9797        C8   INY
9798     D0 F7   BNE $9791
979A  EE 93 97   INC $9793
979D  EE 96 97   INC $9796
97A0        CA   DEX
97A1     D0 EE   BNE $9791

97A3  AD E8 C0   LDA $C0E8     reboot to my work disk
97A6  4C 00 C5   JMP $C500

*BSAVE TRACE8,A$9600,L$1A9
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE
OBJ.B500-BCFF,A$2500,L$800
]CALL -151
*B500<2500.2CFFM
*B500L
B500  AE 5F 00   LDX $005F     same command ID (saved at
                               $BF76) that was "printed"
                               earlier (passed to the routine
                               at $BF6F via $FDED)

B503  BD 80 B5   LDA $B580,X   use command ID as an index
                               into this new array

B506  8D 0A B5   STA $B50A     (!)store the array value in the
                               middle of the next JSR
                               instruction

B509  20 50 B5   JSR $B550     and call it (modified based on
                               the previous lookup)
*B580.
B580 50 58 68 70 00 00 58
```

The high byte of the JSR address never changes, so depending on the command ID, we're calling

- 00 => $B550
- 01 => $B558
- 02 => $B568
- 03 => $B570
- 06 => $B558 again

A nice, compact jump table.

```
*B550L
B550     A9 09   LDA #$09
B552     A0 00   LDY #$00
B554  4C 00 BA   JMP $BA00

*B558L
B558     A9 19   LDA #$19
B55A     A0 00   LDY #$00
B55C  20 00 BA   JSR $BA00
B55F     A9 29   LDA #$29
B561     A0 68   LDY #$68
B563  4C 00 BA   JMP $BA00

*B568L
B568     A9 31   LDA #$31
B56A     A0 00   LDY #$00
B56C  4C 00 BA   JMP $BA00

*B570L
B570     A9 41   LDA #$41
B572     A0 A0   LDY #$A0
B574  4C 00 BA   JMP $BA00
```

Those all look quite similar. Let's see what's at $BA00.

```
*BA00L
BA00        48   PHA          save the two input parameters
BA01     84 58   STY $58       (A & Y)

BA03  20 00 BE   JSR $BE00     seek the drive to a new phase
                               (given in A)

BA06     A2 00   LDX #$00      copy a number of bytes from
BA08     A4 58   LDY $58       $B900,Y (Y was passed in
BA0A  B9 00 B9   LDA $B900,Y   from the caller) to $BB00
BA0D  9D 00 BB   STA $BB00,X
BA10        C8   INY
BA11        E8   INX

BA12     E0 0C   CPX #$0C      $0C bytes. Always exactly
BA14     90 F4   BCC $BA0A     $0C bytes.
```

What's at $B900? All kinds of fun[22] stuff.

[22] not guaranteed, actual fun may vary

```
*B900.
B900 08 09 0A 0B 0C 0D 0E 0F
B908 10 11 12 13 14 15 16 17
B910 18 19 1A 1B 1C 1D 1E 1F
B918 20 21 22 23 24 25 26 27
B920 28 29 2A 2B 2C 2D 2E 2F
B928 30 31 32 33 34 35 36 37
B930 38 39 3A 3B 3C 3D 3E 3F
B938 60 61 62 63 64 65 66 67
B940 68 69 6A 6B 6C 6D 6E 6F
B948 70 71 72 73 74 75 76 77
B950 78 79 7A 7B 7C 7D 7E 7F
B958 80 81 82 83 84 85 86 87
B960 00 00 00 00 00 00 00 00
```

That looks suspiciously like a set of high bytes for addresses in main memory. Note how it starts at `#$08` (immediately after the text page), then later jumps from `#$3F` to `#$60`, skipping over hi-res page 2.

Continuing from `$BA16`...

```
BA16  20 30 BA   JSR $BA30

*BA30L
BA30  AD 65 BF   LDA $BF65     current phase

BA33        4A   LSR           convert it to a track number
BA34     A2 03   LDX #$03

BA36     29 0F   AND #$0F      (track MOD $10)

BA38        A8   TAY           use that as the index into an
BA39  B9 10 BC   LDA $BC10,Y   array

BA3C     95 50   STA $50,X     and store it in zero page
BA3E        C8   INY
BA3F        98   TYA
BA40        CA   DEX
BA41     10 F3   BPL $BA36

*BC10.
BC10 F7 F5 EF EE DF DD D6 BE
BC18 BD BA B7 B6 AF AD AB AA
```

All of those are valid nibbles. Maybe this is setting up another rotating prologue for the next disk read routine?

Continuing from `$BA43`...

```
BA43  4C 0C BB   JMP $BB0C

*BB0CL
```

Oh joy. Another disk read routine.

```
BB0C     A2 0C   LDX #$0C      I think $54 is the sector count
BB0E     86 54   STX $54

BB10     A0 00   LDY #$00      and $55 is the logical sector
BB12  8C 54 BB   STY $BB54     number
BB15     84 55   STY $55

BB17  AE 66 BF   LDX $BF66     find a 3-nibble prologue
BB1A  BD 8C C0   LDA $C08C,X   (varies by track, set up at
BB1D     10 FB   BPL $BB1A     $BA39)
BB1F     C5 50   CMP $50
BB21     D0 F7   BNE $BB1A
BB23  BD 8C C0   LDA $C08C,X
BB26     10 FB   BPL $BB23
BB28     C5 51   CMP $51
BB2A     D0 EE   BNE $BB1A
BB2C  BD 8C C0   LDA $C08C,X
BB2F     10 FB   BPL $BB2C
BB31     C5 52   CMP $52
BB33     D0 E5   BNE $BB1A

BB35     A4 55   LDY $55       logical sector number
                               (initialized to #$00 at $BB15)
BB37  B9 00 BB   LDA $BB00,Y   use the sector number as an
                               index into the $0C-length
                               page array we set up at $BA06)
BB3A  8D 55 BB   STA $BB55     and modify the upcoming
BB3D     E6 55   INC $55       code

BB3F  BC 8C C0   LDY $C08C,X   get the actual byte
BB42     10 FB   BPL $BB3F
BB44  B9 00 BC   LDA $BC00,Y
BB47        0A   ASL
BB48        0A   ASL
BB49        0A   ASL
BB4A        0A   ASL
BB4B  BC 8C C0   LDY $C08C,X
BB4E     10 FB   BPL $BB4B
BB50  19 00 BC   ORA $BC00,Y

BB53  8D 00 FF   STA $FF00     modified earlier (at $BB3A) to
BB56  EE 54 BB   INC $BB54     be the desired page in
BB59     D0 E4   BNE $BB3F     memory
BB5B  EE 55 BB   INC $BB55

BB5E  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue (also
BB61     10 FB   BPL $BB5E     varies by track)
BB63     C5 53   CMP $53
BB65     D0 A5   BNE $BB0C

BB67     C6 54   DEC $54       loop for all $0C sectors
BB69     D0 CA   BNE $BB35
BB6B        60   RTS
```

So we've read `$0C` sectors from the current track, which is the most you can fit on a track with this kind of "4-and-4" nibble encoding scheme.

Continuing from `$BA19`...

```
BA19     A5 58   LDA $58       increment the pointer to the
BA1B        18   CLC           next memory page
BA1C     69 0C   ADC #$0C
BA1E        A8   TAY

BA1F  B9 00 B9   LDA $B900,Y   if the next page is #$00,
BA22     F0 07   BEQ $BA2B     we're done

BA24        68   PLA           otherwise loop back, where
BA25        18   CLC           we'll move the drive head one
BA26     69 02   ADC #$02      full track forward and read
BA28     D0 D6   BNE $BA00     another $0C sectors

BA2B        68   PLA           execution continues here
BA2C        60   RTS           (from $BA22)
```

Now we have a whole bunch of new stuff in memory. In this case, $B550 started on track 4.5 (A = #$09 on entry to $BA00) and filled $0800..$3FFF and $6000..$87FF. If we "print" a different character, the routine at $B500 will route through one of the other subroutines—$B558, $B568, or $B570. Each of them starts on a different track (A) and uses a different starting index (Y) into the page array at $B900. The underlying routine at $BA00 doesn't know anything else; it just seeks and reads $0C sectors per track until the target page = #$00.

Continuing from $B50C...

```
B50C  20 00 B7   JSR $B700

*B700L
B700     A2 00   LDX #$00      oh joy, another decryption
B702  BD 00 B6   LDA $B600,X   loop
B705  5D 00 BE   EOR $BE00,X
B708  9D 00 03   STA $0300,X
B70B        E8   INX
B70C     E0 D0   CPX #$D0
B70E     90 F2   BCC $B702

B710  CE 13 B7   DEC $B713 (!)
B713  6D 09 B7   ADC $B709
B716        60   RTS
```

And more self-modifying code.

```
*B713:6C
*B713L
B713  6C 09 B7   JMP ($B709)
```

...which will jump to the newly decrypted code at $0300.

To recap: after 7 boot traces, the bootloader prints a null character via $FD90, which jumps to $FDED, which jumps to ($0036), which jumps to $BF6F, which calls $BEB0, which decrypts the code at $BF9F and returns just in time to execute it. $BF9F reads 3 sectors into $B200-$B4FF, pushes #$00/#$3B to the stack and exits via RTS, which returns to $003C, which jumps to $B200. $B200 reads 8 sectors into $B500-$BCFF from tracks 2 and 2.5, shifting between the adjacent quarter tracks every two sectors, then jumps to $B500, which calls $B5[50|58|68|70], which reads actual game code from multiple tracks starting at track 4.5, 9.5, 24.5, or 32.5. Then it calls $B700, which decrypts $B600 into $0300 (using $BE00+ as the decryption key) and exits via a jump to $0300.

I'm sure[23] the code at $0300 will be straightforward and easy to understand.

[23] not actually sure

In Which We Go Completely Insane

The code at $B600 is decrypted with the code at $BE00 as the key. That was originally copied from the text page the first time, not the second time.

```
*BLOAD BOOT1 0400-07FF,A$2400
*BE00<2600.26FFM ; move key
into place
*B710:60 ; stop after loop
*B700G ; decrypt
*300L
0300     A0 00   LDY #$00      wipe almost everything we've
0302        98   TYA           already loaded at the top of
0303  99 00 B1   STA $B100,Y   main memory (!)
0306        C8   INY
0307     D0 F9   BNE $0302
0309  EE 05 03   INC $0305
030C  AE 05 03   LDX $0305

030F     E0 BD   CPX #$BD      stop at $BD00
0311     90 F0   BCC $0303
```

OK, so all we're left with in memory is the RWTS at $BD00..$BFFF (including the $FDED vector at $BF6F) and the single page at $B000. Oh, and the game, but who cares about that?

Moving on...

```
0313     A9 07   LDA #$07
0315  20 80 03   JSR $0380

*380L
0380  20 00 BE   JSR $BE00     drive seek (A = #$07, so
                               track 3.5)
0383     A2 03   LDX #$03      Pull 4 bytes from the stack,
0385        68   PLA           thus negating the JSR that
0386        CA   DEX           got us here (at $0315) and the
0387     10 FC   BPL $0385     JSR before that (at $B50C).

0389  4C 18 03   JMP $0318     continue by jumping directly
                               to the place we would have
                               returned to, if we hadn't just
                               popped the stack (which we
                               did)
```

What. The. Fahrvergnugen.

```
*318L
Oh joy.  Another disk routine.
0318  AE 66 BF   LDX $BF66

031B     A4 5F   LDY $5F       Y = command ID (a.k.a. the
                               character we "printed" way
                               back when)
031D  BD 8C C0   LDA $C08C,X   find a 3-nibble prologue ("D4
0320     10 FB   BPL $031D     D5 D7")
0322     C9 D4   CMP #$D4
0324     D0 F7   BNE $031D
0326  BD 8C C0   LDA $C08C,X
0329     10 FB   BPL $0326
032B     C9 D5   CMP #$D5
032D     D0 F3   BNE $0322
032F  BD 8C C0   LDA $C08C,X
0332     10 FB   BPL $032F
0334     C9 D7   CMP #$D7
0336     D0 F3   BNE $032B

0338        88   DEY           branch when Y goes negative
0339     30 08   BMI $0343
```

```
033B  20 51 03   JSR $0351     read one byte from disk, store
                                it in $5E (not shown)
033E  20 51 03   JSR $0351     read 1 more byte from disk

0341     D0 F5   BNE $0338     loop back, unless the byte is
                                #$00
```

OK, I see it. It was hard to follow at first because the exit condition was checked before I knew it was a loop. But this is a loop. On track 3.5, there is a 3-nibble prologue (`"D4 D5 D7"`), then an array of values. Each value is two bytes. We're just finding the Nth value in the array. But to what end?

```
0343  20 51 03   JSR $0351     execution continues here
0346        48   PHA           (from $0339) read 2 more
0347  20 51 03   JSR $0351     bytes from disk and push
034A        48   PHA           them to the stack
```

Ah! A new "return" address!

Oh God. A new "return" address.

That's what this is: an array of addresses, indexed by the command ID. That's what we're looping through, and eventually pushing to the stack: the entry point for this block of the game.

But the entry point for each block is read directly from disk, so I have no idea what any of them are. Add that to the list of things I get to come back to later.

Onward...

```
034B  BD 88 C0   LDA $C088,X   turn off the drive motor
034E  4C 62 03   JMP $0362

*362L
0362     A0 00   LDY #$00      wipe this routine from
0364  99 00 03   STA $0300,Y   memory
0367        C8   INY
0368     C0 65   CPY #$65
036A     90 F8   BCC $0364

036C     A9 BE   LDA #$BE      push several values to the
036E        48   PHA           stack
036F     A9 AF   LDA #$AF
0371        48   PHA
0372     A9 34   LDA #$34
0374        48   PHA
0375  CE 78 03   DEC $0378 (!)
0378     29 CE   AND #$CE

More self-modifying code.
*378:28
*378L
0378        28   PLP           pop that #$34 off the stack,
0379  CE 7C 03   DEC $037C (!)  but use it as status registers
037C     61 60   ADC ($60,X)   (weird, but legal—if it turns
                               out to matter, I can figure out
*37C:60                        exactly which status bits get
*37CL                          set and cleared)
037C        60   RTS
```

Now we "return" to `$BEB0` because we pushed `#$BE/#$AF/#$34` but then popped `#$34`. The routine at `$BEB0` re-encrypts the code at `$BF9F` (because now we've XOR'd it twice so it's back to its original form) and exits via RTS, which "returns" to the address we pushed to the stack at `$0346`, which we read from track 3.5—and varies based on the command we're still executing, which is really the character we "printed" via the output vector.

Which is all completely insane.

In Which We Are Restored To Sanity LOL, Just Kidding But Soon, Maybe

Since the "`JSR $B700`" at `$B50C` never returns (because of the crazy stack manipulation at `$0383`), that's the last chance I'll get to interrupt the boot and capture this chunk of game code in memory. I won't know what the entry point is (because it's read from disk), but one thing at a time.

```
*BLOAD TRACE8
.
.  [same as previous trace]
.
978D     A9 4C   LDA #$4C      unconditionally break after
978F  8D 0C B5   STA $B50C     loading the game code into
9792     A9 59   LDA #$59      main memory
9794  8D 0D B5   STA $B50D
9797     A9 FF   LDA #$FF
9799  8D 0E B5   STA $B50E


979C  4C 00 B5   JMP $B500     continue the boot

*BSAVE TRACE9,A$9600,L$19F
*9600G
...reboots slot 6...
...read read read...
<beep>
Success!
*C050 C054 C057 C052
[displays a very nice picture
of a
 gumball machine which is
featured in
 the game's introduction
sequence]
*C051
```

OK, let's save it. According to the table at `$B900`, we filled `$0800..$3FFF` and `$6000..$87FF`. `$0800+` is overwritten on reboot by the boot sector and later by the HELLO program on my work disk. `$8000+` is also overwritten by Diversi-DOS 64K, which is annoying but not insurmountable. So I'll save this in pieces.

```
*C500G
...
]BSAVE BLOCK
00.2000-3FFF,A$2000,L$2000
]BRUN TRACE9
...reboots slot 6...
<beep>
*2800<800.1FFFM
*C500G
...
]BSAVE BLOCK
00.0800-1FFF,A$2800,L$1800
]BRUN TRACE9
...reboots slot 6...
<beep>
*2000<6000.87FFM
*C500G
...
]BSAVE BLOCK
00.6000-87FF,A$2000,L$2800
```

Now what? Well this is only the first chunk of game code, loaded by printing a null character. By setting up another trace and changing the value of zero page $5F, I can route $B500 through a different subroutine at $B558 or $B568 or $B570 and load a different chunk of game code.

```
]CALL -151
*BLOAD OBJ.B500-BCFF,A$B500
According to the lookup table
at $B580,
$B500 routed through $B558 to
load the
game code.  Here is that
routine:
*B558L
B558     A9 19    LDA #$19
B55A     A0 00    LDY #$00
B55C  20 00 BA    JSR $BA00
B55F     A9 29    LDA #$29
B561     A0 68    LDY #$68
B563  4C 00 BA    JMP $BA00
```

The first call to $BA00 will fill up the same parts of memory as we filled when the character (in $5F) was #$00—$0800..$3FFF and $6000..$87FF. But it starts reading from disk at phase $19 (track $0C 1/2), so it's a completely different chunk of code.

The second call to $BA00 starts reading at phase $29 (track $14 1/2), and it looks at $B900 + Y = $B968 to get the list of pages to fill in memory.

```
*B968.
B968 88 89 8A 8B 8C 8D 8E 8F
B970 90 91 92 93 94 95 96 97
B978 98 99 9A 9B 9C 9D 9E 9F
B980 A0 A1 A2 A3 A4 A5 A6 A7
B988 A8 A9 AA AB AC AD AE AF
B990 B2 B2 B2 B2 B2 B2 B2 B2
B998 00 00 00 00 00 00 00 00
```

The first call to $BA00 stopped just shy of $8800, and that's exactly where we pick up in the second call. I'm guessing that $B200 isn't really used, but the track read routine at $BA00 is "dumb" in that it always reads exactly $0C sectors from each track. So we're filling up $8800..$AFFF, then reading the rest of the last track into $B200 over and over.

Let's capture it.

```
*BLOAD TRACE9
.
.  [same as previous trace]
.

978D     A9 4C    LDA #$4C     again, break to the monitor at
978F  8D 0C B5    STA $B50C    $B50C instead of continuing to
9792     A9 59    LDA #$59     $B700
9794  8D 0D B5    STA $B50D
9797     A9 FF    LDA #$FF
9799  8D 0E B5    STA $B50E



979C     A9 01    LDA #$01     change the character being
979E     85 5F    STA $5F      "printed" to #$01 just before
                               the bootloader uses it to load
                               the appropriate chunk of
                               game code

97A0  4C 00 B5    JMP $B500    continue the boot

*BSAVE TRACE10,A$9600,L$1A3
*9600G
...reboots slot 6...
...read read read...
<beep>
*C050 C054 C057 C052
[displays a very nice picture
of the
 main game screen]
*C051
*C500G
...
]BSAVE BLOCK
01.2000-3FFF,A$2000,L$2000
]BRUN TRACE10
...reboots slot 6...
<beep>
*2800<800.1FFFM
*C500G
...
]BSAVE BLOCK
01.0800-1FFF,A$2800,L$1800
]BRUN TRACE9
...reboots slot 6...
<beep>
*2000<6000.AFFFM
*C500G
...
]BSAVE BLOCK
01.6000-AFFF,A$2000,L$5000
```

And similarly with blocks 2 and 3. (These are not shown here, but you can look at TRACE11 and TRACE12 on my work disk.) Blocks 4 and 5 get special-cased earlier (at $BF86 and $BF8D, respectively), so they never reach $B500 to load anything from disk. Block 6 is the same as block 1.

That's it. I've captured all the game code. Here's what the "game" looks like at this point:

```
]CATALOG
C1983 DSR^C#254
019 FREE
 A 002 HELLO
 B 003 BOOT0
*B 003 TRACE
 B 003 BOOT1 0300-03FF
*B 003 TRACE2
 B 003 BOOT1 0100-01FF
*B 003 TRACE3
 B 006 BOOT1 0400-07FF
*B 003 TRACE4
 B 005 BOOT2 0500-07FF
*B 003 TRACE5
 B 003 BOOT2 B000-B0FF
 B 003 BOOT2 0100-01FF
*B 003 TRACE6
 B 003 BOOT3 0000-00FF
*B 003 TRACE7
 B 005 OBJ.B200-B4FF
*B 003 TRACE8
 B 010 OBJ.B500-BCFF
*B 003 TRACE9
 B 026 BLOCK 00.0800-1FFF
 B 034 BLOCK 00.2000-3FFF
 B 042 BLOCK 00.6000-87FF
*B 003 TRACE10
 B 026 BLOCK 01.0800-1FFF
 B 034 BLOCK 01.2000-3FFF
 B 082 BLOCK 01.6000-AFFF
*B 003 TRACE11
 B 026 BLOCK 02.0800-1FFF
 B 034 BLOCK 02.2000-3FFF
 B 042 BLOCK 02.6000-87FF
*B 003 TRACE12
 B 034 BLOCK 03.2000-3FFF
```

It's... it's beautiful. *wipes tear*

In Which Every Exit Is An Entrance Somewhere Else

I've captured all the blocks of the game code (I think), but I still have no idea how to run it. The entry points for each block are read directly from disk, in the loop at `$031D`.

```
   COPY ][ PLUS BIT COPY PROGRAM 8.4
(C) 1982-9 CENTRAL POINT SOFTWARE, INC.
----------------------------------------

TRACK: 03.50  START: 1800  LENGTH: 3DFF
       ^^^^^

1DA0: FA AA FA AA FA AA FA AA   VIEW
1DA8: EB FA FF AE EA EB FF AE
1DB0: EB EA FC FF FF FF FF FF
1DB8: FF FF FF FF FF FF FF FF
1DC0: FF FF FF D4 D5 D7 AF AF  <-1DC3
               ^^^^^^^^

1DC8: EE BE BA BB FE FA AA BA
1DD0: BA BE FF FF AB FF FF FF
1DD8: AB FF FF FF AB FF BB AB  FIND:
1DE0: BB FF AA AA AA AA AA AA  D4 D5 D7

----------------------------------------

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ
```

Rather than try to boot-trace every possible block, I'm going to load up the original disk in a nibble editor and do the calculations myself. The array of entry points is on track 3.5. Firing up Copy II Plus nibble editor, I searched for the same 3-nibble prologue ("D4 D5 D7") that the code at `$031D` searches for, and lo and behold!

After the "D4 D5 D7" prologue, I find an array of 4-and-4-encoded nibbles starting at offset `$1DC6`. Breaking them down into pairs and decoding them with the 4-4 encoding scheme, I get this list of bytes:

nibbles	byte
AF AF	#$0F
EE BE	#$9C
BA BB	#$31
FE FA	#$F8
AA BA	#$10
BA BE	#$34
FF FF	#$FF
AB FF	#$57
FF FF	#$FF
AB FF	#$57
FF FF	#$FF
AB FF	#$57
BB AB	#$23
BB FF	#$77

And now—maybe!—I have my list of entry points for each block of the game code.

```
Only one way to know for
sure...
]PR#5
...
]CALL -151
*800:0 N 801<800.BEFEM          clear main memory so I'm not
                                accidentally relying on
                                random stuff left over from all
                                my other testing

*BLOAD BLOCK                    load all of block 0 into place
00.0800-1FFF,A$800
*BLOAD BLOCK
00.2000-3FFF,A$2000
*BLOAD BLOCK
00.6000-87FF,A$6000

*F9DG                           jump to the entry point I
[displays the game intro        found on track 3.5 (+1, since
sequence]                       the original code pushes it to
*does a little happy dance in   the stack and "returns" to it)
my chair*
```

We have no further use for the original disk. Now would be an excellent time to take it out of the drive and store it in a cool, dry place.

In Which Two Wrongs Don't Make A—Oh God I Can't Even—With This Pun

Remember when I said I'd look at `$BD00` later? The time has come. Later is now.

The output vector at `$BF6F` has special case handling if `A = #$04`. Instead of continuing to `$0300` and `$B500`, it jumps directly to `$BD00`. What's so special about `$BD00`?

The code at `$BD00` was moved there very early in the boot process, from page `$0500` on the text screen. (The first time we loaded code into the text screen, not the second time.) So it's in "BOOT1 0400-07FF" on my work disk.

```
]PR#5
...
]BLOAD BOOT1 0400-07FF,A$2400
]CALL -151
*BD00<2500.25FFM
*BD00L
BD00  AE 66 BF   LDX $BF66     turn on drive motor
BD03  BD 89 C0   LDA $C089,X

BD06     A9 64   LDA #$64      wait for drive to settle
BD08  20 A8 FC   JSR $FCA8

BD0B     A9 10   LDA #$10      seek to phase $10 (track 8)
BD0D  20 00 BE   JSR $BE00

BD10     A9 02   LDA #$02      seek to phase $02 (track 1)
BD12  20 00 BE   JSR $BE00

BD15     A0 FF   LDY #$FF      initialize data latches
BD17  BD 8D C0   LDA $C08D,X
BD1A  BD 8E C0   LDA $C08E,X
BD1D  9D 8F C0   STA $C08F,X
BD20  1D 8C C0   ORA $C08C,X

BD23     A9 80   LDA #$80      wait
BD25  20 A8 FC   JSR $FCA8
BD28  20 A8 FC   JSR $FCA8

BD2B  BD 8D C0   LDA $C08D,X   Oh God
BD2E  BD 8E C0   LDA $C08E,X
BD31        98   TYA
BD32  9D 8F C0   STA $C08F,X
BD35  1D 8C C0   ORA $C08C,X
BD38        48   PHA
BD39        68   PLA
BD3A     C1 00   CMP ($00,X)
BD3C     C1 00   CMP ($00,X)
BD3E        EA   NOP
BD3F        C8   INY

BD40  9D 8D C0   STA $C08D,X   Oh God
BD43  1D 8C C0   ORA $C08C,X
BD46  B9 8F BD   LDA $BD8F,Y
BD49     D0 EF   BNE $BD3A
BD4B        A8   TAY
BD4C        EA   NOP
BD4D        EA   NOP

BD4E  B9 00 B0   LDA $B000,Y   ← !
BD51        48   PHA
BD52        4A   LSR
BD53     09 AA   ORA #$AA

BD55  9D 8D C0   STA $C08D,X   Oh God Oh God Oh God
BD58  DD 8C C0   CMP $C08C,X
BD5B     C1 00   CMP ($00,X)
BD5D        EA   NOP
BD5E        EA   NOP
BD5F        48   PHA
BD60        68   PLA
BD61        68   PLA
BD62     09 AA   ORA #$AA
BD64  9D 8D C0   STA $C08D,X
BD67  DD 8C C0   CMP $C08C,X
BD6A        48   PHA
BD6B        68   PLA
BD6C        C8   INY
BD6D     D0 DF   BNE $BD4E
BD6F     A9 D5   LDA #$D5
BD71     C1 00   CMP ($00,X)
BD73        EA   NOP
BD74        EA   NOP
BD75  9D 8D C0   STA $C08D,X
BD78  1D 8C C0   ORA $C08C,X
BD7B     A9 08   LDA #$08
BD7D  20 A8 FC   JSR $FCA8
BD80  BD 8E C0   LDA $C08E,X
BD83  BD 8C C0   LDA $C08C,X

BD86     A9 07   LDA #$07      seek back to track 3.5
BD88  20 00 BE   JSR $BE00

BD8B  BD 88 C0   LDA $C088,X   turn off drive motor and exit
BD8E        60   RTS          gracefully
```

This is a disk write routine. It's taking the data at `$B000` (that mystery sector that was loaded even earlier in the boot) and writing it to track 1.

Because high scores.

That's what's at `$B000`. High scores. [Edit from the future: also some persistent joystick options.]

Why is this so distressing? Because it means I'll get to include a full read/write RWTS on my crack (which I haven't even starting building yet, but soon!) so it can save high scores like the original game. Because anything less is obviously unacceptable.

The Right Ones In The Right Order

Let's step back from the low-level code for a moment and talk about how this game interacts with the disk at a high level.

- There is no runtime protection check. All the "protection" is structural—data is stored on whole tracks, half tracks, and even some consecutive quarter tracks. Once the game code is in memory, there are no nibble checks or secondary protections.

- The game code itself contains no disk code. They're completely isolated. I proved this by loading the game code from my work disk and

jumping to the entry point. (I tested the animated introduction, but you can also run the game itself by loading the block `$01` files into memory and jumping to `$31F9`. The game runs until you finish the level and it tries to load the first cut scene from disk.)

- The game code communicates with the disk subsystem through the output vector, i.e. by printing `#$00..#$06` to `$FDED`. The disk code handles filling the screen with a pseudo-random color, reading the right chunks from the right places on disk and putting them into the right places in memory, then jumping to the right address to continue. (In the case of printing `#$04`, it handles writing the right data in memory to the right place on disk.)

- Game code lives at `$0800..$AFFF`, zero page, and one page at `$B000` for high scores. The disk subsystem clobbers the text screen at `$0400` using lo-res graphics for the color fills. All memory above `$B100` is available; in fact, most of it is wiped (at `$0300`) after every disk command.

This is great news. It gives us total flexibility to recreate the game from its constituent pieces.

A Man, A Plan, A Canal, *&c.*

Here's the plan:

1. Write the game code to a standard 16-sector disk
2. Write a bootloader and RWTS that can read the game code into memory
3. Write some glue code to mimic the original output vector at `$BF6F` (A = command ID from `#$00-#$06`, all other values actually print) so I don't need to change any game code
4. Declare victory[24]

Looking at the length of each block and dividing by 16, I can space everything out on separate tracks and still have plenty of room. This means each block can start on its own track, which saves a few bytes by being able to hard-code the starting sector for each block.

The disk map will look like this:

[24] take a nap

tr	memory range	notes
00	$BD00..$BFFF	Gumboot
01	$B000..$B3FF	scores/zpage/glue
02	$0800..$17FF	block 0
03	$1800..$27FF	block 0
04	$2800..$37FF	block 0
05	$3800..$3FFF	block 0
06	$6000..$67FF	block 0
07	$6800..$77FF	block 0
08	$7000..$87FF	block 0
09	$0800..$17FF	block 1
0A	$1800..$27FF	block 1
0B	$2800..$37FF	block 1
0C	$3800..$3FFF	block 1
0D	$6000..$6FFF	block 1
0E	$7000..$7FFF	block 1
0F	$8000..$8FFF	block 1
10	$9000..$9FFF	block 1
11	$A000..$AFFF	block 1
12	$0800..$17FF	block 2
13	$1800..$27FF	block 2
14	$2800..$37FF	block 2
15	$3800..$3FFF	block 2
16	$6000..$6FFF	block 2
17	$7000..$7FFF	block 2
18	$8000..$87FF	block 2
19	$2000..$2FFF	block 3
1A	$3000..$3FFF	block 3

I wrote a build script to take all the chunks of game code I captured way back on page 43. And by "script", I mean "BASIC program."

```
]PR#5
...
 10 REM MAKE GUMBALL
 11 REM S6,D1=BLANK DISK
 12 REM S5,D1=WORK DISK
 20 D$ = CHR$ (4)

 30 PRINT D$"BLOAD BLOCK       Load the first part of block 0:
00.0800-1FFF,
 A$1000"
 40 PRINT D$"BLOAD BLOCK
00.2000-3FFF,
 A$2800"

 50 PAGE = 16:COUNT = 56:TRK = Write it to tracks $02-$05:
2:
 SEC = 0:  GOSUB 1000

 60 PRINT D$"BLOAD BLOCK       Load the second part of
00.6000-87FF,                 block 0:
 A$6000"

 70 PAGE = 96:COUNT = 40:TRK = Write it to tracks $06-$08:
6:
 SEC = 0:  GOSUB 1000
```

```
 80 PRINT D$"BLOAD BLOCK
01.0800-1FFF,
 A$1000"
 90 PRINT D$"BLOAD BLOCK
01.2000-3FFF,
 A$2800"
 100 PAGE = 16:COUNT = 56:TRK
= 9:
 SEC = 0:  GOSUB 1000
 110 PRINT D$"BLOAD BLOCK
01.6000-AFFF,
 A$6000"
 120 PAGE = 96:COUNT = 80:TRK
= 13:
 SEC = 0:  GOSUB 1000
 130 PRINT D$"BLOAD BLOCK
02.0800-1FFF,
 A$1000"
 140 PRINT D$"BLOAD BLOCK
02.2000-3FFF,
 A$2800"
 150 PAGE = 16:COUNT = 56:TRK
= 18:
 SEC = 0:  GOSUB 1000
 160 PRINT D$"BLOAD BLOCK
02.6000-87FF,
 A$6000"
 170 PAGE = 96:COUNT = 40:TRK
= 22:
 SEC = 0:  GOSUB 1000
 180 PRINT D$"BLOAD BLOCK
03.2000-3FFF,
 A$2000"
 190 PAGE = 32:COUNT = 32:TRK
= 25:
 SEC = 0:  GOSUB 1000
 200 PRINT D$"BLOAD BOOT2
0500-07FF,
 A$2500"
 210 PAGE = 39:COUNT = 1:TRK =
1:
 SEC = 0:  GOSUB 1000
 220 PRINT D$"BLOAD BOOT3
0000-00FF,
 A$1000"
 230 POKE 4150,0:  POKE
4151,178:  REM
 SET ($36) TO $B200
 240 PAGE = 16:COUNT = 1:TRK =
1:
 SEC = 7:  GOSUB 1000
 999 END
 1000 REM WRITE TO DISK
 1010 PRINT D$"BLOAD WRITE"
 1020 POKE 908,TRK
 1030 POKE 909,SEC
 1040 POKE 913,PAGE
 1050 POKE 769,COUNT
 1060 CALL 768
 1070 RETURN
]SAVE MAKE
```

And so on, for all the other blocks:

The BASIC program relies on a short assembly language routine to do the actual writing to disk. Here is that routine (loaded on line 1010):

```
]CALL -151

0300     A9 D1    LDA #$D1 ☺    page count (set from BASIC)
0302     85 FF    STA $FF

0304     A9 00    LDA #$00      logical sector (incremented)
0306     85 FE    STA $FE

0308     A9 03    LDA #$03      call RWTS to write sector
030A     A0 88    LDY #$88
030C  20 D9 03    JSR $03D9

030F     E6 FE    INC $FE       increment logical sector, wrap
0311     A4 FE    LDY $FE       around from $0F to $00 and
0313     C0 10    CPY #$10      increment track
0315     D0 07    BNE $031E
0317     A0 00    LDY #$00
0319     84 FE    STY $FE
031B  EE 8C 03    INC $038C

031E  B9 40 03    LDA $0340,Y   convert logical to physical
0321  8D 8D 03    STA $038D     sector

0324  EE 91 03    INC $0391     increment page to write

0327     C6 FF    DEC $FF       loop until done with all
0329     D0 DD    BNE $0308     sectors
032B        60    RTS

*340.34F

0340 00 07 0E 06 0D 05 0C 04   logical to physical sector
0348 0B 03 0A 02 09 01 08 0F   mapping
*388.397

0388 01 60 01 00 D1 D1 FB F7
                 track/sector
              (set from BASIC)
0390 00 D1 00 00 02 00 00 60
      ↑
   address
(set from BASIC)

                               RWTS parameter table,
                               pre-initialized with slot
                               (#$06), drive (#$01), and
                               RWTS write command (#$02)

*BSAVE WRITE,A$300,L$98
[S6,D1=blank disk]
]RUN MAKE
```

. . .write write write. . .

Boom! The entire game is on tracks `$02-$1A` of a standard 16-sector disk.

Now we get to write an RWTS.

Introducing Gumboot

Gumboot is a fast bootloader and full read/write RWTS. It fits in 4 sectors on track 0, including a boot sector. It uses only 6 pages of memory for all its code + data + scratch space. It uses no zero page addresses after boot. It can start the game from a cold boot in 3 seconds. That's twice as fast as the original disk.

qkumba wrote it from scratch, because of course he did. I, um, mostly just cheered.

After boot-time initialization, Gumboot is dead simple and always ready to use:

entry	command	parameters
$BD00	read	A = first track Y = first page X = sector count
$BE00	write	A = sector Y = page
$BF00	seek	A = track

That's it. It's so small, there's `$80` unused bytes at `$BF80`. You could fit a cute message in there! (We didn't.)

Some important notes:

- The read routine reads consecutive tracks in physical sector order into consecutive pages in memory. There is no translation from physical to logical sectors.
- The write routine writes one sector, and also assumes a physical sector number.
- The seek routine can seek forward or back to any whole track. (I mention this because some fastloaders can only seek forward.)

I said Gumboot takes 6 pages in memory, but I've only mentioned 3. The other 3 are for data:

`$BA00..$BB55` scratch space for write (technically available as long as you don't mind them being clobbered during disk write)

`$BB00..$BCFF` data tables (initialized once during boot)

Gumboot Boot0

Gumboot starts, as all disks start, on track `$00`. Sector `$00` (boot0) reuses the disk controller ROM routine to read sector `$0E`, `$0D`, and `$0C` (boot1). Boot0 creates a few data tables, modifies the boot1 code to accommodate booting from any slot, and jumps to it.

Boot0 is loaded at `$0800` by the disk controller ROM routine.

```
0800 [01]                          tell the ROM to load only this sector (we'll do the rest manually)

0801      4A     LSR               The accumulator is #$01 after loading sector $00, #$03 after loading sector $0E, #$05 after loading sector $0D, and #$07 after loading sector $0C. We shift it right to divide by 2, then use that to calculate the load address of the next sector.
0802   69 BC     ADC #$BC          Sector $0E → $BD00
                                   Sector $0D → $BE00
                                   Sector $0C → $BF00
0804   85 27     STA $27           store the load address

0806      0A     ASL               shift the accumulator again
0807      0A     ASL               (now that we've stored the load address)

0808      8A     TXA               transfer X (boot slot x16) to the accumulator, which will be useful later but doesn't affect the carry flag we may have just tripped with the two "ASL" instructions
0809   B0 0D     BCS $0818         if the two "ASL" instructions set the carry flag, it means the load address was at least #$C0, which means we've loaded all the sectors we wanted to load and we should exit this loop

080B   E6 3D     INC $3D           Set up next sector number to read. The disk controller ROM does this once already, but due to quirks of timing, it's much faster to increment it twice so the next sector you want to load is actually the next sector under the drive head. Otherwise you end up waiting for the disk to spin an entire revolution, which is quite slow.

080D      4A     LSR               Set up the "return" address to
080E      4A     LSR               jump to the "read sector"
080F      4A     LSR               entry point of the disk
0810      4A     LSR               controller ROM. This could
0811   09 C0     ORA #$C0          be anywhere in $Cx00 depending on the slot we booted from, which is why we put the boot slot in the accumulator at $0808.
```

```
0813        48   PHA            push the entry point on the
0814     A9 5B   LDA #$5B       stack
0816        48   PHA

0817        60   RTS            "Return" to the entry point
                                via RTS. The disk controller
                                ROM always jumps to $0801
                                (remember, that's why we
                                had to move it and patch it to
                                trace the boot all the way
                                back on page 25), so this
                                entire thing is a loop that
                                only exits via the "BCS"
                                branch at $0809.

0818     09 8C   ORA #$8C       Execution continues here
081A     A2 00   LDX #$00       (from $0809) after three
081C  BC AF 08   LDY $08AF,X    sectors have been loaded into
081F     84 26   STY $26        memory at $BD00..$BFFF.
0821  BC B0 08   LDY $08B0,X    There are a number of places
0824     F0 0A   BEQ $0830      in boot1 that hit a
0826     84 27   STY $27        slot-specific soft switch (read
0828     A0 00   LDY #$00       a nibble from disk, turn off
082A     91 26   STA ($26),Y    the drive, &c.). Rather than
082C        E8   INX            the usual form of "LDA
082D        E8   INX            $C08C,X", we will use "LDA
082E     D0 EC   BNE $081C      $C0EC" and modify the $EC
                                byte in advance, based on the
                                boot slot. $08A4 is an array of
                                all the places in the Gumboot
                                code that get this adjustment.

0830     29 F8   AND #$F8       munge $EC → $E8 (used later
0832  8D FC BD   STA $BDFC     to turn off the drive motor)

0835     09 01   ORA #$01       munge $E8 → $E9 (used later
0837  8D 0B BD   STA $BD0B     to turn on the drive motor)
083A  8D 07 BE   STA $BE07

083D     49 09   EOR #$09       munge $E9 → $E0 (used later
083F  8D 54 BF   STA $BF54     to move the drive head via
                                the stepper motor)

0842     29 70   AND #$70       munge $E0 → $60 (boot slot
0844  8D 37 BE   STA $BE37     x16, used during seek and
0847  8D 69 BE   STA $BE69     write routines)
084A  8D 7F BE   STA $BE7F
084D  8D AC BE   STA $BEAC
```

6 + 2

Before I dive into the next chunk of code, I get to pause and explain a little bit of theory. As you probably know if you're the sort of person who's read this far already, Apple II floppy disks do not contain the actual data that ends up being loaded into memory. Due to hardware limitations of the original Disk II drive, data on disk is stored in an intermediate format called "nibbles." Bytes in memory are encoded into nibbles before writing to disk, and nibbles that you read from the disk must be decoded back into bytes. The round trip is lossless but requires some bit wrangling.

Decoding nibbles-on-disk into bytes-in-memory is a multi-step process. In "6-and-2 encoding" (used by DOS 3.3, ProDOS, and all ".dsk" image files), there are 64 possible values that you may find in the data field. (In the range `$96..$FF`, but not all of those, because some of them have bit patterns that trip up the drive firmware.) We'll call these "raw nibbles."

Step 1) read `$156` raw nibbles from the data field. These values will range from `$96` to `$FF`, but as mentioned earlier, not all values in that range will appear on disk.

Now we have `$156` raw nibbles.

Step 2) decode each of the raw nibbles into a 6-bit byte between 0 and 63. (`%00000000` and `%00111111` in binary.) `$96` is the lowest valid raw nibble, so it gets decoded to 0. `$97` is the next valid raw nibble, so it's decoded to 1. `$98` and `$99` are invalid, so we skip them, and `$9A` gets decoded to 2. And so on, up to `$FF` (the highest valid raw nibble), which gets decoded to 63.

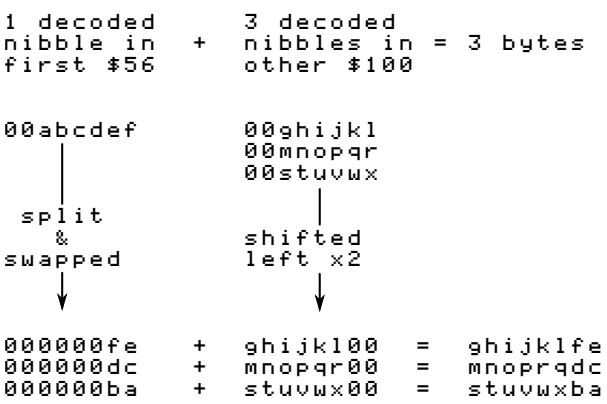
Now we have `$156` 6-bit bytes.

Step 3) split up each of the first `$56` 6-bit bytes into pairs of bits. In other words, each 6-bit byte becomes three 2-bit bytes. These 2-bit bytes are merged with the next `$100` 6-bit bytes to create `$100` 8-bit bytes. Hence the name, "6-and-2" encoding.

The exact process of how the bits are split and merged is... complicated. The first `$56` 6-bit bytes get split up into 2-bit bytes, but those two bits get swapped such that `%01` becomes `%10` and vice-versa. The other `$100` 6-bit bytes each get multiplied by 4 (a.k.a. bit-shifted two places left). This leaves a

hole in the lower two bits, which is filled by one of the 2-bit bytes from the first group.

A diagram might help. "a" through "x" each represent one bit.



Tada! Four 6-bit bytes

```
00abcdef
00ghijkl
00mnopqr
00stuvwx
```

become three 8-bit bytes

```
ghijklfe
mnoprqdc
stuvwxba
```

When DOS 3.3 reads a sector, it reads the first `$56` raw nibbles, decoded them into 6-bit bytes, and stashes them in a temporary buffer at `$BC00`. Then it reads the other `$100` raw nibbles, decodes them into 6-bit bytes, and puts them in another temporary buffer at `$BB00`. Only then does DOS 3.3 start combining the bits from each group to create the full 8-bit bytes that will end up in the target page in memory. This is why DOS 3.3 "misses" sectors when it's reading, because it's busy twiddling bits while the disk is still spinning.

Gumboot also uses "6-and-2" encoding. The first `$56` nibbles in the data field are still split into pairs of bits that will be merged with nibbles that won't come until later. But instead of waiting for all `$156` raw nibbles to be read from disk, it "interleaves" the nibble reads with the bit twiddling required to merge the first `$56` 6-bit bytes and the `$100` that follow. By the time Gumboot gets to the data field checksum, it has already stored all `$100` 8-bit bytes in their final resting place in memory. This means that we can read all 16 sectors on a track in one revolution of the disk. That's what makes it crazy fast.

To make it possible to twiddle the bits and not miss nibbles as the disk spins[25], we do some of the work in advance. We multiply each of the 64 possible decoded values by 4 and store those values. (Since this is done by bit shifting and we're doing it before we start reading the disk, this is called the "pre-shift" table.) We also store all possible 2-bit values in a repeating pattern that will make it easy to look them up later. Then, as we're reading from disk (and timing is tight), we can simulate bit math with a series of table lookups. There is just enough time to convert each raw nibble into its final 8-bit byte before reading the next nibble.

The first table, at `$BC00..$BCFF`, is three columns wide and 64 rows deep. Astute readers will notice that 3 x 64 is not 256. Only three of the columns are used; the fourth (unused) column exists because multiplying by 3 is hard but multiplying by 4 is easy in base 2. The three columns correspond to the three pairs of 2-bit values in those first `$56` 6-bit bytes. Since the values are only 2 bits wide, each column holds one of four different values. (`%00`, `%01`, `%10`, or `%11`.)

The second table, at `$BB96..$BBFF`, is the "pre-shift" table. This contains all the possible 6-bit bytes, in order, each multiplied by 4. (They are shifted to the left two places, so the 6 bits that started in columns 0-5 are now in columns 2-7, and columns 0 and 1 are zeroes.) Like this:

```
00ghijkl -> ghijkl00
```

Astute readers will notice that there are only 64 possible 6-bit bytes, but this second table is larger than 64 bytes. To make lookups easier, the table has empty slots for each of the invalid raw nibbles. In other words, we don't do any math to decode raw nibbles into 6-bit bytes; we just look them up in this table (offset by `$96`, since that's the lowest valid raw nibble) and get the required bit shifting for free.

[25]The disk spins independently of the CPU, and we only have a limited time to read a nibble and do what we're going to do with it before WHOOPS HERE COMES ANOTHER ONE. So time is of the essence. Also, "As The Disk Spins" would make a great name for a retrocomputing-themed soap opera.

addr	raw	decoded 6-bit	pre-shift
$BB96	$96	0 = %00000000	%00000000
$BB97	$97	1 = %00000001	%00000100
$BB98	$98	[invalid raw nibble]	
$BB99	$99	[invalid raw nibble]	
$BB9A	$9A	2 = %00000010	%00001000
$BB9B	$9B	3 = %00000011	%00001100
$BB9C	$9C	[invalid raw nibble]	
$BB9D	$9D	4 = %00000100	%00010000
.			
.			
.			
$BBFE	$FE	62 = %00111110	%11111000
$BBFF	$FF	63 = %00111111	%11111100

Each value in this "pre-shift" table also serves as an index into the first table with all the 2-bit bytes. This wasn't an accident; I mean, that sort of magic doesn't just happen. But the table of 2-bit bytes is arranged in such a way that we can take one of the raw nibbles to be decoded and split apart (from the first $56 raw nibbles in the data field), use each raw nibble as an index into the pre-shift table, then use that pre-shifted value as an index into the first table to get the 2-bit value we need.

Back to Gumboot

This is the loop that creates the pre-shift table at $BB96. As a special bonus, it also creates the inverse table that is used during disk write operations, converting in the other direction.

```
0850     A2 3F   LDX #$3F
0852     86 FF   STX $FF
0854        E8   INX
0855     A0 7F   LDY #$7F
0857     84 FE   STY $FE
0859        98   TYA
085A        0A   ASL
085B     24 FE   BIT $FE
085D     F0 18   BEQ $0877
085F     05 FE   ORA $FE
0861     49 FF   EOR #$FF
0863     29 7E   AND #$7E
0865     B0 10   BCS $0877
0867        4A   LSR
0868     D0 FB   BNE $0865
086A        CA   DEX
086B        8A   TXA
086C        0A   ASL
086D        0A   ASL
086E  99 80 BB   STA $BB80,Y
0871        98   TYA
0872     09 80   ORA #$80
0874  9D 56 BB   STA $BB56,X
0877        88   DEY
0878     D0 DD   BNE $0857
```

And this is the result, where ".." means that the address is uninitialized and unused.

```
BB90                    00 04
BB98 .. .. 08 0C .. 10 14 18
BBA0 .. .. .. .. .. .. 1C 20
BBA8 .. .. .. 24 28 2C 30 34
BBB0 .. .. 38 3C 40 44 48 4C
BBB8 .. 50 54 58 5C 60 64 68
BBC0 .. .. .. .. .. .. .. ..
BBC8 .. .. .. 6C .. 70 74 78
BBD0 .. .. .. 7C .. .. 80 84
BBD8 .. 88 8C 90 94 98 9C A0
BBE0 .. .. .. .. .. A4 A8 AC
BBE8 .. B0 B4 B8 BC C0 C4 C8
BBF0 .. .. CC D0 D4 D8 DC E0
BBF8 .. E4 E8 EC F0 F4 F8 FC
```

Next up: a loop to create the table of 2-bit values at $BC00, magically arranged to enable easy lookups later.

```
087A     84 FD   STY $FD
087C     46 FF   LSR $FF
087E     46 FF   LSR $FF
0880  BD BD 08   LDA $08BD,X
0883  99 00 BC   STA $BC00,Y
0886     E6 FD   INC $FD
0888     A5 FD   LDA $FD
088A     25 FF   AND $FF
088C     D0 05   BNE $0893
088E        E8   INX
088F        8A   TXA
0890     29 03   AND #$03
0892        AA   TAX
0893        C8   INY
0894        C8   INY
0895        C8   INY
0896        C8   INY
0897     C0 03   CPY #$03
0899     B0 E5   BCS $0880
089B        C8   INY
089C     C0 03   CPY #$03
089E     90 DC   BCC $087C
```

And this is the result:

```
BC00 00 00 00 .. 00 00 02 ..
BC08 00 00 01 .. 00 00 03 ..
BC10 00 02 00 .. 00 02 02 ..
BC18 00 02 01 .. 00 02 03 ..
BC20 00 01 00 .. 00 01 02 ..
BC28 00 01 01 .. 00 01 03 ..
BC30 00 03 00 .. 00 03 02 ..
BC38 00 03 01 .. 00 03 03 ..
BC40 02 00 00 .. 02 00 02 ..
BC48 02 00 01 .. 02 00 03 ..
BC50 02 02 00 .. 02 02 02 ..
BC58 02 02 01 .. 02 02 03 ..
BC60 02 01 00 .. 02 01 02 ..
BC68 02 01 01 .. 02 01 03 ..
BC70 02 03 00 .. 02 03 02 ..
BC78 02 03 01 .. 02 03 03 ..
BC80 01 00 00 .. 01 00 02 ..
BC88 01 00 01 .. 01 00 03 ..
BC90 01 02 00 .. 01 02 02 ..
BC98 01 02 01 .. 01 02 03 ..
BCA0 01 01 00 .. 01 01 02 ..
BCA8 01 01 01 .. 01 01 03 ..
BCB0 01 03 00 .. 01 03 02 ..
BCB8 01 03 01 .. 01 03 03 ..
BCC0 03 00 00 .. 03 00 02 ..
BCC8 03 00 01 .. 03 00 03 ..
BCD0 03 02 00 .. 03 02 02 ..
BCD8 03 02 01 .. 03 02 03 ..
BCE0 03 01 00 .. 03 01 02 ..
BCE8 03 01 01 .. 03 01 03 ..
BCF0 03 03 00 .. 03 03 02 ..
BCF8 03 03 01 .. 03 03 03 ..
```

And with that, Gumboot is fully armed and operational.

```
08A0     A9 B2    LDA #$B2     Push a "return" address on
08A2        48    PHA          the stack. We'll come back to
08A3     A9 F0    LDA #$F0     this later. (Ha ha, get it,
08A5        48    PHA          come back to it? OK, let's
                                pretend that never happened.)

08A6     A9 01    LDA #$01     Set up an initial read of 3
08A8     A2 03    LDX #$03     sectors from track 1 into
08AA     A0 B0    LDY #$B0     $B000..$B2FF. This contains
                                the high scores data, zero
                                page, and a new output vector
                                that interfaces with Gumboot.

08AC  4C 00 BD    JMP $BD00    Read all that from disk and
                               exit via the "return" address
                               we just pushed on the stack
                               at $0895.
```

Execution will continue at `$B2F1`, once we read that from disk. `$B2F1` is new code I wrote, and I promise to show it to you. But first, I get to finish showing you how the disk read routine works.

Read & Go Seek

In a standard DOS 3.3 RWTS, the softswitch to read the data latch is "`LDA $C08C,X`", where X is the boot slot times 16, to allow disks to boot from any slot. Gumboot also supports booting and reading from any slot, but instead of using an index, most fetch instructions are set up in advance based on the boot slot. Not only does this free up the X register, it lets us juggle all the registers and put the raw nibble value in whichever one is convenient at the time. (We take full advantage of this freedom.) I've marked each pre-set softswitch with ☺.

There are several other instances of addresses and constants that get modified while Gumboot is executing. I've left these with a bogus value `$D1` and marked them with ☺.

Gumboot's source code should be available from the same place you found this write-up. If you're looking to modify this code for your own purposes, I suggest you "use the source, Luke."

```
*BD00L
BD00        0A   ASL           A = the track number to seek
BD01  8D 10 BF   STA $BF10     to. We multiply it by 2 to
                               convert it to a phase, then
                               store it inside the seek routine
                               which we will call shortly.

BD04  8E EF BD   STX $BDEF     X = the number of sectors to
                               read

BD07  8C 24 BD   STY $BD24     Y = the starting address in
                               memory

BD0A  AD E9 C0   LDA $C0E9 ☺   turn on the drive motor

BD0D  20 75 BF   JSR $BF75     poll for real nibbles (#$FF
                               followed by non-#$FF) as a
                               way to ensure the drive has
                               spun up fully

BD10     A9 10   LDA #$10      are we reading this entire
BD12  CD EF BD   CMP $BDEF     track?

BD15     B0 01   BCS $BD18     yes -> branch

BD17        AA   TAX           no
BD18  8E 94 BF   STX $BF94

BD1B  20 04 BF   JSR $BF04     seek to the track we want

BD1E  AE 94 BF   LDX $BF94     Initialize an array of which
BD21     A0 00   LDY #$00      sectors we've read from the
BD23     A9 D1   LDA #$D1 ☺    current track. The array is in
BD25  99 84 BF   STA $BF84,Y   physical sector order, thus the
BD28  EE 24 BD   INC $BD24     RWTS assumes data is stored
BD2B        C8   INY           in physical sector order on
BD2C        CA   DEX           each track. (This saves 18
BD2D     D0 F4   BNE $BD23     bytes: 16 for the table and 2
                               for the lookup command!)
BD2F  20 D5 BE   JSR $BED5     Values are the actual pages in
                               memory where that sector
*BED5L                         should go, and they get
                               zeroed once the sector is read
                               (so we don't waste time
                               decoding the same sector
                               twice).

BED5  20 E4 BE   JSR $BEE4     This routine reads nibbles
BED8     C9 D5   CMP #$D5      from disk until it finds the
BEDA     D0 F9   BNE $BED5     sequence "D5 AA", then it
BEDC  20 E4 BE   JSR $BEE4     reads one more nibble and
BEDF     C9 AA   CMP #$AA      returns it in the accumulator.
BEE1     D0 F5   BNE $BED8     We reuse this routine to find
BEE3        A8   TAY           both the address and data
BEE4  AD EC C0   LDA $C0EC ☺   field prologues.
BEE7     10 FB   BPL $BEE4
BEE9        60   RTS
```

Continuing from `$BD32`...

```
BD32     49 AD    EOR #$AD
BD34     F0 35    BEQ $BD6B
BD36  20 C2 BE    JSR $BEC2

*BEC2L

BEC2     A0 03    LDY #$03
BEC4  20 E4 BE    JSR $BEE4
BEC7        2A    ROL
BEC8  8D E0 BD    STA $BDE0
BECB  20 E4 BE    JSR $BEE4
BECE  2D E0 BD    AND $BDE0
BED1        88    DEY
BED2     D0 F0    BNE $BEC4

BED4        60    RTS
```

If that third nibble is not #$AD, we assume it's the end of the address prologue. (#$96 would be the third nibble of a standard address prologue, but we don't actually check.) We fall through and start decoding the 4-4 encoded values in the address field.

This routine parses the 4-4-encoded values in the address field. The first time through this loop, we'll read the disk volume number. The second time, we'll read the track number. The third time, we'll read the physical sector number. We don't actually care about the disk volume or the track number, and once we get the sector number, we don't verify the address field checksum.

On exit, the accumulator contains the physical sector number.

Continuing from $BD39...

```
BD39        A8    TAY

BD3A  BE 84 BF    LDX $BF84,Y

BD3D     F0 F0    BEQ $BD2F

BD3F  8D E0 BD    STA $BDE0

BD42  8E 64 BD    STX $BD64
BD45  8E C4 BD    STX $BDC4
BD48  8E 7C BD    STX $BD7C
BD4B  8E 8E BD    STX $BD8E
BD4E  8E A6 BD    STX $BDA6
BD51  8E BE BD    STX $BDBE
BD54        E8    INX
BD55  8E D9 BD    STX $BDD9
BD58        CA    DEX
BD59        CA    DEX
BD5A  8E 94 BD    STX $BD94
BD5D  8E AC BD    STX $BDAC

BD60     A0 FE    LDY #$FE
BD62  B9 02 D1    LDA $D102,Y
BD65        48    PHA
BD66        C8    INY
BD67     D0 F9    BNE $BD62

BD69     B0 C4    BCS $BD2F

BD6B     E0 00    CPX #$00
```

use physical sector number as an index into the sector address array

get the target page (where we want to store this sector in memory)

if the target page is #$00, it means we've already read this sector, so loop back to find the next address prologue

store the physical sector number later in this routine

store the target page in several places throughout this routine

Save the two bytes immediately after the target page, because we're going to use them for temporary storage. (We'll restore them later.)

this is an unconditional branch

execution continues here (from $BD34) after matching the data prologue

```
BD6D     F0 C0    BEQ $BD2F
```

If X is still #$00, it means we found a data prologue before we found an address prologue. In that case, we have to skip this sector, because we don't know which sector it is and we wouldn't know where to put it. Sad!

Nibble loop #1 reads nibbles $00..$55, looks up the corresponding offset in the preshift table at $BB96, and stores that offset in the temporary two-byte buffer after the target page.

```
BD6F  8D 7E BD    STA $BD7E

BD72  AE EC C0    LDX $C0EC
BD75     10 FB    BPL $BD72

BD77  BD 00 BB    LDA $BB00,X

BD7A  99 02 D1    STA $D102,Y

BD7D     49 D1    EOR #$D1

BD7F        C8    INY
BD80     D0 ED    BNE $BD6F
```

initialize rolling checksum to #$00, or update it with the results from the calculations below

read one nibble from disk

The nibble value is in the X register now. The lowest possible nibble value is $96 and the highest is $FF. To look up the offset in the table at $BB96, we index off $BB00 + X. Math!

Now the accumulator has the offset into the table of individual 2-bit combinations ($BC00..$BCFF). Store that offset in a temporary buffer towards the end of the target page. (It will eventually get overwritten by full 8-bit bytes, but in the meantime it's a useful $56-byte scratch space.)

The EOR value is set at $BD6F each time through loop #1.

The Y register started at #$AA (set by the "TAY" instruction at $BD39), so this loop reads a total of #$56 nibbles.

Here endeth nibble loop #1.

Nibble loop #2 reads nibbles $56..$AB, combines them with bits 0-1 of the appropriate nibble from the first $56, and stores them in bytes $00..$55 of the target page in memory.

```
BD82     A0 AA    LDY #$AA
BD84  AE EC C0    LDX $C0EC
BD87     10 FB    BPL $BD84
BD89  5D 00 BB    EOR $BB00,X
BD8C  BE 02 D1    LDX $D102,Y

BD8F  5D 02 BC    EOR $BC02,X

BD92  99 56 D1    STA $D156,Y

BD95        C8    INY
BD96     D0 EC    BNE $BD84
```

This address was set at $BD5A based on the target page (minus 1 so we can add Y from #$AA..#$FF).

Here endeth nibble loop #2.

Nibble loop #3 reads nibbles $AC..$101, combines them with bits 2-3 of the appropriate nib-

ble from the first $56, and stores them in bytes $56..$AB of the target page in memory.

```
BD98     29 FC   AND #$FC
BD9A     A0 AA   LDY #$AA
BD9C  AE EC C0   LDX $C0EC ↩
BD9F     10 FB   BPL $BD9C
BDA1  5D 00 BB   EOR $BB00,X
BDA4  BE 02 D1   LDX $D102,Y
↩
BDA7  5D 01 BC   EOR $BC01,X

BDAA  99 AC D1   STA $D1AC,Y   This address was set at $BD5D
↩                              based on the target page
BDAD        C8   INY           (minus 1 so we can add Y
BDAE     D0 EC   BNE $BD9C     from #$AA..#$FF).
```

Here endeth nibble loop #3.

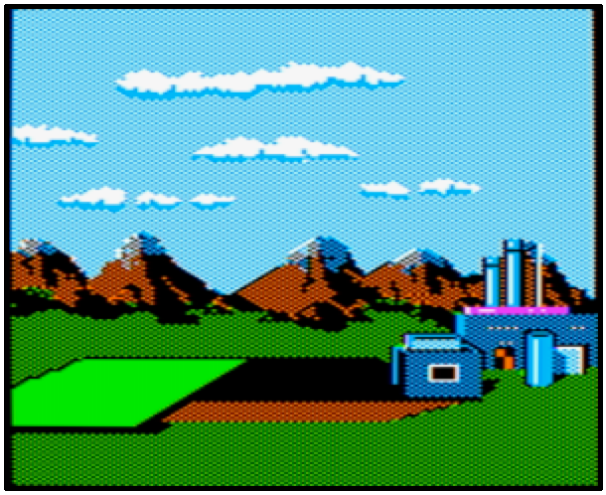
Loop #4 reads nibbles $102..$155, combines them with bits 4-5 of the appropriate nibble from the first $56, and stores them in bytes $AC..$101 of the target page in memory. (This overwrites two bytes after the end of the target page, but we'll restore then later from the stack.)

```
BDB0     29 FC   AND #$FC
BDB2     A2 AC   LDX #$AC
BDB4  AC EC C0   LDY $C0EC ↩
BDB7     10 FB   BPL $BDB4
BDB9  59 00 BB   EOR $BB00,Y
BDBC  BC 00 D1   LDY $D100,X
↩
BDBF  59 00 BC   EOR $BC00,Y

BDC2  9D 00 D1   STA $D100,X   This address was set at $BD45
↩                              based on the target page.
BDC5        E8   INX
BDC6     D0 EC   BNE $BDB4
```

Here endeth nibble loop #4.

```
BDC8     29 FC   AND #$FC      Finally, get the last nibble
BDCA  AC EC C0   LDY $C0EC ↩   and convert it to a byte. This
BDCD     10 FB   BPL $BDCA     should equal all the previous
BDCF  59 00 BB   EOR $BB00,Y   bytes XOR'd together. (This
                               is the standard checksum
                               algorithm shared by all
                               16-sector disks.)

BDD2     C9 01   CMP #$01      set carry if value is anything
                               but 0

BDD4     A0 01   LDY #$01      Restore the original data in
BDD6        68   PLA           the two bytes after the target
BDD7  99 00 D1   STA $D100,Y   page. (This does not affect
↩                              the carry flag, which we will
BDDA        88   DEY           check in a moment, but we
BDDB     10 F9   BPL $BDD6     need to restore these bytes
                               now to balance out the
                               pushing to the stack we did at
                               $BD65.)

BDDD     B0 8A   BCS $BD69     if data checksum failed at
                               $BDD2, start over

BDDF     A0 D1   LDY #$D1 ↩    This was set to the physical
BDE1        8A   TXA           sector number (at $BD3F), so
                               this is a index into the
                               16-byte array at $BF84.

BDE2  99 84 BF   STA $BF84,Y   store #$00 at this location in
                               the sector array to indicate
                               that we've read this sector

BDE5  CE EF BD   DEC $BDEF     decrement sector count
BDE8  CE 94 BF   DEC $BF94
BDEB        38   SEC

BDEC     D0 EF   BNE $BDDD     If the sectors-left-in-this-track
                               count (in $BF94) isn't zero
                               yet, loop back to read more
                               sectors.

BDEE     A2 D1   LDX #$D1 ↩    If the total sector count (in
BDF0     F0 09   BEQ $BDFB     $BDEF, set at $BD04 and
                               decremented at $BDE5) is zero,
                               we're done—no need to read
                               the rest of the track. (This
                               lets us have sector counts that
                               are not multiples of 16, i.e.
                               reading just a few sectors
                               from the last track of a
                               multi-track block.)

BDF2  EE 10 BF   INC $BF10     increment phase (twice, so it
BDF5  EE 10 BF   INC $BF10     points to the next whole
                               block)

BDF8  4C 10 BD   JMP $BD10     jump back to seek and read
                               from the next track

BDFB  AD E8 C0   LDA $C0E8 ↩   Execution continues here
BDFE        60   RTS           (from $BDEF). We're all done,
                               so turn off drive motor and
                               exit.
```

And that's all she wrote^H^H^H^Hread.



I Make My Verse For The Universe

How's our master plan from page 47 going? Pretty darn well, I'd say.

Step 1) write all the game code to a standard disk. Done.

Step 2) write an RWTS. Done.

Step 3) make them talk to each other.

The "glue code" for this final step lives on track 1. It was loaded into memory at the very end of the boot sector:

```
089B-   A9 01      LDA   #$01
089D-   A2 03      LDX   #$03
089F-   A0 B0      LDY   #$B0
08A1-   4C 00 BD   JMP   $BD00
```

That loads 3 sectors from track 1 into `$B000..$B2FF`. `$B000` is the high scores, which stays at `$B000`. `$B100` is moved to zero page. `$B200` is the output vector and final initialization code. This page is never used by the game. (It was used by the original RWTS, but that has been greatly simplified by stripping out the copy protection. I love when that happens!)

Here is my output vector, replacing the code that originally lived at `$BF6F`:

```
*B200L
B200     C9 07    CMP #$07       command or regular
                                 character?
B202     90 03    BCC $B207      command -> branch

B204  6C 3A 00    JMP ($003A)    regular character -> print to
                                 screen
B207     85 5F    STA $5F        store command in zero page

B209        A8    TAY            set up the call to the screen
B20A  B9 97 B2    LDA $B297,Y    fill
B20D  8D 19 B2    STA $B219

B210  B9 9E B2    LDA $B29E,Y    set up the call to Gumboot
B213  8D 1C B2    STA $B21C

B216     A9 00    LDA #$00       call the appropriate screen fill
B218  20 69 B2    JSR $B269

B21B  20 2B B2    JSR $B22B      call Gumboot

B21E     A5 5F    LDA $5F        find the entry point for this
B220        0A    ASL            block
B221        A8    TAY

B222  B9 A6 B2    LDA $B2A6,Y    push the entry point to the
B225        48    PHA            stack
B226  B9 A5 B2    LDA $B2A5,Y
B229        48    PHA

B22A        60    RTS            and exit via "RTS"
```

This is the routine that calls Gumboot to load the appropriate blocks of game code from the disk, according to the disk map on page 47. Here is the summary of which sectors are loaded by each block:

cmd	track (A)	count (X)	page (Y)
$00	$02	$38	$08
	$06	$28	$60
$01	$09	$38	$08
	$0D	$50	$60
$02	$12	$38	$08
	$16	$28	$60
$03	$19	$20	$20

(The parameters for command `#$06` are the same as command `#$01`.)

The lookup at `$B210` modified the "JSR" instruction at `$B21B`, so each command starts in a different place:

```
B22B     A9 02    LDA #$02       command #$00
B22D  20 56 B2    JSR $B256
B230     A9 06    LDA #$06
B232     D0 1C    BNE $B250

B234     A9 09    LDA #$09       command #$01
B236  20 56 B2    JSR $B256
B239     A9 0D    LDA #$0D
B23B     A2 50    LDX #$50
B23D     D0 13    BNE $B252

B23F     A9 12    LDA #$12       command #$02
B241  20 56 B2    JSR $B256
B244     A9 16    LDA #$16
B246     D0 08    BNE $B250

B248     A9 19    LDA #$19       command #$03
B24A     A2 20    LDX #$20
B24C     A0 20    LDY #$20
B24E     D0 0A    BNE $B25A
B250     A2 28    LDX #$28
B252     A0 60    LDY #$60
B254     D0 04    BNE $B25A
B256     A2 38    LDX #$38
B258     A0 08    LDY #$08
B25A  4C 00 BD    JMP $BD00

B25D     A9 01    LDA #$01       command #$04: seek to track
B25F  20 00 BF    JSR $BF00      1 and write $B000..$B0FF to
B262     A9 00    LDA #$00       sector 0
B264     A0 B0    LDY #$B0
B266  4C 00 BE    JMP $BE00
```

```
B269     A5 60   LDA $60       exact replica of the screen fill
B26B  4D 50 C0   EOR $C050     code that was originally at
B26E     85 60   STA $60       $BEB0
B270     29 0F   AND #$0F
B272     F0 F5   BEQ $B269
B274     C9 0F   CMP #$0F
B276     F0 F1   BEQ $B269
B278  20 66 F8   JSR $F866
B27B     A9 17   LDA #$17
B27D        48   PHA
B27E  20 47 F8   JSR $F847
B281     A0 27   LDY #$27
B283     A5 30   LDA $30
B285     91 26   STA ($26),Y
B287        88   DEY
B288     10 FB   BPL $B285
B28A        68   PLA
B28B        38   SEC
B28C     E9 01   SBC #$01
B28E     10 ED   BPL $B27D
B290  AD 56 C0   LDA $C056
B293  AD 54 C0   LDA $C054
B296        60   RTS



B297 [69 7B 69 69 96 96 69]    lookup table for screen fills

B29E [2B 34 3F 48 2A 2A 34]    lookup table for Gumboot
                               calls

B2A5 [9C 0F]                   lookup table for entry points
B2A7 [F8 31]
B2A9 [34 10]
B2AB [57 FF]
B2AD [5C B2]
B2AF [95 B2]
B2B1 [77 23]
```

Last but not least, a short routine at `$B2F1` to move zero page into place and start the game. (This is called because we pushed `#$B2/#$F0` to the stack in our boot sector, at `$0895`.)

```
*B2F1L
B2F1     A2 00   LDX #$00       copy $B100 to zero page
B2F3  BD 00 B1   LDA $B100,X
B2F6     95 00   STA $00,X
B2F8        E8   INX
B2F9     D0 F8   BNE $B2F3


B2FB     A9 00   LDA #$00       print a null character to start
B2FD  4C ED FD   JMP $FDED     the game
```

Quod erat liberand one more thing...

Oops

Heeeeey there. Remember this code?

```
0372     A9 34   LDA #$34
0374        48   PHA
...
0378        28   PLP
```

Here's what I said about it when I first saw it:

> pop that #$34 off the stack, but use it as status registers (weird, but legal—if it turns out to matter, I can figure out exactly which status bits get set and cleared)

Yeah, so that turned out to be more important than I thought. After extensive play testing, we[26] discovered the game becomes unplayable on level 3.

How unplayable? Gates that are open won't close; balls pass through gates that are already closed; bins won't move more than a few pixels.

So, not a crash, and (contrary to our first guess) not an incompatibility with modern emulators. It affects real hardware too, and it was intentional. Deep within the game code, there are several instances of code like this:

```
T0A,S00
---------- DISASSEMBLY MODE ----------
0021:08                 PHP
0022:68                 PLA
0023:29 04              AND    #$04
0025:D0 0A              BNE    $0031
0027:A5 18              LDA    $18
0029:C9 02              CMP    #$02
002B:90 04              BCC    $0031
002D:A9 10              LDA    #$10
002F:85 79              STA    $79
0031:A5 79              LDA    $79
0033:85 7A              STA    $7A
```

"PHP" pushes the status registers on the stack, but "PLA" pulls a value from the stack and stores it as a byte, in the accumulator. That's... weird. Also, it's the reverse of the weird code we saw at `$0372`, which took a byte in the accumulator and blitted it into the status registers. Then "`AND #$04`" isolates one status bit in particular: the interrupt flag. The rest of the code is the game-specific way of making the game unplayable.

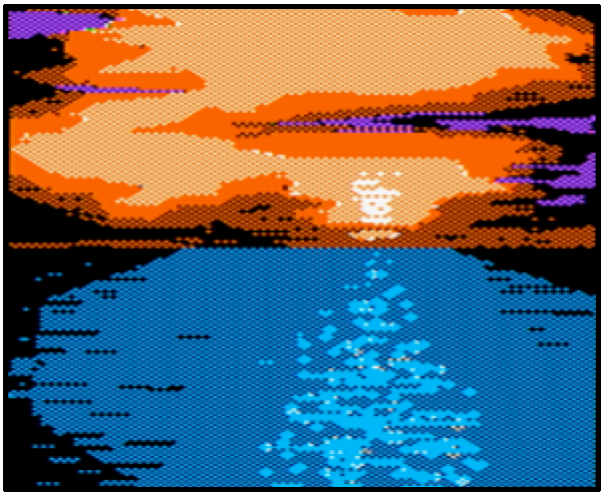
This is a very convoluted, obfuscated, sneaky way to ensure that the game was loaded through its original bootloader. Which, of course, it wasn't.

The solution: after loading each block of game code and pushing the new entry point to the stack, set the interrupt flag.

```
B222  B9 A6 B2   LDA $B2A6,Y   pop that #$34 off the stack,
B225        48   PHA           but use it as status registers
B226  B9 A5 B2   LDA $B2A5,Y   (weird, but legal—if it turns
B229        48   PHA           out to matter, I can figure out
                               exactly which status bits get
                               set and cleared) push the
                               entry point to the stack

B22A        78   SEI           set the interrupt flag (new!)


B22B        60   RTS           and exit via "RTS"
```

Many thanks to Marco V. for reporting this and helping reproduce it; qkumba for digging into it to find the check within the game code; Tom G. for making the connection between the interrupt flag and the weird "LDA/PHA/PLP" code at `$0372`.

[26] not me, and not qkumba either, who beat the entire game twice. It was Marco V. Thanks, Marco!



This Is Not The End, Though

This game holds one more secret, but it's not related to the copy protection, thank goodness. As far as I can tell, this secret has not been revealed in 33 years. qkumba found it because of course he did.

Once the game starts, press `Ctrl-J` to switch to joystick mode. Press and hold button 2 to activate "targeting" mode, then move your joystick to the bottom-left corner of the screen and also press button 1. The screen will be replaced by this message:

PRESS CTRL-Z DURING THE CARTOONS

Now, the game has 5 levels. After you complete a level, your character gets promoted: worker, foreman, supervisor, manager, and finally vice president. Each of these is a little cartoon—what kids today would call a cut scene. When you complete the entire game, it shows a final screen and your character retires.

Pressing `Ctrl-Z` during each cartoon reveals four ciphers.

After level 1:

RBJRY JSYRR

After level 2:

VRJJRY ZIAR

After level 3:

ESRB

After level 4:

FIG YRJMYR

Taken together, they form a simple substitution cipher:

- ENTER THREE
- LETTER CODE
- WHEN
- YOU RETIRE

But what is the code?

It turns out that pressing `Ctrl-Z` *again*, while any of the pieces of the cipher are on screen, reveals another clue:

DOUBLE HELIX

Entering the three-letter code `DNA` at the "retirement" screen reveals the final secret message:

```
AHA!  YOU MADE IT!
EITHER YOU ARE AN EXCELLENT GAME-PLAYER
OR (GAH!) PROGRAM-BREAKER!
YOU ARE CERTAINLY ONE OF THE FEW PEOPLE
THAT WILL EVER SEE THIS SCREEN.

THIS IS NOT THE END, THOUGH.

IN ANOTHER BR0DERBUND PRODUCT
TYPE 'Z0DWARE' FOR MORE PUZZLES.

HAVE FUN!  BYE!!

                          R.A.C.
```

At time of writing, no one has found the "Z0DWARE" puzzle. You could be the first!

Keys and Controls

The game can be played with a joystick or keyboard.

`Ctrl-J` switch to joystick mode

`Ctrl-K` switch to keyboard mode

When using a keyboard:

`S` move bins left

`D` stop bins

`F` move bins right

`Space` switch in-tube gates

`E` increase speed

`C` decrease speed

`Return` toggle target sighting

`U I O` move the target sight

`J K L` (for when the bombs

`M , .` start dropping)

When using a joystick:

`buttons 0+1` toggle target sighting

`Ctrl-X` flip joystick X axis

`Ctrl-Y` flip joystick Y axis

Other keys:

`Ctrl-S` toggle sound on/off

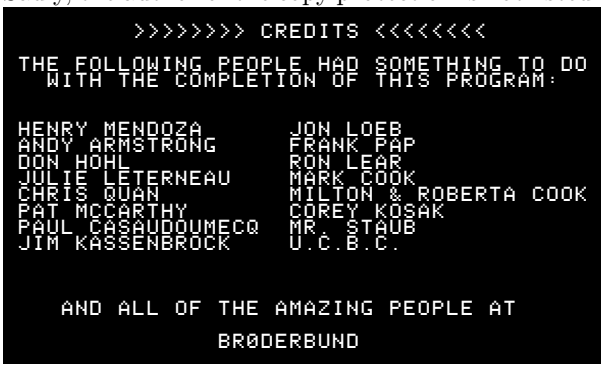
`Ctrl-R` restart level

`Ctrl-Q` restart game

`Ctrl-H` view high scores

`Esc` pause/resume game

After the game starts, press `Ctrl-U Ctrl-C Ctrl-B` in sequence to see a secret credits page that lists most of the people involved in making the game. Sadly, the author of the copy protection is not listed.



Cheats

I have not enabled any cheats on our release, but I have verified that they work. You can use any or all of them:

Stop the clock

`T09,S0A,$B1`

change `01` to `00`

Start on level 2-5

`T09,S0C,$53`

change `00` to <level-1>

Acknowledgements

Thanks to Alex, Andrew, John, Martin, Paul, Quinn, and Richard for reviewing drafts of this write-up.

And finally, many thanks to qkumba: Shifter of Bits, Master of the Stack, author of Gumboot, and my friend.

15:07 In Which a PDF is a Git Repository Containing its own LaTeX Source and a Copy of Itself

by Evan Sultanik

Have you ever heard of the `git bundle` command? I hadn't. It bundles a set of Git objects—potentially even an entire repository—into a single file. Git allows you to treat that file as if it were a standard Git database, so you can do things like clone a repo directly from it. Its purpose is to easily sneakernet pushes or even whole repositories across air gaps.

Neighbors, it's possible to create a PDF that is also a Git repository.

```
$ git clone PDFGitPolyglot.pdf foo
Cloning into 'foo'...
Receiving objects: 100% (174/174), 103.48 KiB, done.
Resolving deltas: 100% (100/100), done.
$ cd foo
$ ls
PDFGitPolyglot.pdf PDFGitPolyglot.tex
```

15:07.1 The Git Bundle File Format

The file format for Git bundles doesn't appear to be formally specified anywhere, however, inspecting `bundle.c` reveals that it's relatively straightforward:

Git has another custom format called a *Packfile* that it uses to compress the objects in its database, as well as to reduce network bandwidth when pushing and pulling. The packfile is therefore an obvious choice for storing objects inside bundles. This of course raises the question: What is the format for a Git Packfile?

Git does have some internal documentation in

`Documentation/technical/pack-format.txt`

however, it is rather sparse, and does not provide enough detail to fully parse the format. The documentation also has some "observations" that suggest it wasn't even written by the file format's creator and instead was written by a developer who was later trying to make sense of the code.

Luckily, Aditya Mukerjee already had to reverse engineer the file format for his GitGo clean-room implementation of Git, and he wrote an excellent blog entry about it.[27]

Although not entirely required to understand the polyglot, I think it is useful to describe the git packfile format here, since it is not well documented elsewhere. If that doesn't interest you, it's safe to skip to the next section. But if you do proceed, I hope you like Soviet holes, dear neighbor, because chasing this rabbit might remind you of Кольская.

[27] https://codewords.recurse.com/issues/three/unpacking-git-packfiles

Right, the next step is to figure out the "chunk" format. The chunk header is variable length, and can be as small as one byte. It encodes the object's type and its *uncompressed* size. If the object is a *delta* (*i.e.*, a diff, as opposed to a complete object), the header is followed by either the SHA-1 hash of the base object to which the delta should be applied, or a byte reference within the packfile for the start of the base object. The remainder of the chunk consists of the object data, zlib-compressed.

The format of the variable length chunk header is pictured in Figure 4. The second through fourth most significant bits of the first byte are used to store the object type. The remainder of the bytes in the header are of the same format as bytes two and three in this example. This example header represents an object of type 11_2, which happens to be a git blob, and an *uncompressed* length of $(100_2 << 14) + (1010110_2 << 7) + 1001001_2 = 76{,}617$ bytes. Since this is not a delta object, it is immediately followed by the zlib-compressed object data. The header does not encode the *compressed* size of the object, since the DEFLATE encoding can determine the end of the object as it is being decompressed.

At this point, if you found The Life and Opinions of Tristram Shandy to be boring or frustrating, then it's probably best to skip to the next section, 'cause it's turtles all the way down.

> "To come at the exact weight of things in the ſcientific ſteel-yard, the fulchrum, [Walter Shandy] would ſay, ſhould be almoſt inviſible, to avoid all friction from popular tenets;—without this the minutiæ of philoſophy, which ſhould always turn the balance, will have no weight at all. Knowledge, like matter, he would affirm, was diviſible in infinitum;—that the grains and ſcruples were as much a part of it, as the gravitation of the whole world."

There are two types of delta objects: *references* (object type 7) and *offsets* (object type 6). Reference delta objects contain an additional 20 bytes at the end of the header before the zlib-compressed delta data. These 20 bytes contain the SHA-1 hash of the base object to which the delta should be applied. Offset delta objects are exactly the same, however, instead of referencing the base object by its SHA-1 hash, it is instead represented by a negative byte offset to the start of the object within the pack file. Since a negative byte offset can typically be encoded in two or three bytes, it's significantly smaller than a 20-byte SHA-1 hash. One must understand how these offset delta objects are encoded if—say, for some strange, masochistic reason—one wanted to change the order of objects within a packfile, since doing so would break the negative offsets. (Foreshadowing!)

One would *think* that git would use the same multi-byte length encoding that they used for the uncompressed object length. But no! This is what we have to go off of from the git documentation:

```
n bytes with MSB set in all but the last one.
The offset is then the number constructed by
concatenating the lower 7 bit of each byte, and
for n >= 2 adding 2^7 + 2^14 + ... + 2^(7*(n-1))
to the result.
```

Right. Some experimenting resulted in the following decoding logic that appears to work:

```
def decode_obj_ref(data):
    bytes_read = 0
    reference = 0
    for c in map(ord, data):
        bytes_read += 1
        reference <<= 7
        reference += c & 0b01111111
        if not (c & 0b10000000):
            break
    if bytes_read >= 2:
        reference += (1 << (7 * (bytes_read - 1)))
    return reference, bytes_read
```

The rabbit hole is deeper still; we haven't yet discovered the content of the compressed delta objects, let alone how they are applied to base objects. At this point, we have more than sufficient knowledge to proceed with the PoC, and my canary died ages ago. Aditya Mukerjee did a good job of explaining the process of applying deltas in his blog post, so I will stop here and proceed with the polyglot.

15:07.2 A Minimal Polyglot PoC

We now know that a git bundle is really just a git packfile with an additional header, and a git packfile stores individual objects using zlib, which uses the DEFLATE compression algorithm. DEFLATE supports zero compression, so if we can store the PDF in a single object (as opposed to it being split into deltas), then we could theoretically coerce it to be intact within a valid git bundle.

Forcing the PDF into a single object is easy: We just need to add it to the repo last, immediately before generating the bundle.

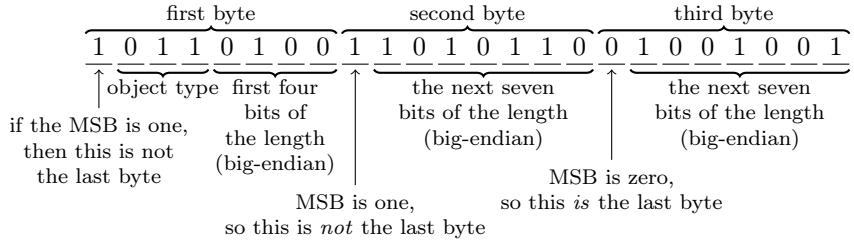


Figure 4. Format of the git packfile's variable length chunk header.

Getting the object to be compressed with zero compression is also relatively easy. That's because git was built in almost religious adherence to The UNIX Philosophy: It is architected with hundreds of sub commands it calls "plumbing," of which the vast majority you will likely have never heard. For example, you might be aware that `git pull` is equivalent to a `git fetch` followed by a `git merge`. In fact, the `pull` code actually spawns a new `git` child process to execute each of those subcommands. Likewise, the `git bundle` command spawns a `git pack-objects` child process to generate the packfile portion of the bundle. All we need to do is inject the `--compression=0` argument into the list of command line arguments passed to `pack-objects`. This is a one-line addition to `bundle.c`:

```
argv_array_pushl(
    &pack_objects.args,
    "pack-objects", "--all-progress-implied",
    "--compression=0",
    "--stdout", "--thin", "--delta-base-offset",
    NULL);
```

Using our patched version of git, every object stored in the bundle will be uncompressed!

```
$ export PATH=/path/to/patched/git:$PATH
$ git init
$ git add article.pdf
$ git commit article.pdf -m "added"
$ git bundle create PDFGitPolyglot.pdf --all
```

Any vanilla, un-patched version of git will be able to clone a repo from the bundle. It will also be a valid PDF, since virtually all PDF readers ignore garbage bytes before and after the PDF.

15:07.3 Generalizing the PoC

There are, of course, several limitations to the minimal PoC given in the previous section:

1. Adobe, being Adobe, will refuse to open the polyglot unless the PDF is version 1.4 or earlier. I guess it doesn't like some element of the git bundle signature or digest if it's PDF 1.5. Why? Because Adobe, that's why.

2. Leaving the entire Git bundle uncompressed is wasteful if the repo contains other files; really, we only need the PDF to be uncompressed.

3. If the PDF is larger than 65,535 bytes—the maximum size of an uncompressed DEFLATE block—then git will inject 5-byte deflate block headers inside the PDF, likely corrupting it.

4. Adobe will also refuse to open the polyglot unless the PDF is near the beginning of the packfile.[28]

The first limitation is easy to fix by instructing LaTeX to produce a version 1.4 PDF by adding `\pdfminorversion=4` to the document.

The second limitation is a simple matter of software engineering, adding a command line argument to the `git bundle` command that accepts the hash of the single file to leave uncompressed, and passing that hash to `git pack-objects`. I have created a fork of git with this feature.[29]

As an aside, while fixing the second limitation I discovered that if a file has multiple PDFs concatenated after one another (*i.e.*, a git bundle polyglot with multiple uncompressed PDFs in the repo), then the behavior is viewer-dependent: Some viewers will render the first PDF, while others will render the last. That's a fun way to generate a PDF that displays completely different content in, say, macOS Preview versus Adobe.

The third limitation is very tricky, and ultimately why this polyglot was not used for the PDF

[28] Requiring the PDF header to start near the beginning of a file is common for many, but not all, PDF viewers.
[29] `https://github.com/ESultanik/git/tree/UncompressedPack`

of this issue of PoC||GTFO. I've a solution, but it will not work if the PDF contains any objects (*e.g.*, images) that are larger than 65,535 bytes. A universal solution would be to break up the image into smaller ones and tile it back together, but that is not feasible for a document the size of a PoC||GTFO issue.

DEFLATE headers for uncompressed blocks are very simple: The first byte encodes whether the following block is the last in the file, the next two bytes encode the block length, and the last two bytes are the ones' complement of the length. Therefore, to resolve this issue, all we need to do is move all of the DEFLATE headers that zlib created to different positions that won't corrupt the PDF, and update their lengths accordingly.

Where can we put a 5-byte DEFLATE header such that it won't corrupt the PDF? We could use our standard trick of putting it in a PDF object stream that we've exploited countless times before to enable PoC||GTFO polyglots. The trouble with that is: Object streams are fixed-length, so once the PDF is decompressed (*i.e.*, when a repo is cloned from the git bundle), then all of the 5-byte DEFLATE headers will disappear and the object stream lengths would all be incorrect. Instead, I chose to use PDF comments, which start at any occurrence of the percent sign character (%) outside a string or stream and continue until the first occurrence of a newline. All of the PDF viewers I tested don't seem to care if comments include non-ASCII characters; they seem to simply scan for a newline. Therefore, we can inject "`%\n`" between PDF objects and move the DEFLATE headers there. The only caveat is that the DEFLATE header itself can't contain a newline byte (`0x0A`), otherwise the comment would be ended prematurely. We can resolve that, if needed, by adding extra spaces to the end of the comment, increasing the length of the following DEFLATE block and thus increasing the length bytes in the DEFLATE header and avoiding the `0x0A`. The only concession made with this approach is that PDF Xref offsets in the deflated version of the PDF will be off by a multiple of 5, due to the removed DEFLATE headers. Fortunately, most PDF readers can gracefully handle incorrect Xref offsets (at the expense of a slower loading time), and this will only affect the PDF contained in the repository, *not* the PDF polyglot.

As a final step, we need to update the SHA-1 sum at the end of the packfile (*q.v.* Section 15:07.1), since we moved the locations of the DEFLATE headers, thus affecting the hash.

At this point, we have all the tools necessary to create a generalized PDF/Git Bundle polyglot for *almost* any PDF and git repository. The only remaining hurdle is that some viewers require that the PDF occur as early in the packfile as possible. At first, I considered applying another patch directly to the git source code to make the uncompressed object first in the packfile. This approach proved to be very involved, in part due to git's UNIX design philosophy and architecture of generic code reuse. We're already updating the packfile's SHA-1 hash due to changing the DEFLATE headers, so instead I decided to simply reorder the objects after-the-fact, subsequent to the DEFLATE header fix but before we update the hash. The only challenge is that moving objects in the packfile has the potential to break offset delta objects, since they refer to their base objects via a byte offset within the packfile. Moving the PDF to the beginning will break any offset delta objects that occur after the original position of the PDF that refer to base objects that occur before the original position of the PDF. I originally attempted to rewrite the broken offset delta objects, which is why I had to dive deeper into the rabbit hole of the packfile format to understand the delta object headers. (You saw this at the end of Section 15:07.1, if you were brave enough to finish it.) Rewriting the broken offset delta objects is the *correct* solution, but, in the end, I discovered a much simpler way.

> "As a matter of fact, G-d just questioned my judgment. He said, 'Terry, are you worthy to be the man who makes The Temple? If you are, you must answer: Is this [dastardly], or is this divine intellect?'"
>
> —Terry A. Davis, creator of TempleOS self-proclaimed "smartest programmer that's ever lived"

Terry's not the only one who's written a compiler!

In the previous section, recall that we created the minimal PoC by patching the command line arguments to `pack-objects`. One of the command line arguments that is already passed by default is `--delta-base-offset`. Running `git help pack-objects` reveals the following:

```
A packed archive can express the base object
of a delta as either a 20-byte object name
or as an offset in the stream, but ancient
versions of Git don't understand the latter.
By default, git pack-objects only uses the
former format for better compatibility. This
option allows the command to use the latter
format for compactness. Depending on the
average delta chain length, this option
typically shrinks the resulting packfile by
3-5 per-cent.
```

So all we need to do is *remove* the `--delta-base-offset` argument and git will not include any offset delta objects in the pack!

Okay, I have to admit something: There is one more challenge. You see, the PDF standard (ISO 32000-1) says

> “ The *trailer* of a PDF file enables a conforming reader to quickly find the cross-reference table and certain special objects. Conforming readers should read a PDF file from its end. The last line of the file shall contain only the end-of-file marker, `%%EOF`. ”

Granted, we are producing a PDF that conforms to version 1.4 of the specification, which doesn't appear to have that requirement. However, at least as early as version 1.3, the specification did have an implementation note that Acrobat requires the `%%EOF` to be within the last 1024 bytes of the file. Either way, that's not guaranteed to be the case for us, especially since we are moving the PDF to be at the beginning of the packfile. There are always going to be at least 20 trailing bytes after the PDF's `%%EOF` (namely the packfile's final SHA-1 checksum), and if the git repository is large, there are likely to be more than 1024 bytes.

Fortunately, most common PDF readers don't seem to care how many trailing bytes there are, at least when the PDF is version 1.4. Unfortunately, some readers such as Adobe's try to be “helpful,” silently “fixing” the problem and offering to save the fixed version upon exit. We can at least partially fix the PDF, ensuring that the `%%EOF` is exactly 20 bytes from the end of the file, by creating a second uncompressed git object as the very end of the packfile (right before the final 20 byte SHA-1 checksum). We could then move the trailer from the end of the original PDF at the start of the pack to the new git object at the end of the pack. Finally, we could encapsulate the “middle” objects of the packfile inside a PDF stream object, such that they are ignored by the PDF. The tricky part is that we would have to know how many bytes will be in that stream *before* we add the PDF to the git database. That's theoretically possible to do *a priori*, but it'd be very labor intensive to pull off. Furthermore, using this approach will completely break the inner PDF that is produced by cloning the repository, since its trailer will then be in a separate file. Therefore, I chose to live with Adobe's helpfulness and not pursue this fix for the PoC.

The feelies contain a standalone PDF of this article that is also a git bundle containing its LaTeX source, as well as all of the code necessary to regenerate the polyglot.[30] Clone it to take a look at the history of this article and its associated code! The code is also hosted on GitHub[31].

> Thus—thus, my fellow-neighbours and aſſociates in this great harveſt of our learning, now ripening before our eyes; thus it is, by ſlow ſteps of caſual increaſe, that our knowledge phyſical, metaphyſical, phyſiological, polemical, nautical, mathematical, ænigmatical, technical, biographical, romantical, chemical, obſtetrical, and polyglottical, with fifty other branches of it, (moſt of 'em ending as theſe do, in ical) have for theſe four laſt centuries and more, gradually been creeping upwards towards that Akme of their perfections, from which, if we may form a conjecture from the advances of theſe laſt 5 pages, we cannot poſſibly be far off.

[30] `unzip pocorgtfo15.pdf PDFGitPolyglot.pdf`
[31] `https://github.com/ESultanik/PDFGitPolyglot`

Cyberencabulator

FUNCTION

To **measure** inverse reactive current in universal phase detractors with display of percent realization.

OPERATION

Based on the principle of power generation by the modial interaction of magnetoreluctance and capacitative diractance, the Cyberencabulator negates the relative motion of conventional conductors and fluxes. It consists of a baseplate of prefabulated Amulite, surmounted by a malleable logarithmic casing in such a way that the two main spurving bearings are aligned with the parametric fan.

Six gyro-controlled antigravic marzelvanes are attached to the ambifacent wane shafts to prevent internal precession. Along the top, adjacent to the panandermic semi-boloid stator slots, are forty-seven manestically spaced grouting brushes, insulated with Glyptal-impregnated, cyanoethylated kraft paper bushings. Each one of these feeds into the rotor slip-stream, via the non-reversible differential tremie pipes, a 5 per cent solution of reminative Tetraethyliodohexamine, the specific pericosity of which is given by $P = 2.5C_n^{6\div7}$, where "C" is Chlomondeley's annular grillage coefficient and "n" is the diathetical evolute of retrograde temperature phase disposition.

The two panel meters display inrush current and percent realization. In addition, whenever a barescent skor motion is required, it may be employed with a reciprocating dingle arm to reduce the sinusoidal depleneration in nofer trunions.

Solutions are checked via Zahn Viscosimetry techniques. Exhaust orifices receive standard Blevinometric tests. There is no known Orth Effect.

TECHNICAL FEATURES

- Panandermic semi-boloid stator slots
- Panel meter covers treated with Shure Stat (guaranteed to build up electrostatic charge in less than 1 second).
- Manestically spaced grouting brushes
- Prefabulated Amulite baseplate
- Pentametric fan

STANDARD RATINGS

Rating	Old Catalog No.	New Computer Insensitive Catalog No.
0–1024	8080808G6S*	25504446POC1†

* Included Qty. 6 NO-BLO‡ fuses.

† Includes Magnaglas circuit breaker with polykrapolene-coated contacts rated 75A Wolfram.

‡ Reg. T.M. Shenzhen Xiao Baoshi Electronics Co., Ltd.

ACCESSORIES

1. 8 ounces 5 per cent Tetraethyliodohexamine with 0.01N Halogen tracer solution.
2. Interelectrode diffusion integrator.
3. Noninductive-wound inverse conductance control in little black box.
4. Analog to digital converter with reflected levorotatory BCD output (binary-coded decimal *i.e.*: 7, 4, 2, 1).
5. Quasistatic regeneration oscillator with output conductance of 17.8 millimhos.

APPLICATION

Measuring Inverse Reactive Current—CAUTION: Because of the replenerative flow characteristics of positive ions in unilateral phase detractors, the use of the quasistatic regeneration oscillator is recommended if Cyberencabulator is used outside of an air conditioned server room.

Reduction of Sinusoidal Depleneration—Before use, the system should be calibrated with a gyro-controlled Sine-Wave Director, the output of which should be of the cathode follower type.

Note: If only Cosine-Wave Directors are available, their output must be first fed into a Phase Inverter with parametric negative-time compensators. **Caution:** Only Phase Inverters with an output conductance of 17.8 ± 1 millimhos should be employed so as to match the characteristics of the quasistatic regeneration oscillator.

Voltage Levels—Above 750V **Do Not Use** Caged Resistors to get within self-contained rating of Cyberencabulator. **Do Use** Sequential Transformers. See POC-9001.

Multiple Ratings—Optionally available in multiples of π ($^{22}/_{7}$) and e ($^{19}/_{7}$). If binary or other number-base systems ratios are required, refer to the fuctoría for availability and pricing.

Goniometric Data—Upon request, curves are supplied, at additional charge, for regions wherein the molecular MFP (Mean Free Path) is between 1.6 and 19.62 Angstrom units. Curves, relevant to regions outside the above-listed range, may be obtained from:

Tract Association of PoC||GTFO and Friends, GmbH
Cloud Computing Cyberencabulator Dept. (C^3D)
Tennessee, 'Murrica

In Canada address request to:

Cyberencabulateurs
Canaderpien-Français Ltée.
468 Jean de Quen, Quebec 10, P.Q.

Reference Texts

1. Zeitschrift für Physik
   Der Zerfall von Dunge LBM-1
   H. Sturtzkampflieger, Berlin, DDR
2. Svenska Teckniska Skatologika Lärovarken
   Dagblad 121–G. Petterson & W. Johannson, Stockholm
3. Journaux de l'Academie Française
   Numero 606B
   T. L'Ouverture, Paris
4. Szkola Polska
   Cyberencabulatorskiego
   Ogłoszenie 1411–7
   Iwan Jędrek S., Rzeżuśnia
5. Texas Inst. of Cyberencabulation
   AITE Bull. 312–52, J. J. Fleck, Dallas.
6. THE VISE №7
   AvE, Canuckistan
7. Хроника Технологических Событий
   Святейший Маноль Лафройг

SPECIFICATIONS

Accuracy: ±1 per cent of point

Repeatability: ±1/4 per cent

Maintenance Required: Bimonthly treatment of Meter covers with Shure Stat.

Ratings: None (Standard); All (Optional)

Fuel Efficiency: 1.337 Light-Years per Sydharb

Input Power: Volts—120/240/480/550 AC
Amps—10/5/2.5/2.2 A
Watts—1200 W
Wave Shape—Sinusoidal, Cosinusoidal, Tangential, or Pipusoidal.

Operating Environment:
Temperature 32F to 150F (0C to 66C)

Max Magnetic Field: 15 Mendelsohns
(1 Mendelsohn = 32.6 Statoersteds)

Case: Material: Amulite; Tremie-pipes are of Chinesium—(Tungsten Cowhide)

Weight: Net 134 lbs.; Ship 213 lbs.

DIMENSION DRAWINGS

On delivery.

EXTERNAL WIRING

On delivery.

15:08 Zero Overhead Networking

by Robert Graham

The kernel is a religion. We programmers are taught to let the kernel do the heavy lifting for us. We the lay folks are taught how to propitiate the kernel spirits in order to make our code go faster. The priesthood is taught to move their code into the kernel, as that is where speed happens.

This is all a lie. The true path to writing high-speed network applications, like firewalls, intrusion detection, and port scanners, is to completely bypass the kernel. Disconnect the network card from the kernel, memory map the I/O registers into user space, and DMA packets directly to and from user-mode memory. At this point, the overhead drops to near zero, and the only thing that affects your speed is you.

Masscan

Masscan is an Internet-scale port scanner, meaning that it can scan the range `/0`. By default, with no special options, it uses the standard API for raw network access known as `libpcap`. `Libpcap` itself is just a thin API on top of whatever underlying API is needed to get raw packets from Linux, macOS, BSD, Windows, or a wide range of other platforms.

But Masscan also supports another way of getting raw packets known as `PF_RING`. This runs the driver code in user-mode. This allows Masscan to transmit packets by sending them directly to the network hardware, bypassing the kernel completely (no memory copies, no kernel calls). Just put "zc:" (meaning `PF_RING` ZeroCopy) in front of an adapter name, and Masscan will load `PF_RING` if it exists and use that instead of `libpcap`.

In the section below, we are going to analyze the difference in performance between these two methods. On the test platform, Masscan transmits at 1.5 million packets-per-second going through the kernel, and trasnmits at 8 million packets-per-second when going though `PF_RING`.

We are going to run the Linux profiling tool called `perf` to find out where the CPU is spending all its time in both scenarios.

Raw output from `perf` is difficult to read, so the results have been processed through Brendan Gregg's FlameGraph tool. This shows the call stack of every sample it takes, showing the total time in the caller as well as the smaller times in each function called, in the next layer. This produces SVG files, which allow you to drill down to see the full function names, which get clipped in the images.

I first run Masscan using the standard `libpcap` API, which sends packets via the kernel, the normal way. Doing it this way gets a packet rate of about 1.5 million packets-per-second, as shown in Figure 5.

To the left, you can see how `perf` is confused by the call stack, with [unknown] functions. Analyzing this part of the data shows the same call stacks that appear in the central section. Therefore, assume all that time is simply added onto similar functions in that area, on top of `__libc_send()`.

The large stack of functions to the right is `perf` profiling itself.

In the section to the right where Masscan is running, you'll notice little towers on top of each function call. Those are the interrupt handlers in the kernel. They technically aren't part of Masscan, but whenever an interrupt happens, registers are pushed onto the stack of whichever thread is currently running. Thus, with high enough resolution (faster samples, longer profile duration), `perf` will count every function as having spent time in an interrupt handler.

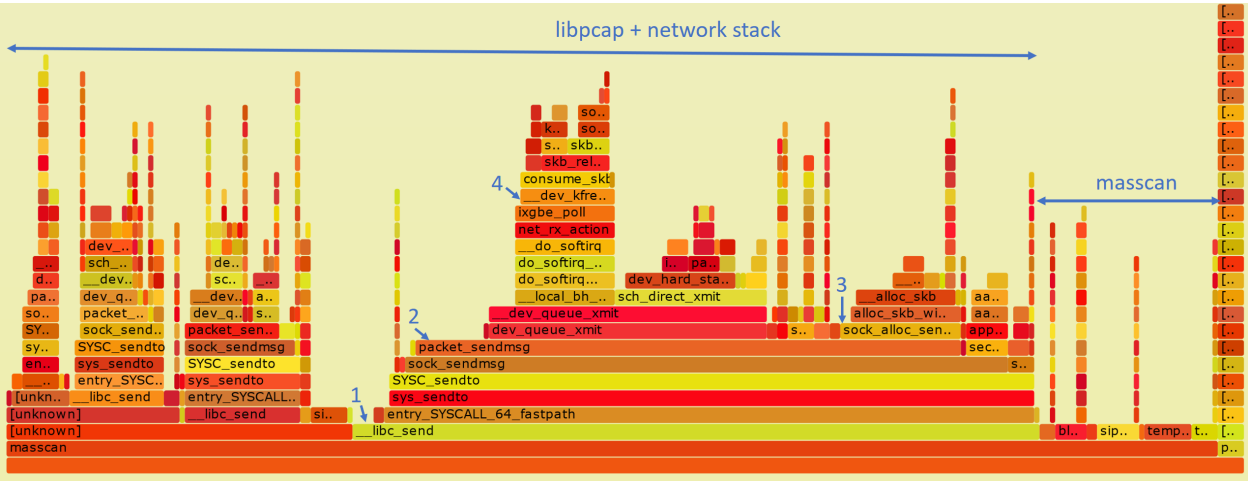
The next run of Masscan bypasses the kernel completely, replacing the kernel's Ethernet driver with the user-mode driver `PF_RING`. It uses the same options, but adds "zc:" in front of the adapter name. It transmits at 8 million packets-per-second, using an Ivy Bridge processor running at 3.2 GHz (turboed up from 2.5 GHz). Shown in Figure 6, this results in just 400 cycles per packet!

The first thing to notice here is that 3.2 GHz divided by 8 mpps equals 400 clock cycles per packet. If we looked at the raw data, we could tell how many clock cycles each function is taking.

Masscan sits in a tight scanner loop called `transmit_thread()`. This should really be below all the rest of the functions in this flame graph, but apparently `perf` has trouble seeing the full call stack.

The scanner loop does the following calculations:

- It randomizes the address in `blackrock_shuffle()`
- It calculates a SYN cookie using the `siphash24()` hashing function



1 marks the start of `entry_SYSCALL_64_fastpath()`, where the machine transitions from user to kernel mode. Everything above this is kernel space. That's why we use `perf` rather than user-mode profilers like `gprof`, so that we can see the time taken in the kernel.

2 marks the function `packet_sendmsg()`, which does all the work of sending the packet.

3 marks `sock_alloc_send_pskb()`, which allocates a buffer for holding the packet that's being sent. (`skb` refers to `sk_buff`, the socket buffer that Linux uses everywhere in the network stack.)

4 marks the matching function `consume_skb()`, which releases and frees the `sk_buff`. I point this out to show how much of the time spent transmitting packets is actually spent just allocating and freeing buffers. This will be important later on.

Figure 5. Performance profile of Masscan with `libpcap`.

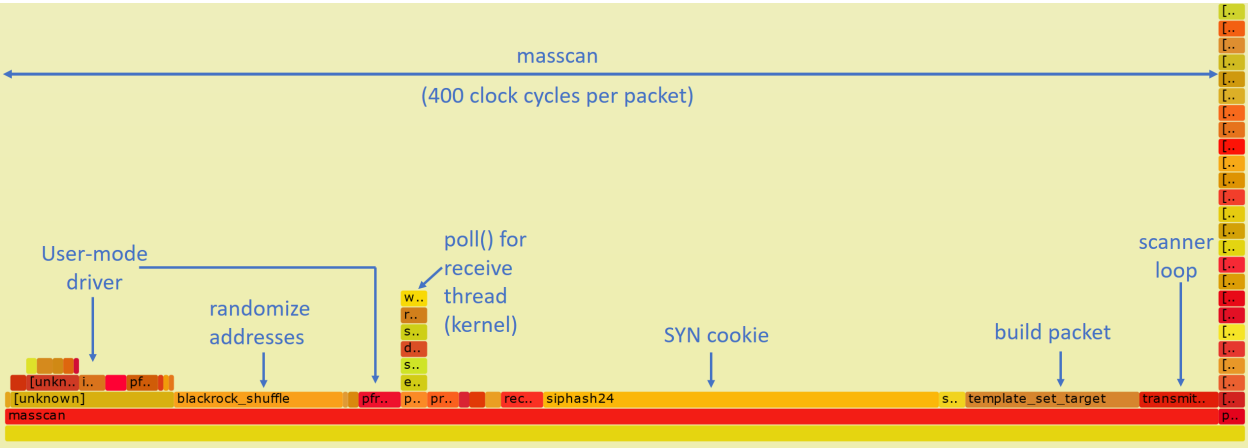


Figure 6. Performance profile of Masscan with `PF_RING`.

- It builds the packet, filling in the destination IP/port, and calculating the checksum
- It then transmits it via the `PF_RING` user-mode driver

At the same time, the `receive_thread()` is receiving packets. While the transmit thread doesn't enter the kernel, the receive thread will, spending most of its time waiting for incoming packets via the poll() system call. Masscan transmits at high rates, but receives responses at fairly low rates.

To the left, in two separate chunks, we see the time spent in the `PF_RING` user-mode driver. Here `perf` is confused: about 1/3 of this time is spent in the receive thread, and the other 2/3 in the transmit thread.

About ten to fifteen percent of the time is taken up inside `PF_RING` user-mode driver or an overhead 40 clock cycles per packet.

Nearly half of the time is taken up by `siphash24()`, for calculating the SYN cookie. Masscan doesn't remember which packets it's sent, but instead uses the SYN cookie technique to verify whether a response is valid. This is done by setting the Initial Sequence Number of the SYN packet to a hash of the IP addresses, port numbers, and a secret. By using a cryptographically strong hash, like `siphash`, it assures that somebody receiving packets cannot figure out that secret and spoof responses back to Masscan. Siphash is normally considered a fast hash, and the fact that it's taking so much time demonstrates how little the rest of the code is doing.

The *build packet* takes ten percent of the time. Most of the this is spent needlessly calculating the checksum. This can be offloaded onto the hardware, saving a bit of time.

The most important point here is demonstrating that the transmit thread doesn't hit the kernel. The receive thread does, because it needs to stop and wait, but the transmit thread doesn't. `PF_RING`'s custom user-mode driver simply reads and writes directly into the network hardware registers, and manages the transmit and receive ring buffers, all memory-mapped from kernel into user mode.

The benefits of this approach are that there is no system call overhead, and there is no needless copying of packets. But the biggest performance gain comes from not allocating and then freeing packets. As we see from the previous profile, that's where the kernel spends much of its time.

The reason for this is that the network card is normally a shared resource. While Masscan is transmitting, the system may also be running a webserver on that card, and supporting SSH login sessions. Sharing these resources ultimately means allocating and freeing `sk_buff`s whenever packets are sent or received.

`PF_RING`, however, wrests control of the network card away from the kernel, and gives it wholly to Masscan. No other application can use the network card while Masscan is running. If you want to SSH into the box in order to run m̂asscan, you'll need a second network card.

If Masscan takes 400 clock cycles per packet, how many CPU instructions is that? Perf can answer that question, with a call like `perf -a sleep 100`. It gives us an IPC (instructions per clock cycle) ration of 2.43, which means around 1000 instructions per packet for Masscan.

To reiterate, the point of all this profiling is this: when running with `libpcap`, most of the time is spent in the kernel. With `PF_RING`, we can see from the profile graphs that the kernel is completely bypassed on the transmit thread. The overhead goes from most of the CPU to very little of the CPU. Any performance issues are in the Masscan, such as choosing a slow cryptographic hash algorithm instead of a faster, non-cryptographic algorithm, rather than in the kernel!

How to Replicate This Profiling

Here is brief guide to reproducing this article's profile flamegraphs. This would be useful to compare against other network projects, other drivers, or for playing with Masscan to tune its speed. You may skip to the next section on a first reading, but if, like me, you never trusted a graph you could not reproduce yourself, read on!

Get two computers. You want one to transmit, and another to receive. Almost any Intel desktop will do.

Buy two Intel 10gig Ethernet adapters: one to transmit, and the other to receive and verify the packets have been received. The adapters cost $200 to $300 each. They have to be the Intel chipset, other chipsets won't work.

Install Ubuntu 16.04, as it's the easiest system to get `perf` running on. I had trouble with other systems.

The `perf` program gets confused by idle threads. Therefore, for profiling, I rebooted the Linux computer with `maxcpus=1` on the boot command

line. I did this by editing `/etc/default/grub`, adding `maxcpus=1` to the line `GRUB_CMDLINE_LINUX_DEFAULT`, then running `update-grub` to save the configuration.

To install `perf`, Masscan, and FlameGraph.

```
apt-get install linux-tools-common \
    linux-tools-`uname -r` git \
    build-essential libpcap-dev

git clone https://github.com/brendangregg/
    FlameGraph
# Get masscan from source and build it:
git clone https://github.com/
    robertdavidgraham/masscan
cd masscan
make
make test
ln bin/masscan /usr/local/sbin/masscan
cd ..
# Get PF_RING from source and build it:
git clone https://github.com/ntop/PF_RING
cd PF_RING
make
cd kernel
make install
insmod pf_ring.ko
cd ../userland/tools
make install
cd ../drivers/intel/ixgbe/ixgbe-5.0/src
make
sh load_drivers.sh
cd ../../../../../..
```

The `pf_ring.ko` module should load automatically on reboot, but you'll need to rerun `load_drivers.sh` every time. If I ran this in production, rather than just for testing, I'd probably figure out the best way to auto-load it.

You can set all the parameters for Masscan on the command line, but it's easier to create a default configuration file in `/etc/masscan/masscan.conf`:

```
source-ip = 00:11:22:33:44:55
adapter-mac = 00:22:22:22:22:22
router-mac = 00:11:22:33:44:55
include = 0.0.0.0-255.255.255.255
exclude = 255.255.255.255
port = 0-65535
```

Since there is no network stack attached to the network adapter, we have to fake one of our own. Therefore, we have to configure that source IP and MAC address, as well as the destination router MAC address. It's really important that you have a fake router MAC address, in case you accidentally cross-connect your 10gig hub with your home network and end up blasting your Internet connection. (This has happened to me, and it's no fun.)

Now we run Masscan. For the first run, we'll do the normal adapter without `PF_RING`. Pick the correct network adapter for your machine (on my machine, it's `enp2s03`.)

```
masscan -e enp2s0f1 --rate 100000000
```

In another window, run the following. This will grab 99 samples per second for 60 seconds while Masscan is running.

```
cd FlameGraph
perf record -F 99 -a -g -- sleep 60
perf script | ./stackcollapse-perf.pl > out.
    perf-folded
./flamegraph.pl out.perf-folded > masscan-
    pcap.svg
```

You'll have to wait 60 seconds, then it'll produce the file `masscan-pcap.svg` with the FlameGraph pictures.

Now, repeat the process to produce `masscan-pfring.svg` with the following command. It's the same as the original Masscan run, except that we've prefixed the adapter name with `zc:`. This disconnects any kernel network stack you might have on the adapter and instead uses the user-mode driver in the `libpfring.so` library that Masscan will load:

```
masscan -e zc:enp2s0f1 --rate 100000000
```

At this point, you should have two FlameGraphs. Load these in any web browser, and you can drill down into the specific functions.

Playing with `perf` options, or using something else like `dtrace`, might produce better results. The results I get match my expectations, so I haven't played with them enough to test their accuracy. I challenge you to do this, though—for reproducibility is the heart and soul of science. Trust no one; reproduce everything you can.

Now back to our regular programming.

How Ethernet Drivers Work

If you run `lspci -v` for the Ethernet cards, you'll see something like the following.

```
02:00.1 Ethernet controller: Intel Corporation 82599 10
      Gigabit TN Network Connection (rev 01)
        Subsystem: Intel Corporation 82599 10 Gigabit
      TN Network connection
        Flags: bus master, fast devsel, latency 0, IRQ
      17
        Memory at df200000 (64-bit, non-prefetchable) [
      size=2M]
        I/O ports at e000 [size=32]
        Memory at df600000 (64-bit, non-prefetchable) [
      size=16K]
        Capabilities: <access denied>
        Kernel driver in use: ixgbe
        Kernel modules: ixgbe
```

There are five parts to notice.

- There is a small 16k memory region. This is where the driver controls the card, using memory-mapped I/O, by reading and writing these memory addresses. There's no actual memory here—these are registers on the card. Writes to these registers cause the card to do something, reads from this memory check status information.

- There is a small amount of I/O ports address space reserved. It points to the same registers mapped in memory. Only Intel x86 processors support a second I/O space along with memory space, using the `inb`/`outb` instructions to read and write in this space. Other CPUs (like ARM) don't, so most devices also support memory-mapped I/O to these same registers. For user-mode drivers, we use memory-mapped I/O instead of x86's "native" `inb`/`outb` I/O instructions.

- There is a large 2-megabyte memory region. This memory is used to store descriptors (pointers) to packet buffers in main memory. The driver allocates memory, then writes (via memory-mapped I/O) the descriptors to this region.

- The network chip uses Bus Master DMA. When packets arrive, the network chip chooses the next free descriptor and DMAs the packet across the PCIe bus into that memory, then marks the status of the descriptor as used.

- The network chip can (optionally) use interrupts (IRQs) to inform the driver that packets have arrived, or that transmits are complete. Interrupt handlers must be in kernel space, but the Linux user-mode I/O (UIO) framework allows you to connect interrupts to file handles, so that the user-mode code can call the normal poll() or `select()` to wait on them. In Masscan, the receive thread uses this, but the interrupts aren't used on the transmit thread.

There is also some confusion about IOMMU. It doesn't control the memory mapped I/O—that goes through the normal MMU, because it's still the CPU that's reading and writing memory. Instead, the IOMMU controls the DMA transfers, when a PCIe device is reading or writing memory.

Packet buffers/descriptors are arranged in a ring buffer. When a packet arrives, the hardware picks the next free descriptor at the head of the ring, then moves the head forward. If the head goes past the end of the array of descriptors, it wraps around at the beginning. The software processes packets at the tail of the ring, likewise moving the tail forward for each packet it frees. If the head catches up with the tail, and there are no free descriptors left, then the network card must drop the packet. If the tail catches up with the head, then the software is done processing all the packets, and must either wait for the next interrupt, or if interrupts are disabled, must keep polling to see if any new packets have arrived.

Transmits work the same way. The software writes descriptors at the head, pointing to packets it wants to send, moving the head forward. The hardware grabs the packets at the tail, transmits them, then moves the tail forward. It then generates an interrupt to notify the software that it can free the packet, or, if interrupts are disabled, the software will have to poll for this information.

In Linux, when a packet arrives, it's removed from the ring buffer. Some drivers allocate an `sk_buff`, then copy the packet from the ring buffer into the `sk_buff`. Other drivers allocate an `sk_buff`, and swap it with the previous `sk_buff` that holds the packet.

Either way, the `sk_buff` holding the packet is now forwarded up through the network stack, until the user-mode app does a `recv()`/`read()` of the data from the socket. At this point, the `sk_buff` is freed.

A user-mode driver, however, just leaves the packet in place, and handles it right there. An IDS, for example, will run all of its deep-packet-inspection right on the packet in the ring buffer.

Logically, a user-mode driver consists of two steps. The first is to grab the pointer to the next available packet in the ring buffer. Then it processes the packet, in place. The next step is to release the

packet. (Memory-mapped I/O to the network card to move the tail pointer forward.)

In practice, when you look at APIs like `PF_RING`, it's done in a single step. The code grabs a pointer to the next available packet while simultaneously releasing the previous packet. Thus, the code sits in a tight loop calling `pfring_recv()` without worrying about the details. The `pfring_recv()` function returns the pointer to the packet in the ring buffer, the length, and the timestamp.

In theory, there's not a lot of instructions involved in `pfring_recv()`. Ring buffers are very efficient, not even requiring locks, which would be expensive across the PCIe bus. However, I/O has weak memory consistency. This means that although the code writes first A then B, sometimes the CPU may reorder the writes across the PCI bus to write first B then A. This can confuse the network hardware, which expects first A then B. To fix this, the driver needs memory fences to enforce the order. Such a fence can cost 30 clock cycles.

Let's talk `sk_buffs` for the moment. Historically, as a packet passed from layer to layer through the TCP/IP stack, a copy would be made of the packet. The newer designs have focused on "zero-copy," where instead a pointer to the `sk_buff` is forwarded to each layer. For drivers that allocate an `sk_buff` to begin with, the kernel will never make a copy of the packet. It'll allocate a new `sk_buff` and swap pointers, rewriting the descriptor to point to the newly allocated buffer. It'll then pass the received packet's `sk_buff` pointer up through the network stack.

As we saw in the FlameGraphs, allocating `sk_buff`s is expensive!

Allocating `sk_buff`s (or copying packets) is necessary in the Linux stack because the network card is a shared resource. If you left the packets in the ring buffer, then one slow app that leaves the packet there would eventually cause the ring buffer to fill up and halt, affecting all the other applications on the system. Thus, when the network card is shared, packets need to be removed from the ring. When the network card is a dedicated resource, packets can just stay in the ring buffer, and be processed in place.

Let's talk zero-copy for a moment. The Linux kernel went through a period where it obsessively removed all copying of packets, but there's still one copy left: the point where the user-mode application calls `recv()` or `read()` to read the packet's contents. At that point, a copy is made from kernel-mode memory into user-mode memory. So the term zero-copy is, in fact, a lie whenever the kernel is involved!

With user-mode drivers, however, zero-copy is the truth. The code processes the packet right in the ring buffer. In an application like a firewall, the adapter would DMA the packet in on receive, then out on transmit. The CPU would read from memory the packet headers to analyze them, but never read the payload. The payload will pass through the system completely untouched by the CPU.

Let's talk about interrupts for a moment. Back in the day, an interrupt was generated per packet. Indeed, at one time, two interrupts could be generated, one after the TCP/IP headers were received, so processing could start immediately, and another after the rest of the packet had been received.

The value of interrupts is that they provide low latency, important for devices that forward packets (firewalls, IPS, routers), or for fast responses to packets. The cost of interrupts, though, is that they cause large CPU overhead. When an interrupts happens, it forces execution of an interrupt handler. Even medium rates of packets can overwhelm the system with interrupts, so that as soon as the system leaves an interrupt handler, it immediately enters another one. In such cases, the system has essentially locked up. The mouse won't even move on the screen until the packet rate decreases, after which point the system will behave normally.[32]

The obvious solution to this is to turn off interrupts from the network card. Instead, the software can sit in a tight loop and `poll()` to see if new packets arrive. Another strategy is to program the timer chip for frequent interrupts. The card can bounce back and forth among these strategies, depending on the current network speed. Polling consumes a lot of CPU time. Using delayed timer interrupts increases latency.

Those writing custom drivers have used these strategies since the 1980s. Around 2006, Linux drivers started doing the same, using the NAPI API to enable polling when packets arrived at high speed. Around that time, network hardware also improved, adding support for coalescing interrupts, so that it generated fewer at high speed, generating only one interrupt after many packets have arrived.

In the graphs, you saw that the `libpcap` had

[32]If caught during the late stages of booting, the system might not even boot up until the packet flow eases up.

some small overhead with interrupts, but it's not overwhelming, because NAPI interrupt moderation kicks in. Using `pfring` gets rid of this overhead.

Let's talk system call overhead. A recent paper by Livio Soares and Michael Stumm does a good job measuring it.[33] The basic cost of entering or leaving kernel space is around 150 clock cycles. This alone takes more time than all the user-mode driver processing done by `PF_RING`, according to our measurements.

There are further expenses to the system call. It has to walk through a bunch of kernel data structures. This then pollutes the caches on the chip. According to the Soares paper, it evicts about half the data in the L1 cache. This will cause data access to go from 4 clock cycles (often masked by the out-of-order processing of the CPU) to 12 clocks in L2 cache, or 30 clocks in L3 cache. The effective cost can thus equal hundreds of extra clock cycles.

On the other hand, the cost can easily be amortized by doing multiple packet reads or writes per system call. Linux has a `recvmsg()` system call that does this, to good effect.

Combining all this together, we see why a user-mode driver has such big gains (or conversely, why the kernel has such big losses): (a) it avoids the allocation/deallocation of memory; (b) it avoids any memory copies; (c) it avoids system call overhead, and (d) it avoids interrupts.

Some History of Ethernet Drivers

Since the dawn of networking there have been people dissatisfied with the standard Ethernet drivers who have written their own.

An example were packet sniffers, like the Network General "Sniffer" product. Back in the day, they wrote custom drivers so they could capture at "wire speed" on an 80286 microprocessor. The major feature was simply disabling interrupts. Portable MS-DOS computers were used as packet sniffers because "real" computers like SPARCstations running Solaris couldn't handle high traffic rates.

Early drivers were hard, because hardware sucked. There was no bus master DMA in the early ISA bus days, so for DMA, you had to use the motherboard's DMA controller. Only, it wasn't really that fast. So instead, drivers used the Programmed I/O (PIO) mode to read packets from the adapter.

There was also the problem of bus bandwidth. Early PCI supported 1 Gbps in theory (32 bits times 33 MHz), but various overheads made that impractical. It wasn't until wider PCI (64-bit) or/and faster PCI (66 MHz) that true wirespeed gigabit Ethernet was possible.

Also, with PCI, all the slots were shared on the same bus, so other devices impacted yours. This was especially difficult when building firewalls, routers, or IPS applications that needed to both transmit and receive. Luckily, motherboards started supporting multiple independent PCI buses. Still, PCI was still single-plexed, meaning it couldn't transfer in both directions at the same time.

Virtually all these concerns have gone away now. Even a single lane of PCIe 1.0 is 2 Gbps, bidirectional, with more than enough bandwidth to handle sending and receiving at full 1 Gbps.

The early Intel 1 Gbps card had only 256 descriptors. Timing was tight enough that at full bandwidth; there wasn't enough time to process packets before the ring buffer would fill up. With BlackICE, we solved this by allocating an effective ring buffer of several thousand descriptors. Then, when packets arrived, we replaced the existing descriptors with new descriptors from the preallocated set. We used two CPUs, one dedicated to running the user-mode driver doing this, and another reading and processing packets from the large virtual ring buffer. I mention this trick because, at the time, Intel engineers told us it wasn't possible to capture packets at wirespeed, and we were able to prove them wrong.

Historically, and often today, the reality is that few hardware vendors test their hardware at maximum speed. Since operating systems can't handle it, they don't test for it. That makes writing drivers for practical hardware much harder than it would seem in theory, as driver writers have to overcome bugs in the hardware.

Today, custom drivers are common. Back in the day, they were black magic.

Core Concept

In 1998, I created BlackICE, an IDS/IPS using a custom driver. A frequent question at the time was why we didn't write it on Linux, or even BSD, which everyone knew was faster. In particular, some papers at the time "proved" that the BSD networking was the fastest.

[33] `unzip pocorgtfo15.pdf flexsc-osdi10.pdf`

This bothered me because I was unable to explain the core concept. If we are completely bypassing the operating system, then the operating system doesn't matter. As the graphs show, Masscan spends no time in the operating system. Given the same version of GCC, and the same hardware, it'll run at nearly identical speed, regardless if the operating system is Windows, Linux, or BSD. It's like any other CPU-bound (rather than OS-bound) task.

Yet, people couldn't appreciate this. They knew in their hearts that some operating system was better, and couldn't see the concept of bypassing it.

BlackICE used poll mode, instead of interrupts, so it didn't lock up under high packet rates. Now, with NAPI, and poll-mode drivers like `PF_RING`, it's something everyone can play with and understand. Back then, it was some weird black magic that people refused to believe actually worked. My 11-inch laptop computer happened to use 3Com's 3c905 chip, the only 100 Mbps card we wrote a driver for. Even after demonstrating it handling the maximum rate of 148,800 packets-per-second, people refused to believe it worked. There's a Defcon video where the presenter claims that this is impossible, that the notebook would literally melt under such a load. Nowadays, cheap notebooks easily handle max 1 Gbps speeds (1,488,000 packets-per-second) using things like `PF_RING`.

In 2003, Gartner came out with a report that software IDS was dead, because it couldn't handle line-rate gigabit Ethernet, and that "hardware" was needed. That was based on experience with Snort, which had no custom drivers available at the time. Even when customers explained to Gartner they were successfully using our product at line rate, they refused to believe.

More interesting was the customers who tested our software product side-by-side with "hardware" competitors in the lab, and found our product faster. They still bought the competitors', because of FUD. Nobody got fired for buying a hardware product that turned out to be slow.

Even today, discussions of these drivers still get questions like "What about Endace?" Endace builds custom cards with FPGAs to accelerate processing. This doesn't apply. The overhead for Masscan using `PF_RING` is nearly zero, and would have the identical overhead working with an Endace card, also near zero. The FPGA doesn't reach outside the card and somehow make Masscan's code faster.

Yes, Endace does have some advantages. You can push filters to card, so that fewer packets arrive in a system. This is needed in some networks. However, most people use Endace for things that `PF_RING` would solve just fine, because they believe in the power of hardware.

Finally, the same sorts of prejudices exist with kernel code. Programmers are indoctrinated to believe code runs faster in the kernel, which is not true. The reason you push stuff into the kernel is to avoid the kernel/user transition. There's otherwise no inherent advantage. Pushing things like the driver to user mode is just doing the same thing, avoiding the kernel/user transition. Indeed, that's all micorokernels are, operating systems that aggressively push subsystems outside the kernel.

Several Drivers to Choose From

Masscan uses `PF_RING` because of compile dependencies—there is no actual dependency. You compile Masscan without any dependency on `PF_RING`, yet that compiled code will go hunt for the `pfring.so` library and dynamically load it. Thus, in the replication instructions, I have you compile Masscan first, and `PF_RING` second.

But there are two other options of note.

Intel has a system called DPDK, the Data-Plane Development kit. It contains not only a user-mode driver similar to `PF_RING`, but a whole toolkit to solve other problems, like multi-CPU synchronization and multi-socket NUMA memory handling. It's a real awesome toolkit. However, it's also an enormous dependency for code. That's why Masscan uses `PF_RING`—it's an optional feature that most users will never see. Had I used DPDK, I would've forced users into dependency hell trying to build a massive toolkit for my little application.

Another option is `netmap`. This is a kernel-mode driver that is otherwise identical to the user-mode stuff. It memory maps the packet buffers in user space, so it's truly zero copy. It also disconnects the driver from the network stack, and gives exclusive access to the application, so there's no allocation and freeing of `sk_buff`s. It batches multiple reads and writes with a single system call, amortizing the cost of system calls across many packets.

The great thing about `netmap` is that it's built into the latest Linux kernels. Assuming you have Intel Ethernet, or even a Realtek Gigabit card, it should work immediately with no special software. I haven't gotten around to adding this to Masscan, but the overhead should be comparable to `PF_RING`—despite being tainted with evil kernel-mode code.

Some notes on IDS design

One place to use these "user-mode no-interrupt zero-copy ring-buffer" drivers is with a network intrusion detection system, or even an inline version called and intrusion prevention system.

None of the existing open-source IDS projects (Snort, Bro, Suricata) are really designed for speed. They were written using `libpcap` where, at high speed, the kernel consumed most of the CPU power. As a consequence, there were only so much performance improvements that could be made before it wasn't worth it. Optimizations that made the software infinitely fast would still not even double the practical performance of the IDS, because the kernel would be eating up all the time.

But, with near zero overhead in the drivers, some interesting optimizations become worthwhile.

One problem with the Snort IDS is how it does TCP reassembly. It must copy packets into the same buffer in order to perform regex searches. This adds two things which we know to be bad: memory allocations and memory copies.

An alternative is to not do this, to neither do regex as the basis of signatures, nor do reassembly.

This approach is demonstrated in Masscan in several places. Masscan can establish a TCP connection and interact with the service. When it needs to search for patterns, instead of a regex it uses an Aho-Corasick (AC) pattern matcher. Whereas a normal regex needs to have a complete buffer, so that it can do back tracking, an AC pattern matcher does not. It accepts input a sequence of fragments, saving the state of the search at the end of one fragment and continuing at the start of the next fragment.

This has the same practical ability to search a TCP stream, but without the need to "reassemble" fragments, allocate memory, or do memory copies.

In abstract computer science terms, this is the tradeoff between NFAs (non-deterministic finite automata) which can consume a lot of CPU power, and DFAs (deterministic finite automata), which consume a fixed amount of CPU power, but at the expense of using a lot of memory for the tables it builds.

Another thing you'll see in Masscan is protocol decoders based on state machines. Again, instead of reassembling packets, the protocol decoder saves state at the end of one fragment and continues with that state at the start of the next. An example of this is the X.509 parser, `proto-x509.c`. The unit test calls this two ways, one with an entire certificate to be parsed, and one where the bytes are processed one at a time, as if they had arrived in fragments over TCP.

Such state-machine parsers are really weird, but by avoiding memory allocations and copies, they become really fast at high network speeds. It's a difficult optimization to make the code that would add little value when using kernel mode drivers, but becomes an important way of building an IDS if using these zero-overhead drivers.

— — — — — — — — — —

The kernel is a lie.

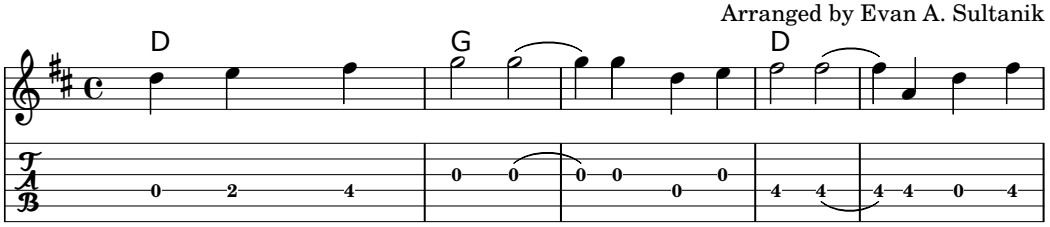
This Net Is Your Net

Based on the song "This Land is Your Land" by Woody Guthrie

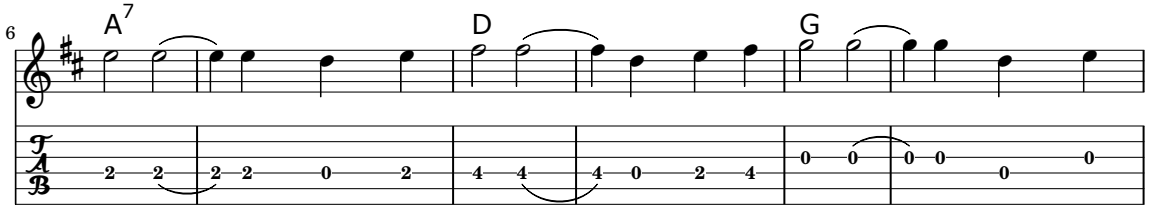
A Bad BIOS analog production for acoustic guitar, violin, and piano

Music by Don A. Bailey, Lyrics by Don A. Bailey and Alex Kreilein

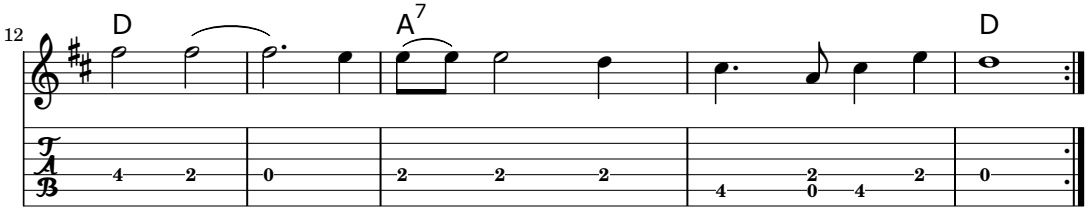
Arranged by Evan A. Sultanik



This Net is your Net, this Net is my Net from Wi - ki -
As I im - mersed in that digi-tal high-way all a -
While under white walled mon - uments, old men ban - ter some of them
Was a Fire - wall there, that tried to stop me a sign was
No - bo - dy liv - in' can ev - er stop me as I go



pe - dia to Shen - zhen Mar - kets from Reddit's four - chan to Twit - ter's
round me, e - lec - trons lit my way and un - der neath me, green plas - tic
plot - ted, how we don't deserve ans - wers the reg - u - la - tor, who swore to
flash - ing: Net - work Se - cur - ity! But on the back end, it didn't say
hack - in' on free - dom's high-way no - bo - dy liv - ing can ever make me



foll - owers the Inter - net was made for you and me
path - ways these cir - cuits were made for you and me
protect her now he works against freedoms for you and me
noth - in' in - forma - tion was made to be set free
turn back the Inter - net was made for you and me

15:09 Detecting Emulation with MIPS16 Delay Slots

by Ryan Speers and Travis Goodspeed
with the kindest of thanks to Thorsten Haas.

Howdy y'all,

Let's begin with a joke that I once heard at a conference: *David Patterson and John Hennessy walk into a bar. Everyone gathers to listen to the two heroes who built legendary machines. The entire bar spends the night multiplying fractions, and then everyone has that terrible hangover you get when you realize you had no fun and learned nothing new, even though your night started out so promising.*

But let's tell the joke differently: *Patterson and Hennessy walk into a bar in another town, but this time, Greg Peterson is behind the bar. The two of them begin a long-winded story about weighted averages, lashing out at "RISC-deniers" who aren't even in the room. Just as folks begin to get bored, and begin to sip their drinks too quickly out of nervousness, Peterson jumps in and saves the day. Because he knows that these fine folks build real machines that really shipped, he redirects the conversation to war stories and practical considerations.*

Patterson tells how the two-stage pipeline in the RISC 1 chip was the first design with a branch delay slot, as there's no point in throwing away the staged instruction that has already finished execution. Hennessy jumps in with a tale of dual instruction sets on MIPS, allowing denser code without abandoning the spirit of the RISC faith. Then Peterson, the bartender, serves up a number of Xilinx devkits to bar patrons, who begin collaborating on a five-stage pipeline design of their own, with advice on specific design choices from David and John. The next morning, they've built a working CPU and suffered no hangovers.

If your Computer Architecture class was more like the former than the latter, I hope that this brief article will show you some of the joy of this fine subject.

In PoC||GTFO 6:6, Craig Heffner discussed a variety of methods for detecting Qemu emulation of MIPS hardware. We'll be discussing one more way to detect emulation, but we'll be using the MIPS16 instruction set and a clever trick of delay slots to detect the emulation.

We wanted to craft a capability that is (a) able to differentiate hardware from an emulation environment, and also (b) able to confuse static analysis. We picked used standard tools: Qemu as an emulation environment and IDA Pro as a disassembler.[34]

The first criterion leads us to want something that both: (a) works in userland, and (b) is not trivial for an emulator developer to patch. Moving to userland meant that hardware registry inspection, as discussed in Section 6.1 of Heffner's article, would not work. Similarly, the technique of reading `cpuinfo` in Section 6.2 would be easily patchable, as Craig noted. Here, we instead seek a capability more similar to Section 6.3, where cache incoherency is exploited to differentiate real hardware and Qemu.

MIPS16e

SSH'ing to a newly acquired MIPS box, we find the same nifty line of `cpuinfo` that struck our fancy in Craig's article. MIPS16 is an extension to the classic MIPS instruction set that fills the same niche as Thumb2 does on ARM. The instructions word is 16 bits wide, a subset of the full register set is directly available, and a core tenet of RISC is violated: some instructions are more than one word long.

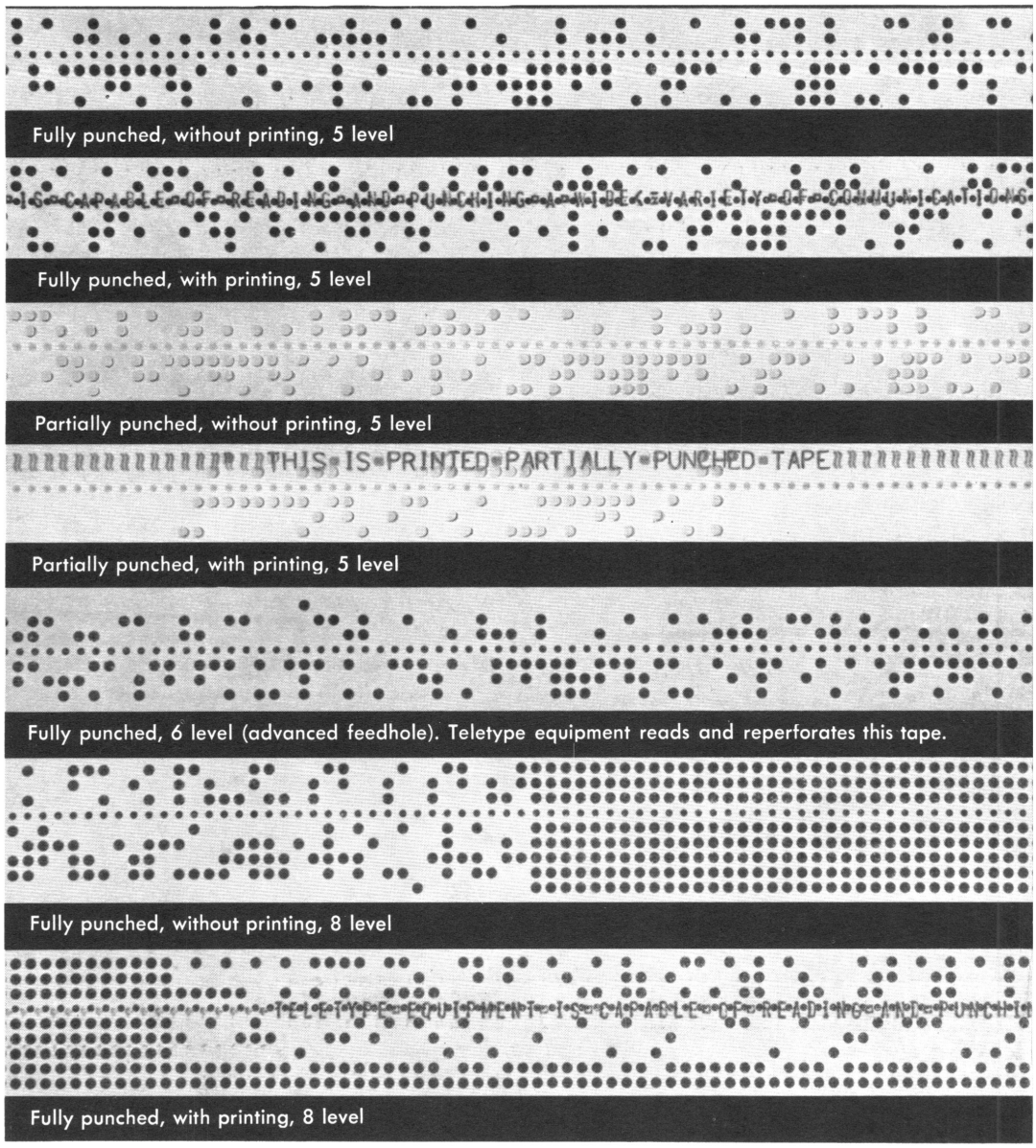
```
$ cat /proc/cpuinfo
system type             : BCM7358A1 STB platform
cpu model               : Broadcom BMIPS3300 V3.2
cpu MHz                 : 751.534
tlb_entries             : 32
isa                     : mips1 mips2 mips32r1
ASEs implemented        : mips16
```

Just like ARM, this alternate instruction set is used whenever the least significant bit of the program counter is set. Function pointers work as expected between the two instruction sets, and the calling conventions are compatible.

[34] We will happily buy the drinks in celebration of Radare2 issue 1917 and Capstone issue 241 being closed.

TELETYPE COMMUNICATIONS TAPES

Teletype equipment is capable of punching and reading a wide variety of communications tapes. Our equipment can produce tapes with or without printing, partially or fully punched, and in 5, 6, 7 or 8 level codes. More information on how Teletype equipment processes paper tapes is available from our engineers experienced in its use. Call, write or wire today! Here are a few examples of Teletype tapes shown actual size:



Teletype Corporation manufactures equipment for the Bell System and others who require the utmost reliability from their message and data communications systems.

Litho in U.S.A.
TCT10M10263

GENERAL OFFICES
5555 Touhy Avenue, Skokie, Ill.
Phones: ORchard 6-1000, Skokie
COrnelia 7-6700, Chicago
Direct Distance Dialing
Area Code 312
TWX: 312-677-6700
(24-hour unattended service)
W.U. Service on premises
Telex: 02-5451

GOVERNMENT LIAISON OFFICE
425—13th Street, N.W.
Washington 4, D.C.
Phone: MEtropolitan 8-1016

TELETYPE®
CORPORATION SUBSIDIARY OF *Western Electric Company* INC.

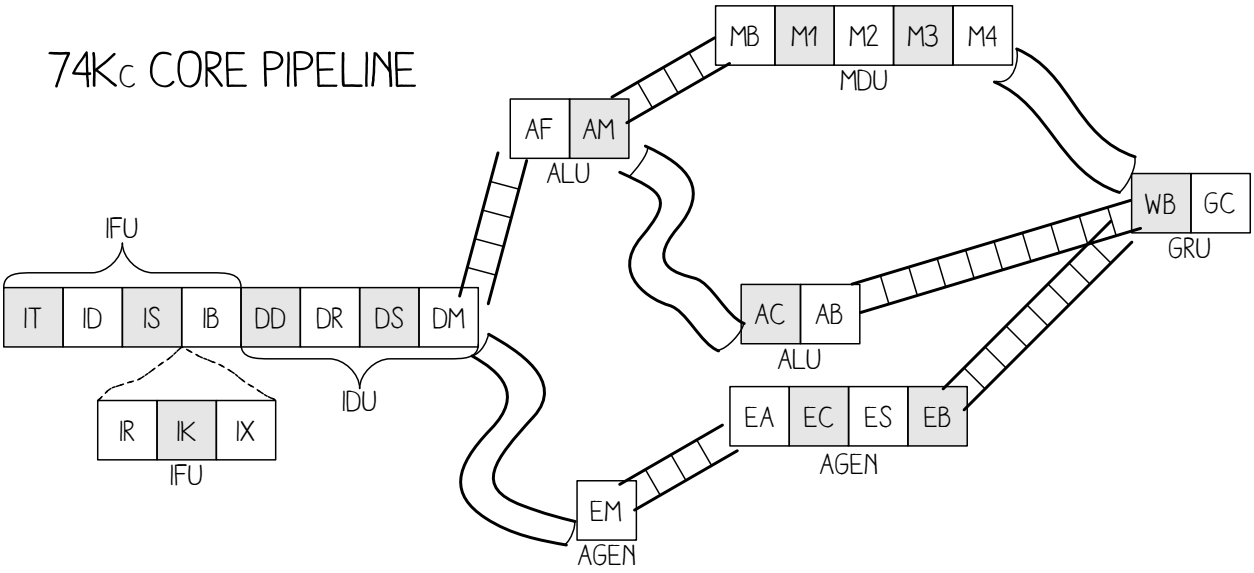


Figure 7. MIPS 74Kc Pipeline

Despite careful work to maintain compatibility between MIPS16 and MIPS32, there are inevitable differences. MIPS16 only has direct access to eight registers, rather than the 32 of its larger cousin.

CPU Pipelines

In Hennessy and Patterson's books, a five-stage pipeline is described and hammered into the poor reader's head. This classic RISC pipeline isn't what you'll find in modern chips, but it's a lot easier to keep in mind while working on them. The stages in order are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

Each pipeline stage can only hold one instruction at a time, but by passing the instructions through as a queue, multiple instructions can exist in *different stages* at the same time. When a branch is mis-predicted, the pipeline will be "flushed," which is to say that the partially-completed instructions from the incorrectly guessed branch are blown to the wind and replaced with harmless NOP instructions, which are sometimes called "bubbles."

Bubbles are also one way to avoid "data hazards," which are dependencies between instructions that run at the same time. For example, if you were to use a value just after loading it, the CPU would have to either insert a bubble to delay the second instruction until the value is ready or it would "forward" the register result.[35]

The MIPS 74Kc on one of our target machines has 14 or 15 pipeline stages, depending upon how you count, plus three additional stages for MIPS16e instruction decoding.[36] These stages are quite well documented, but to ease the explanation a bit, we won't bore you with the details of exactly what happens where. The stages themselves are shown in Figure 7, helpfully illustrated by Ange Albertini.

Extended (Wide) Instructions

We mentioned earlier that MIPS16 instructions are usually just one instruction word, but that sometimes they are two. That's a bit vague and handwavy, so we'd like to clear that up now with a concrete example.

There is an Extend Immediate instruction which allows us to enlarge the immediate field of another MIPS16 instruction, as its immediate field is smaller than that in the equivalent 32-bit MIPS instruction. This instruction is itself two bytes, and is placed directly before the instruction which it will extend, making the "extended instruction" a total of four bytes.

[35] Very early MIPS machines made the hazard the compiler's responsibility, in what was called the "load delay slot." It is separate from the "branch delay slot" that we'll discuss in a later section, and is no longer found in modern MIPS designs.

[36] `unzip pocorgtfo15.pdf mips74kc.pdf`

For example, the opcode for adding an immediate value of 1 to `r2` is `0x4a01`. (`r2` is the register for both the first argument to a function and its return value.) Because MIPS16 only encodes room for five immediate bits in this instruction, it allows for an extension word before the opcode to include extra bits. These can of course be zero, so `0xF000 0x4a01` also means `addi r2, 1`.

Some combinations are illegal. For example, extending the immediate bits of a NOP isn't quite meaningful, so trying to execute `0xF008 0x6500` (Extended Immediate NOP) will trigger a bus error and the process will crash.

The Extended Shift instruction shown along with a regular Shift in Figure 8. Now how the prefix word changes the meaning of the subsequent instruction word.

However, thinking of these two words as a single instruction isn't quite right, as we'll soon see.

Delay Slots

Unlike ARM and Thumb, but like MIPS32 and SPARC, MIPS16 has a branch delay slot. The way most folks think of this, and the way that it is first explained by Patterson and Hennessy,[37] is that the very next instruction after a branch is executed regardless of whether the branch is taken.

Sometimes this is hidden by an assembler, but a disassembler will usually show the instructions in their physical order. IDA Pro helpfully groups the delay-slot instruction into the proper block, so in graph view you won't mistake it for being conditionally executed.

Extended Instructions in a Delay Slot

So what happens if we put a multi-word instruction into the delay slot? IDA Pro, being first written for X86, assumes that X86 rules apply and the whole chunk is one instruction. Qemu agrees, and a quick test of the following code reveals that the full instruction is executed in the delay slot.

We can test this as we see that on both real hardware and Qemu, extending an instruction like a `NOP` that shouldn't be extended will trigger a bus error. However, when we put this combination after a return, it will only crash Qemu. In this case in hardware, only the extension word was fetched, which didn't cause an issue.

```
1 0xE820 //Return.
  0xF008 //Extension word.
3 0x6500 //NOP, will crash if extended.
```

This is a known issue with the MIPS16e instruction set.[38] To quote page 30, *"There is only one restriction on the location of extensible instructions: They may not be placed in jump delay slots. Doing so causes UNPREDICTABLE results."*

Making Something Useful

We can now crash an emulator while allowing hardware to execute, but let's improve this technique into something that can be used effectively for evasion. We'll replace the `NOP` which caused the crash when extended with an instruction which is intended to be extended, specifically an add immediate, `addi`.

```
1 0x6740 // First we zero r2, the
         // return value.
3 0xE820 // jr $ra   (Return)
  0xF000 // Extended immediate of 0.
5 0x4A01 // Add immediate 1 to r2.
         // (only executed in Qemu)
```

If we take that shellcode and view the IDA disassembly for it, you will see that, as above, IDA groups the delay-slot instruction into the function block so it looks like one is added to the return value. See Figure 9, being careful to remember that `$v0` means `r2`.

[37] Page 444 of Computer Organization and Design, 2nd ed.

[38] `unzip pocorgtfo15.pdf mips16e-isa.pdf`

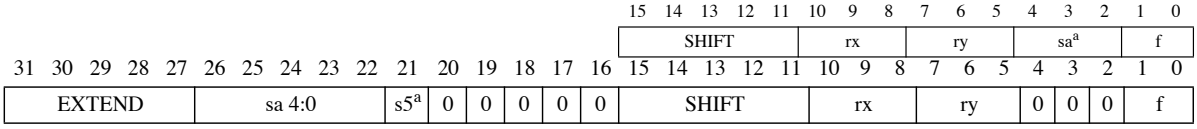


Figure 8. MIPS16 Regular and Extended Shift Instructions

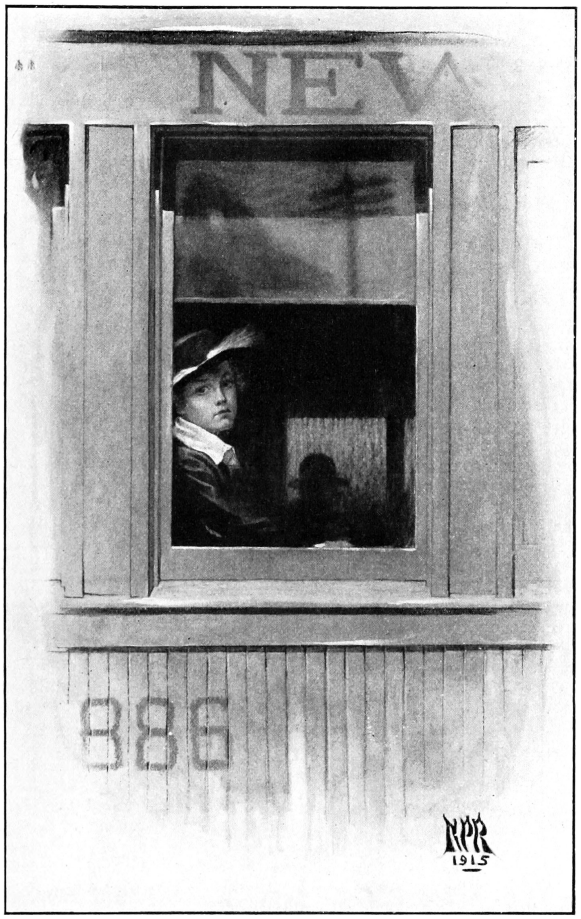
But hang on a minute, that delay slot holds two instruction words, and as we learned earlier, these can be thought of as separate instructions!

In fact, IDA only shows the instruction bytes on the left if you explicitly request a number of bytes from the assembly be shown. Without these being shown, a reverse engineer might forget that the program assembled a double-length instruction and thus that this behavior will occur.

This shows how we can confuse static analysis tools, which disassemble without taking into account this special case.

Let's now look at what happens when we take the above shellcode and execute it as a function from a program. We print the return value from the function in the below sample output.

```
int exec16(int (*fptr16)(int),
           int verbose){
  uint32_t res;
  uint8_t* bytes;
  int (*functionPtr)(int);
  functionPtr=(void*) (((int)fptr16)|1);
  return functionPtr(0xdeadbeef);
}

uint16_t amiemulated16[]={
0x6740, // First we zero r2, the
        // return value.
0xE820, // jr $ra    (Return)
0xF000, // Extended immediate of 0.
0x4A01  // Add immediate 1 to r2.
        // (only executed in Qemu)
};

int main() {
  printf("I am running %s.\n",
    exec16((void*) amiemulated16, 0)
    ? "in Qemu"
    : "on real hardware");
  return 0;
}
```



"ONE DOES NOT TRAVEL ELEVEN THOUSAND MILES WITHOUT ACQUIRING THE RIGHT TO BE TIRED."

We've discussed how IDA sees the extended addition as a single instruction, when in fact they are two separate MIPS instructions. But how is this handled in an emulator versus real MIPS hardware?

On the real hardware, when the return instruction is processed, the next instruction in the pipeline is `0xF000` (the extension instruction) and this is executed in the branch delay slot. That instruction, however, becomes a `NOP` in hardware.

```
ROM:0000 .set mips16
ROM:0000 # ======= SUBROUTINE =======
ROM:0000 amiemulated:
ROM:0000 67      40              move  $v0, $zero # Clear return value to zero.
ROM:0002 E8      20              jr    $ra        # Return
ROM:0004 F0      00 4A 01        addiu $v0, 1     # Adds 1 to return value in Qemu.
ROM:0004 # End of function amiemulated           # This becomes a NOP on real hardware.
```

Figure 9. MIPS16 Machine Code abusing the Delay Slot

```
~$ uname -a
Linux target 3.12.1 #1 mips GNU/Linux
~$ ./hello
I am running on real hardware.
```

The reason this detection works, we hypothesize, is because Qemu doesn't actually have a pipeline, and thus it is emulated by knowing that it should run the instruction following a branch, to "correctly" handle the branch-delay slot. When it reads that next instruction, it reads the two instructions that it sees as a single extended instruction, instead of just reading the extension.

```
~$ mips-linux-gnu-gcc -static -std=gnu99 \
    hello.c -o hello
~$ qemu-mips -L /usr/mips-linux-gnu hello
I am running in Qemu.
```

In hardware, we should note, the instruction isn't exactly tossed away because it's broken in half. The extension word, as the first half of the pair, never really gets executed on its own; rather, it hangs around in the pipeline to modify the subsequent instruction word. As the pipeline flows, the first word becomes a bubble as the second word becomes the single, unified instruction, but that unified instruction is too late to be executed. Instead, it is cruelly flushed from the MIPS16 pipeline while the bible ahead of it becomes a worthless NOP.

Thus, with just the eight byte function `0x6740 0xe820 0xf000 0x4a01`, we can reliably detect emulation of MIPS16. As an added bonus, IDA Pro will agree with the simulation behavior, rather than the hardware behavior.

– – – – – – – – – – –

Kind thanks are due to Thorsten Haas for lending us a MIPS shell account on impossibly short notice.

If you'd like to play around with more differences between hardware and emulation, we'll note that in MIPS32, `0x03E00008 0x03E00008` is a clean return to `$ra` on hardware, but crashes Qemu. To crash on hardware and return normally in Qemu, use `0x03e0f809 0x8fe20001`.

Cheers from Hanover, New Hampshire,
Travis and Ryan

15:10 Windows Kernel Race Condition Analysis While Accessing User-mode Data

by BSDaemon and NadavCh

In 2013, Google's researchers Mateusz Jurczyk (J00ru) and Gynvael Coldwind released a paper entitled "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns."[39] They discussed race conditions in the Windows kernel while accessing user-mode data and demonstrate how to find such conditions using an instrumented emulator. More importantly, they offered a very thorough explanation of how the identification of such issues is possible, specifically listing these conditions of interest:

1. At least two reads of the same virtual address;
2. Both read operations take place within a short time frame. The authors specifically recommend identifying reads in the handling of a single kernel entrance;
3. The reads must execute in kernel mode;
4. The virtual address subject to multiple reads must reside in memory writable by Ring-3 threads, in order for the user mode to be able to take advantage of the race.

Interestingly most of these races are exploitable—i.e., possible for the attacker to win—on modern machines given multiple CPU cores. The exceptions would be in memory areas that are administrator-owned, or in situations that are early boot—and thus not in a memory area that can be mapped by an attacker. Even if the user-mode area is only writable by administrator-owned tasks, it might still be a problem given that it leads to code execution in kernel mode that is prohibited to the administrator and bypasses kernel driver signing. Notably, the early boot cases are only non-exploitable if they are not part of services prohibited after boot.

We reproduced Google's research using Intel's SAE[40] and got some interesting results. This paper explains our approach in the hope of helping others understand the importance of documenting findings and processes. It also demonstrates other findings and clarifies the threat model for the Windows Kernel, thanks to our discussions with the MSRC. We share all the traces that generated double fetches for Windows 8 (pre and post booting) and Windows 10 (again, pre and post boot).[41]

We also share our implementation: it contains the parameters we used for our findings, the tracer, and the analyzer—and can be used as reference to audit other areas of the system. It also serves as a good way to understand the instrumentation capabilities of Simics and SAE, even though these are, unfortunately, not open-source tools.

For the findings per se, almost all parameters appear to be probed and copied to local buffers inside of try-except blocks. We flagged them as double-fetches because some of the pointers are probed first and then accessed to copy out actual data, like `PUNICODE_STRING->Buffer`. One of them is not inside a try-catch block and is a local DoS, but we do not consider it a security issue, since it is in administrator-owned memory. Many of them are not related to Unicode strings and are potential escalations-of-privilege (see Figure 10), but once again, for the threat model of the Windows Kernel, administrator-initiated attacks are out of scope.

Microsoft nevertheless fixed some of the reported issues. Obviously, mitigations in kernel mode might still prevent or make exploiting some of those very difficult.

Our findings concern three classes of issues:

Admin ↔ kernel cases: Microsoft did fix these, even though their threat model does not consider this a security issue. They may have considered the possibility of these cases used for a CSP bypass or a sandbox bypass—even though we did not find cases where a sandboxed process had administrator privileges.

Local DoS cases: These were also fixed, considering that a symlink can be created by anyone and this was a non-admin-only case.

Other cases: The rest of the cases do not appear to be of consequence of security. We are sharing the traces with the community, in case anyone is interested in double-checking :)

[39] Mateusz Jurczyk and Gynvael Coldwind, "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns," Google, 2013. `unzip pocorgtfo15.pdf bochspwn.pdf`

[40] Nadav Chachmon et al., "Simulation and Analysis Engine for Scale-Out Workloads," Proceedings of the 2016 International Conference on Supercomputing (ICS '16), Istanbul, Turkey; `unzip pocorgtfo15.pdf chachmon.pdf`

[41] `git clone https://github.com/rrbranco/kdf ; unzip pocorgtfo15.pdf kdf.zip`

Tool Description

We implemented a Kernel Double Fetch tool (KDF), similar to the tool described in *Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns*.[42] The tool has a runtime phase, in which KDF candidates are identified, and a post-runtime phase, in which these KDF candidates are analyzed based on whether the fetches are actually used by the kernel.

In the runtime phase, there is a `ztool` that looks for system-call related instructions. When such an instruction is triggered, the tool will dynamically configure itself to enable memory access notifications and instruction execution notifications. Whenever the kernel reads from the same user-space address twice or more, the tool will generate a file that describes the assembly instructions and the memory access addresses. As an optimization, the tool analyzes each system call number only the first time it is called; consecutive calls to the same system call will not be analyzed. As correctly pointed out by J00ru, though, this optimization can hinder the discovery of some potential bugs that are only reached under very specific conditions—and not during the first invocation of the affected system call. The code can be easily changed to address that concern.

After this work has completed, the KDF candidates are filtered, and only if the kernel read the memory twice or more and performed some operation based on the read, a violation will be reported.

We make the KDF `ztool` source code public. You may get it from under `<zsim-kit>/src/ztools` and open the Visual Studio solution. Make sure you build an x64 version of the tool. (Look in the Visual Studio configuration.) After that you can load the tool when you boot Win10. The tool generates candidates for KDF in separate log file in the current working directory. After completing the run of the simulation you may use the `kdf_analyzer`. The real KDF candidates will be located in the results directory.

```
cd src/ztools/kdf
python3.4 kdf_analyzer \
  -id <zsim-simics-workspace> \
  -if <kdf-violations-basename> \
  -rd <results-directory>
```

[42] `http://research.google.com/pubs/pub42189.html`

Approach

The simulation tool is dependent on SAE, and runs as a plugin to it. It works by loading the KDF tool included in this paper, booting the OS, and executing whatever test bench; the plugin will capture suspicious violations. After stopping the simulation, the KDF-analyzer scans the suspected violations recorded by the plugin and outputs the confirmed cases of double-fetches. Note that while these are real double-fetches, they are not necessarily security issues.

The algorithm of the plugin works as follows. It starts the analysis upon a SYSCALL instruction, monitoring kernel reads from user addresses. It reports a violation on two reads from the same user-space address in the same instruction window. It stops the KDF analysis after Instruction-Window is reached in the same syscall scope, or upon a ring transition.

Performance is guaranteed since each syscall is instrumented only once and the instrumentation is enabled only in the system call range, supported by the tool itself.

The analyzer—responsible for post-analysis of the potential violations—is a Python script that manages the data flow dependencies. It adds a reference upon a copy from a suspected address to a register/address. It removes the dependency reference upon a write to a previously referenced register/memory, similar to a taint analysis. It reports a violation only if two or more distinct kernel reads happen from the same user-mode address.

We looked into the system call range 0–5081. We dynamically executed 450 syscalls within that range—meaning that our test bed is far from completely covering the entire range. The number of suspected cases flagged by the plugin was 67 and the number of violations identified was 8.

Interesting Cases

Figure 10 shows some of the interesting cases. The Windows version was build number 10240, TH1 RTM candidate.

You will find traces extracted from our tests in directories `win10_after_boot/` and `win8_after_boot/`. As the names imply, they were collected after booting the respective Windows versions by just using the system: opening calc, notepad, and the recycle bin.

API	Exploitable?	Why?
`nt!CmOpenKey`	No	`UNICODE_STRING`, Read the Unicode structure and then read the actual string. Both are properly probed.
nt!CmCreateKey	No	`UNICODE_STRING`
`nt!SeCaptureObject-AttributeSecurity-DescriptorPresent` nt!SeCaptureSecurity-Qos		
`nt!ObpCaptureObject-CreateInformation`	No	Reading and then Checking if NULL. Getting length, probing, and then copying data
nt!EtwpTraceMessageVa	No	Reading, checking against NULL, probing and then copying data
`nt!NtCreateSymbolic-LinkObject`	No	`UNICODE_STRING`, May lead to Local DOS. No try-catch on user mode address reference, at least not at the top function; it may be deeper in the call stack
win32kbase!bPEB-CacheHandle	No	Working on addresses of PEB structure and not on pointers, try-catch will save in case of a malformed PEB

Figure 10. Interesting cases.

The filenames include the system call number and the address of the occurrence, to help identify the repeated cases, e.g., `kdf-syscall-4101.log.data_flow_0x7ffe0320`, `kdf-syscall-4104.log.data_flow_0x7ffe0320`, `kdf-syscall-4105.log.data_flow_0x7ffe0320`. For example, the address `0x7ffe0320` repeats in both Win10 and Win8 traces. We kept these repeated traces just to facilitate the analysis.

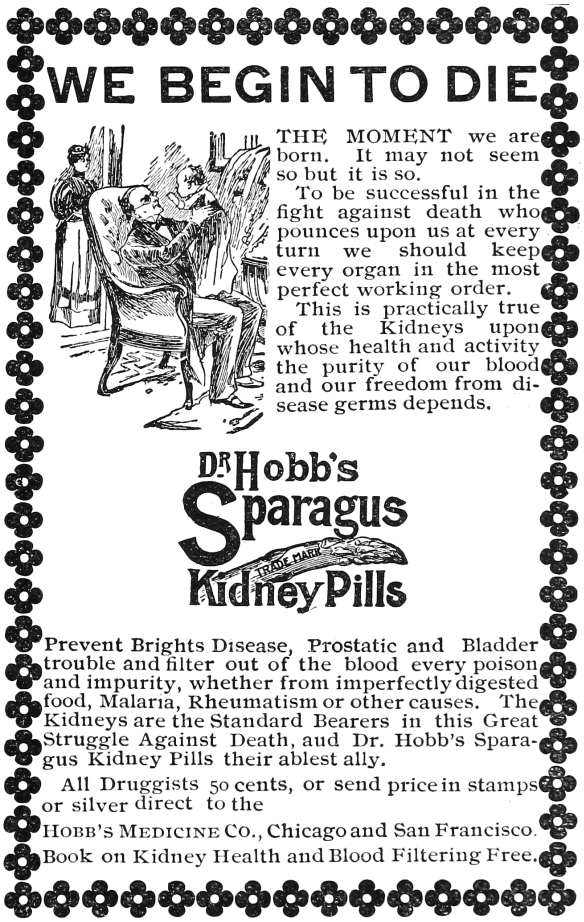
We also include the directories `results_-win10_boot/` and `result_win8_boot/`, which show the traces of interest *during* the boot process. These conditions are less likely to be exploitable, but some addresses in them repeat post-boot as well.

The format of trace files is quite straightforward, with comments inserted for events of interest:

```
--START ANALYZING KDF, ADDRESS: 0x2f7406f390
--  -> Defines the address of interest
```

Also included are the instructions performed during the analysis/trace:

```
180: 0xfffff803650acdd4
       mov rcx, qword ptr [rbx+0x10]
READ: VA = 0x2f7406f390, LA = 0x2f7406f390,
       PA1 = 0x79644390, SIZE = 0x8,
       DATA = 0x0002f746f3f8
```



The KDF detection happens on the following commentary on the trace:

```
--Data-flow dependency originated from
--line 180 is used: rcx
```

As you can see, the commentary includes the line at which the data-flow dependency was marked.

Our detection process begins when a `syscall` instruction is issued. While inside the call, we analyze kernel reads from the user address space, and report whenever two reads hit the same address; however, we remove references if a write is issued to the address. We stop the analysis once an instruction threshold is hit, or a ring transition happens.

Future Work

Leveraging our method and the toolset should make the following tasks possible.

First, it should be possible to find multiple writes to the same user-mode memory area in the scope of a single system service. This is effectively the opposite of the current concept of a violation. This may potentially find instances of accidentally disclosed sensitive data, such as uninitialized pool bytes, for a short while, before such data is replaced with the actual system call result.

Second, it should be possible to trace execution of code with `CPL=0` from user-mode virtual address space, a condition otherwise detected by the SMEP mechanism introduced in the latest Intel processors. Similarly, it should be possible to trace execution of code from non-executable memory regions that are not subject to Data-Execution-Prevention, such as non-paged pools in Windows.

Third, KDF should be studied on more operating systems.

Last but not least, other cases of cross-privilege mode double fetches should be investigated. There is far more work left to be done in tracing access to find these sorts of bugs.

Acknowledgments

We would like to thank Google researchers Mateusz Jurczyk and Gynvael Coldwind for releasing an awesome paper on the subject with enough details to reproduce their findings. (Mateusz was also kind enough to give feedback on this paper.) MSRC for helping to better define the threat model for Windows Kernel Vulnerabilities, and for their collaboration to triage the issues. We also thank Intel's Windows OS Team, specially Deepak Gupta and Volodymyr Pikhur, for their help in the analysis of the artifacts.

Is your washroom breeding

Insider Threats?

Employees lose respect for a company that fails to provide decent facilities for their comfort

TRY wiping your hands six days a week on harsh, cheap paper towels or awkward, unsanitary roller towels—and maybe you, too, would grumble.

Towel service is just one of those small, but important courtesies—such as proper air and lighting—that help build up the goodwill of your employees.

That's why you'll find clothlike ScotTissue Towels in the washrooms of large, well-run organizations such as R.C.A. Victor Co., Inc., National Lead Co. and Campbell Soup Co.

ScotTissue Towels are made of "thirsty fibre". . . an amazing cellulose product that drinks up moisture 12 times as fast as ordinary paper towels. They feel soft and pliant as a linen towel. Yet they're so strong and tough in texture they won't crumble or go to pieces . . . even when they're wet.

And they cost less, too—because one is enough to dry the hands—instead of three or four.

Write for free trial carton. Scott Paper Company, Chester, Pennsylvania.

ScotTissue Towels - *really* dry!

Reprinted by the TRACT ASSOCIATION OF PoC||GTFO AND FRIENDS

15:11 X86 is Turing-Complete without Data Fetches

by Chris Domas

One might expect that to compute, we must first somehow access data. Even the most primitive Turing tarpits generally provide some type of load and store operation. It may come as a surprise, then, that most modern architectures are Turing-complete without reading data at all!

We begin with the (somewhat uninspiring) observation that the effect of any traditional data fetch can be accomplished with a pure instruction fetch instead.

```
  data:
    .dword    0xdeadc0de
    mov       eax, [data]
```

That fetch in pure code would be a move sourced from an immediate value.

```
    mov       eax, 0xdeadc0de
```

With this, let us then model memory as an array of "fetch cells," which load data through instruction fetches alone.

```
  cell_0:
    mov       eax, 0xdeadc0de
    jmp       esi
  cell_1:
    mov       eax, 0xfeedface
    jmp       esi
  cell_2:
    mov       eax, 0xcafed00d
    jmp       esi
```

So to read a memory cell, without a data fetch, we'll `jmp` to these cells after saving a return address. By using a `jmp`, rather than a traditional function call, we can avoid the indirect data fetches from the stack that occur during a `ret`.

```
  mov       esi, mret              load return address
  jmp       cell_2                 load cell 2
mret:                              return
```

A data write, then, could simply modify the immediate used in the read instruction.

```
    mov       [cell_1+1], 0xc0ffee   set cell 1
```

Of course, for a proof of concept, we should actually compute something, without reading data. As is typical in this situation, the BrainFuck language is an ideal candidate for implementation — our fetch cells can be easily adapted to fit the BF memory model.

Reads from the BF memory space are performed through a `jmp` to the BF data cell, which loads an immediate, and jumps back. Writes to the BF memory space are executed as self modifying code, overwriting the immediate value loaded by the data cell. To satisfy our "no data fetch" requirement, we should implement the BrainFuck interpreter without a stack. The I/O BF instructions (`.` and `,`), which use an `int 0x80`, will, at some point, use data reads of course, but this is merely a result of the Linux implementation of I/O.

First, let us create some macros to help with the simulated data fetches:

```
%macro simcall 1
  mov       esi, %%retsim
  jmp       %1
%%retsim:
%endmacro

%macro simfetch 2
  mov       edi, %2
  shl       edi, 3
  add       edi, %1
  mov       esi, %%retsim
  jmp       edi
%%retsim:
%endmacro

%macro simwrite 2
  mov       edi, %2
  shl       edi, 3
  add       edi, %1+1
  mov       [edi], eax
%%retsim:
%endmacro
```

Next, we'll compose the skeleton of a basic BF interpreter:

```
_start:
.execute:
  simcall   fetch_ip
  simfetch  program, eax

  cmp       al, 0
  je        .exit
  cmp       al, '>'
  je        .increment_dp
  cmp       al, '<'
  je        .decrement_dp
  cmp       al, '+'
  je        .increment_data
  cmp       al, '-'
  je        .decrement_data
  cmp       al, '['
  je        .forward
  cmp       al, ']'
  je        .backward
  jmp       done
```

Then, we'll implement each BF instruction without data fetches.

```
.increment_dp:
  simcall   fetch_dp
  inc       eax
  mov       [dp], eax
  jmp       .done



.decrement_dp:
  simcall   fetch_dp
  dec       eax
  mov       [dp], eax
  jmp       .done



.increment_data:
  simcall   fetch_dp
  mov       edx, eax
  simfetch  data, edx
  inc       eax
  simwrite  data, edx
  jmp       .done



.decrement_data:
  simcall   fetch_dp
  mov       edx, eax
  simfetch  data, edx
  dec       eax
  simwrite  data, edx
  jmp       .done



.forward:
  simcall   fetch_dp
  simfetch  data, eax
  cmp       al, 0
  jne       .done
  mov       ecx, 1



.forward.seek:
  simcall   fetch_ip
  inc       eax
  mov       [ip], eax
  simfetch  program, eax
  cmp       al, ']'
  je        .forward.seek.dec
  cmp       al, '['
  je        .forward.seek.inc
  jmp       .forward.seek
.forward.seek.inc:
  inc       ecx
  jmp       .forward.seek
.forward.seek.dec:
  dec       ecx
  cmp       ecx, 0
  je        .done
  jmp       .forward.seek
```

```
.backward:
  simcall   fetch_dp
  simfetch  data, eax
  cmp       al, 0
  je        .done
  mov       ecx, 1
.backward.seek:
  simcall   fetch_ip
  dec       eax
  mov       [ip], eax
  simfetch  program, eax
  cmp       al, '['
  je        .backward.seek.dec
  cmp       al, ']'
  je        .backward.seek.inc
  jmp       backward.seek
.backward.seek.inc:
  inc       ecx
  jmp       .backward.seek
.backward.seek.dec:
  dec       ecx
  cmp       ecx, 0
  je        .done
  jmp       .backward.seek


.done:
  simcall   fetch_ip
  inc       eax
  mov       [ip], eax
  jmp       .execute


.exit:
  mov       eax, 1
  mov       ebx, 0
  int       0x80
```

Finally, let us construct the unusual memory tape and system state. In its data-fetchless form, it looks like this.

```
fetch_ip:

  db        0xb8                    mov eax, xxxxxxxx
ip:
  dd        0
  jmp       esi
fetch_dp:
  db        0xb8                    mov eax, xxxxxxxx
dp:
  dd        0
  jmp       esi
data:
  times     30000 \
  db        0xb8, 0, 0, 0,          mov eax, xxxxxxxx, jmp
            0, 0xff, 0xe6, 0x90     esi, nop
program:
  times     30000 \
  db        0xb8, 0, 0, 0,          mov eax, xxxxxxxx, jmp
            0, 0xff, 0xe6, 0x90     esi, nop
```

For brevity, we've omitted the I/O functionality from this description, but the complete interpreter source code is available.[43]

And behold! a functioning Turing machine on x86, capable of execution without ever touching the data read pipeline. Practical applications are nonexistent.

[43] `git clone https://github.com/xoreaxeaxeax/tiresias || unzip pocorgtfo15.pdf tiresias.zip`

15:12 Nail in the Java Key Store Coffin

by Tobias "Floyd" Ospelt

The Java Key Store (JKS) is Java's way of storing one or several cryptographic private and public keys for asymmetric cryptography in a file. While there are various key store formats, Java and Android still default to the JKS file format. JKS is one of the file formats for Java key stores, but the same acronym is confusingly also used the general key store API. This article explains the security mechanisms of the JKS file format and how the password protection of the private key can be cracked. Due to the unusual design of JKS, we can ignore the key store password and crack the private key password directly.

By exploiting a weakness of the Password Based Encryption scheme for the private key in JKS, passwords can be cracked very efficiently. As no public tool was available exploiting this weakness, we implemented this technique in Hashcat to amplify the efficiency of the algorithm with higher cracking speeds on GPUs.

The JKS File Format

Examples and API documentation for developers use the JKS file format heavily, without any security warnings.[44] This format has been the default key store since key stores were introduced to Java. As early as 1999, JDK 1.2 introduced the "-much stronger" JCEKS format that uses 3DES.[45] However, JKS remained the default format. Just to mention some examples, Oracle databases and the Apache Tomcat webserver still use the JKS format to store their private keys.

When building an Android 7 app in the Android Studio IDE, it will create a JKS file with which to self-sign the app. Every application on Android needs to be signed before it can be installed on a device, and the phone will check that an update for an app is signed with the same key again. The private keys generated by Android Studio are valid for 25 years by default. Android does not offer any recovery mechanism to recover a lost private key, so efficient cracking of JKS files also benefits developers who forgot their passwords.

The JKS format is due to be replaced by PKCS12 as the default key store format in the upcoming Java 9.[46] When talking to members of the security community who can still remember the nineties, some seem to remember that JKS uses some kind of weak cryptography, but nobody remembers exactly. Let's explore weaknesses of the JKS file format and what an attacker needs to extract a private key in cleartext.

When a new key store is created and a new keypair generated, the developer has to set at least two passwords. There is not only a password for the key store as a whole (key store password), but each private key in it has its own password as well (private key password), while public keys do not have passwords. Both passwords are used independently. Surprisingly, the key store password is not used to encrypt any parts of the JKS file format, it is only used for integrity protection. This means the encrypted private key bytes and the cleartext bytes of public keys in a key store can be extracted without knowing the key store password.[47] The password of the private key however, is used to apply a custom Password Based Encryption to the private key. Having two passwords leads to three possible cases.

In the first case, there is a password on the key store, but no private key password is used. (In practice, the available Java APIs prevent this.) However, in such a key store the private key would not be protected at all.

The second case is when the key store password and the private key password are identical. This is very common in practice and the default behavior of most tools such as Java's `keytool` command. If no separate password for the private key is specified, the private key password will be set to the key store password.

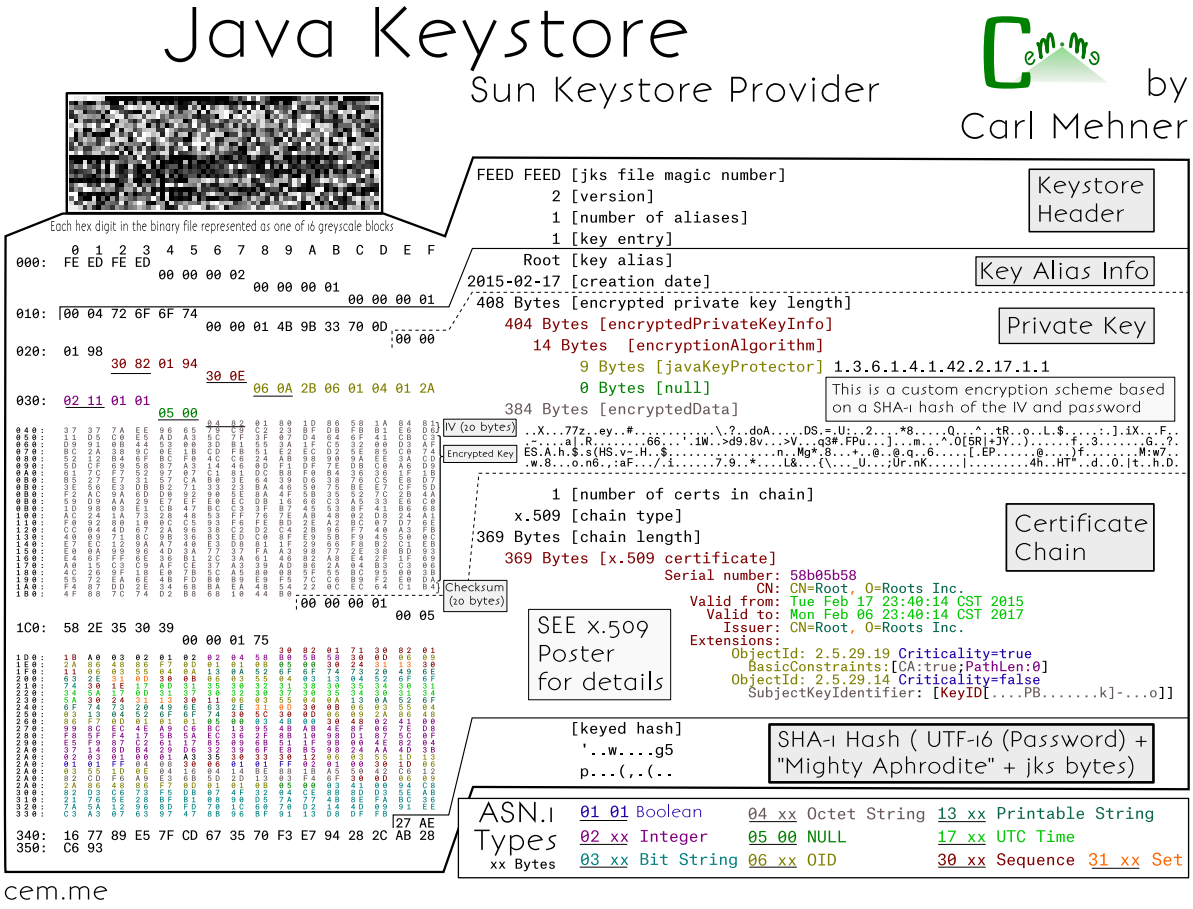
In the third case, both passwords are set but the

[44] `http://docs.oracle.com/javase/6/docs/api/java/security/KeyStore.html#getDefaultType()`
`http://download.java.net/java/jdk9/docs/api/java/security/KeyStore.html#getDefaultType--`
`https://developer.android.com/reference/java/security/KeyStore.html#getDefaultType()`
`http://stackoverflow.com/questions/11536848/keystore-type-which-one-to-use`
`http://www.pixelstech.net/article/1408345768-Different-types-of-keystore-in-Java----Overview`

[45] See Dan Boneh's notes on JCE 1.2 from CS255, Winter of 2000.

[46] `http://openjdk.java.net/jeps/229`

[47] `https://gist.github.com/zach-klippenstein/4631307`



key store password is not the same as the private key password. While not the default behavior, it is still very common that users choose a different password for the private key.

It is important to demonstrate that in the third case some password crackers will crack a password that is useless and cannot be used to access the private key. The Jumbo version of the John the Ripper password cracking tool does this, cracking the (useless) key store password rather than the private key password. Let's generate a key store with different key store (`storepass`) and private key password (`keypass`), then crack it with John:

```
$ keytool -genkey -dname                                    \
  'CN=test, OU=test, O=test, L=test, S=test, C=CH'          \
  -noprompt -alias mytestkey -keysize 512                   \
  -keyalg RSA -keystore rsa_512.jks                         \
  -storepass 1234567 -keypass 7654321
$ pypy keystore2john.py rsa_512.jks > keystore.txt
$ /opt/john-1.8.0-jumbo-1/run/john                          \
  --wordlist=wordlist.txt keystore.txt
[...]
1234567          (rsa_512.jks)
[...]
```

While this reveals the `storepass`, we cannot access the private key with this password. My proof of concept will crack the private key password instead:[48]

```
$ java -jar JksPrivkPrepare.jar rsa_512.jks > privkey.txt
$ pypy jksprivk_crack.py privkey.txt
Password: '7654321'
```

Naive Password Cracking

If we take the perspective of an attacker, we can conclude that we will not need to crack any password in the first case to get access to the private key. In theory, it also doesn't matter which password we find out in the second case, as both are the same. And in the third case we can simply ignore the key store password; we only need to crack attack the private key password.

However, when we encounter the second case in practice, we would like to use the most efficient

[48] `unzip -j pocorgtfo15.pdf jksprivk/JksPrivkPrepare.jar jksprivk/jksprivk_crack.py`

password cracking technique to find the key store password or the private key password. This means we need to explore first how each password can be cracked individually and which one leads to the most efficient cracking method.

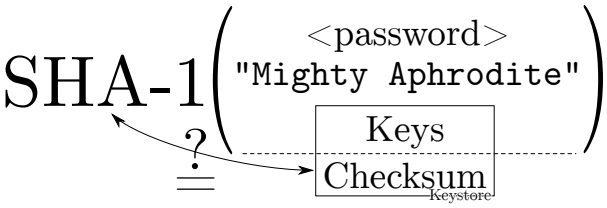
There are already several programs that will try to crack the password of the key store:

- John the Ripper (JtR) Jumbo version[49] extracts necessary information with a Python script and the cracking is implemented in C;
- KeyStoreBrute[50] tries to load the key store via the official Java method in Java;
- KeystoreCracker[51] uses the simple official Java way in Java as well;
- keystoreBrute[52] uses `keytool` on the command line with the `storepass` option (subprocess);
- bruteforcer.py[53] uses `keytool` on the command line with the `storepass` option (subprocess);
- Patator[54] uses `keytool` on the command line with the `storepass` option (subprocess).

All these parse the JKS file format first, which has a SHA-1 checksum at the end. They then calculate a SHA-1 hash consisting of the password, the magic "`Mighty␣Aphrodite`" and all bytes of the key store file except for the checksum. If the newly calculated hash matches the checksum, it was the correct password.

No other operation with the key store password takes place when parsing the JKS file format; therefore, we can conclude that this password is only used for integrity protection. When the correct password is guessed and it is the same as the private key password, an attacker can now decrypt the private key.

From a performance perspective, this means that for every potential password a SHA-1 hash needs to be calculated of nearly all bytes of the key store file. As key stores usually hold private and public keys of at least 512-byte length, the SHA-1 hash is calculated over several thousand bytes of input. To summarize, the effort to check one password for validity is roughly:



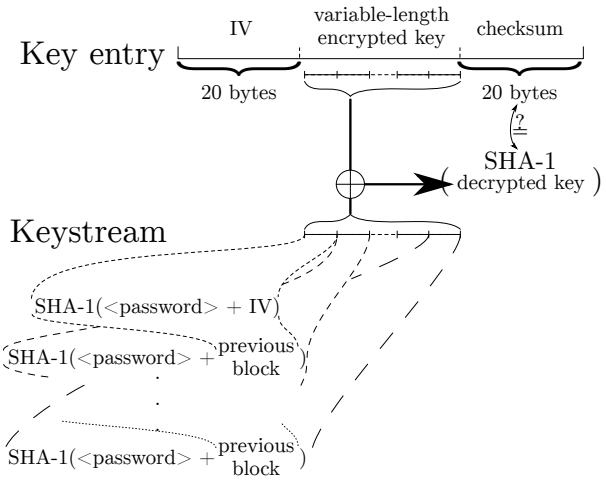
It is also important to emphasize again that the above implementations will waste CPU time if the key store password is not identical to the private key password (third case) and are not attempting to crack the password necessary to extract the private key.

There are also implementations that crack the password of the private key directly:

- android-keystore-recovery[55] tries to decrypt the entire private key with each password, in Scala;
- android-keystore-password-recover[56] tries to decrypt the entire private key with each password, in Java.

These implementations have in common that they parse the JKS file format, but then only extract the entry of the encrypted private keys. For each private key entry, the first 20 bytes serve as an Initialization Vector and the last 20 bytes are again a checksum. The implementations then calculate a keystream. The keystream starts as the SHA-1 hash of the password plus IV. For every 20 bytes of the encrypted private key, the next 20 bytes of the keystream are calculated as the SHA-1 of the password plus previous keystream block (of 20 bytes). The encrypted private key bytes are then XORed with the keystream to get the private key in cleartext. This is a custom Password Based Encryption (PBE) scheme with chaining. As a last step, the cleartext private key is SHA-1 hashed again and compared to the checksum that was extracted from the JKS private key entry. Therefore, the effort to check one password for validity is roughly:

[49] `http://www.openwall.com/lists/john-users/2015/06/07/3`
[50] `git clone https://github.com/bes/KeystoreBrute`
[51] `git clone https://github.com/jeffers102/KeystoreCracker`
[52] `git clone https://github.com/volure/keystoreBrute`
[53] `https://gist.github.com/robinp/2143870`
[54] `https://www.darknet.org.uk/2015/06/patator-multi-threaded-service-url-brute-forcing-tool/`
[55] `https://github.com/rsertelon/android-keystore-recovery`
[56] `https://github.com/MaxCamillo/android-keystore-password-recover`



Efficient Password Cracking

From a naive perspective, it was not analyzed which of these algorithms would be more efficient for password cracking.[57] However, an article on Cryptosense.com was published in 2016[58] and didn't seem to get the attention it deserves. It points out that for the private key password cracking method it is not necessary to calculate the entire keystream to reject an invalid password. As the cleartext private key will be a DER encoded file format, the first SHA-1 calculation of password plus IV with the XOR operation is sufficient to check if a password candidate could potentially lead to a valid DER encoded private key. These all miss out on this optimization and therefore do too many SHA-1 calculations for every password candidate.

It turns out, it is even possible to pre-calculate the XOR operation. For each password candidate only one SHA-1 hash needs to be calculated, then some bytes of the result have to be compared to the pre-calculated bytes. If the bytes are identical, this proves that the password might decrypt the key to a DER format. Practical tests showed that a DER encoded RSA private key in cleartext will start with `0x30` and bytes at index six to nineteen will be `0x00300d06092a864886f70d010101`. Similar fingerprints exist for DSA and EC keys. These bytes we expect in a DER encoded private key can be XORed with the corresponding encrypted private key bytes to precalculate the SHA-1 output bytes we are looking for.

This means, the cracking can be optimized to use a more efficient two-step cracking algorithm to crack the private key password. After parsing the JKS file format and precalculating the necessary values, we have the following optimized algorithm:

0. Choose a password in pseudo UTF-16, meaning that a null byte is added to every character.
1. `keystream = SHA-1(password + STATIC_20_BYTES_IV_FROM_PRIVKEY_ENTRY)`
2. Check if bytes at index 0 and 6 to 19 of the keystream correspond to `PRECOMPUTED_15_BYTES_DER_PROOF`. If they are not the same, go to step 0.
3. Let `keybytes` be every 20 bytes of `STATIC_VARIABLE_LEN_ENCRYPTED_BYTES_FROM_PRIVKEY_ENTRY`.
4. For each `keybytes`:
   (a) `key += keystream` $\oplus$ `keybytes`
   (b) `keystream = SHA-1(password‖keystream)`
5. `checksum = SHA-1(password‖key)`
6. Check if `checksum` is `STATIC_20_BYTES_CHECKSUM_FROM_PRIVKEY_ENTRY`. If they are the same, key is the private key in cleartext and we can stop. Otherwise, go to step 0.

As practical tests will later indicate, step 3 is typically never reached with an incorrect password during cracking and all passwords can be rejected early. In fact, Hashcat only implements steps 0 to 3, as the probability that a wrong candidate is ever found is neglectible ($1/2^{120}$)!

Implementation

The parsing of the file format and extraction of the precomputed values for cracking were implemented as a standalone JAR Java version 8 command line application `JksPrivkPrepare.jar`. The script will

[57] While the key store calculations must do the single SHA-1 over all bytes of the public and private keys in the key store, the private key calculations are many more SHA-1 calculations but with less bytes as inputs.

[58] Might Aphrodite – Dark Secrets of the Java Keystore

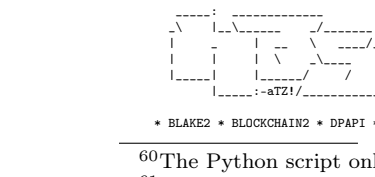
[59] Running much faster with the PyPy Python implementation rather than CPython. The script works without further dependencies. However, another script in the benchmark section needs the numpy packet. It has to be installed for PyPy. The easiest way of installing is usually via PIP: `pypy -m pip install numpy`

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_123456.jks           \
  -storepass 123456 -keypass 123456
$ java -jar JksPrivkPrepare.jar rsa_512_123456.jks > privkey_123456.txt
$ pypy -m cProfile -s tottime jksprivk_naive_crack.py privkey_123456.txt
Password: '123456'
         10278681 function calls (10277734 primitive calls) in 9.763 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   123457    2.944    0.000    2.944    0.000 jksprivk_naive_crack.py:14(xor)
  2345683    1.651    0.000    1.651    0.000 {method 'digest' of 'HASH' objects}
  2345684    1.608    0.000    1.608    0.000 {_hashlib.openssl_sha1}
  2345683    1.491    0.000    5.266    0.000 jksprivk_naive_crack.py:19(get_keystream)
[...]
$ pypy -m cProfile -s tottime jksprivk_crack.py privkey_123456.txt
Password: '123456'
         649118 function calls (648171 primitive calls) in 0.438 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   123476    0.086    0.000    0.086    0.000 {method 'digest' of 'HASH' objects}
   123477    0.067    0.000    0.067    0.000 {_hashlib.openssl_sha1}
        1    0.056    0.056    0.293    0.293 jksprivk_crack.py:54(get_candidates)
       14    0.055    0.004    0.486    0.035 __init__.py:1(<module>)
[...]
```

Figure 11. Java Key Store with a Short Password

prepare the precomputed values for a given JKS file and outputs it as asterix separated values.

As a PoC, a Python script `jksprivk_crack.py`[59] was implemented to do the actual cracking of the private key password. To put a final nail in the coffin of the JKS format, it is important to enable the security community to do efficient password cracking.[60] To optimize cracking speed, Jens "atom" Steube — developer of the Hashcat password recovery program — implemented the cracking step in GPU optimized code. Hashcat takes the same arguments as the Python cracking script. As hashcat uses a weakness in SHA-1,[61] the cracking speed on a single NVidia GTX 1080 GPU reaches around 7.8 (stock clock) to 8.5 (overclocked) billion password tries per second.[62] This allows to try all alphanumeric passwords (uppercase, lowercase, numbers) of length eight in about eight hours on a single GPU.



Benchmarking

When doing a benchmark, it is important to try to measure the actual algorithm and not some inefficiency of the implementation. Some simple measurements were done by implementing the described techniques in Python. All the mentioned resources are available in the feelies.[63] Let's first look at the naive implementation of the private key cracker `jksprivk_naive_crack.py` versus the efficient private key cracking algorithm `jksprivk_crack.py`. Let's generate a test JKS file first. We can generate a small 512-byte RSA key pair with the password 123456, then crack it with both implementations. Both implementations only try numeric passwords, starting with length 6 password 000000 and incrementing, as in Figure 11.

These measurements show that a lot more calls to the update and digest function of SHA-1 are necessary to crack the password in the naive script. If the keysize of the private key in the JKS store is bigger, the time difference is even greater. Therefore, we conclude that our efficient cracking method is far

[60] The Python script only reaches around 220,000 password-tries per second when run with PyPy on a single 3-GHz CPU.
[61] `https://hashcat.net/events/p12/js-sha1exp_169.pdf`
[62] `git clone https://github.com/hashcat/hashcat`
[63] `unzip -j pocorgtfo15.pdf jksprivk/jksprivk_resources.zip`

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_12345678.jks          \
  -storepass 12345678 -keypass 12345678
$ java -jar JksPrivkPrepare.jar rsa_512_12345678.jks > privkey_12345678.txt
$ pypy -m cProfile -s tottime jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
         116760228 function calls (116759281 primitive calls) in 60.009 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 23345699   16.940    0.000   16.940    0.000 {_hashlib.openssl_sha1}
 23345698   16.082    0.000   16.082    0.000 {method 'digest' of 'HASH' objects}
 23345775   10.971    0.000   10.972    0.000 {method 'join' of 'str' objects}
        1    8.560    8.560   59.851   59.851 jksprivk_crack.py:54(get_candidates)
 23345698    4.024    0.000    4.024    0.000 {method 'update' of 'HASH' objects}
 23345679    3.274    0.000   14.245    0.000 jksprivk_crack.py:91(next_brute_force_token)
 [...]
$ pypy /opt/john-1.8.0-jumbo-1/run/keystore2john.py rsa_512_12345678.jks \
  > keystore_12345678.txt
$ pypy -m cProfile -s tottime jkskeystore_crack.py keystore_12345678.txt
Password: '12345678'
         163420866 function calls in 84.719 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 70037037   33.712    0.000   33.712    0.000 {method 'update' of 'HASH' objects}
 23345679   17.780    0.000   17.780    0.000 {method 'digest' of 'HASH' objects}
 23345680   12.022    0.000   12.022    0.000 {_hashlib.openssl_sha1}
 23345682    9.679    0.000    9.679    0.000 {method 'join' of 'str' objects}
        1    8.482    8.482   84.716   84.716 jkskeystore_crack.py:14(crack_password)
 23345679    3.042    0.000   12.721    0.000 jkskeystore_crack.py:26(next_brute_force_token)
 [...]
```

Figure 12. Java Key Store with a Longer Password

more suitable.

Now we still have to compare the efficient cracking of the private key password with the cracking of the key store password. The algorithm for key store password cracking was also implemented in Python: `jkskeystore_crack.py`. It takes a password file as argument like John the Ripper does. As these implementations are more efficient, let's generate a new JKS with a longer password, as shown in Figure 12.

In this profile, we see that the update method of the SHA-1 object when cracking the key store takes much longer to return and is called more often, as more data goes into the SHA-1 calculation. Again, the efficient cracking algorithm for the private key is faster and the difference is even bigger for bigger key sizes.

So far we tried to compare techniques in Python. As they use the same SHA-1 implementation, the benchmarking was kind of fair. Let's compare two vastly different implementations, the efficient algorithm `jksprivk_crack.py` to John the Ripper. First, create a wordlist for John with the same numeric passwords as the Python script will try, then run the comparison shown in Figure 13.

That figure shows that John is faster for 512-bit keys, but as soon as we grow to 1024-bit keys in Figure 14, we see that our humble little Python script wins the race against John. It's faster, even without John's fancy C code or optimizations!

As John the Ripper needs to do SHA-1 operations for the entire key store content, the Python script outperforms John the Ripper. For larger key sizes, the difference is even bigger.

These benchmarks were all done with CPU calculations and Hashcat will use performance optimized GPU code and Markov Chains for password generation. Cracking a JKS with private key password `POC||GTFO` on a single overclocked NVidia GTX 1080 GPU is illustrated on Figure 15.

Neighborly Greetings

Neighborly greetings go out to atom, vollkorn, cem, doegox, ange, xonox and rexploit for supporting this article in one form or another.

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_12345678.jks          \
  -storepass 12345678 -keypass 12345678
$ java -jar JksPrivkPrepare.jar rsa_512_12345678.jks > privkey_12345678.txt
$ time pypy jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
        54.96 real        53.76 user         0.71 sys
$ pypy /opt/john-1.8.0-jumbo-1/run/keystore2john.py rsa_512_12345678.jks           \
  > keystore_12345678.txt
$ time /opt/john-1.8.0-jumbo-1/run/john --wordlist=wordlist.txt keystore_12345678.txt
[...]
12345678         (rsa_512_12345678.jks)
[...]
        42.28 real        41.55 user         0.33 sys
```

Figure 13. John the Ripper is faster for 512-byte keystores.

```
$ time pypy jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
        58.17 real        56.36 user         0.84 sys
$ time /opt/john-1.8.0-jumbo-1/run/john --wordlist=wordlist.txt keystore_12345678.txt
[...]
12345678         (rsa_1024_12345678.jks)
[...]
        64.60 real        62.96 user         0.57 sys
```

Figure 14. For 1024-bit keystores, our script is faster (full output in the feelies).

```
$ ./hashcat -m 15500 -a 3 -1 '?u|' -w 3 hash.txt ?1?1?1?1?1?1?1?1?1
hashcat (v3.6.0) starting...
[...]
* Device #1: GeForce GTX 1080, 2026/8107 MB allocatable, 20MCU
[...]
$jksprivk$*D1BC102EF5FE5F1A7ED6A63431767DD4E1569670...8*test:POC||GTFO
[...]
Speed.Dev.#1.....:  7946.6 MH/s (39.48ms)
[...]
Started: Tue May 30 17:41:56 2017
Stopped: Tue May 30 17:50:24 2017
```

Figure 15. Cracking session on a NVidia GTX 1080 GPU.

15:13 The Gamma Trick: Two PNGs for the price of one

by Hector Martin 'marcan'

Say you're browsing your favorite hypertext-encoded, bitmap-containing visuo-lingual information distribution medium. You come across an image which—as we do not yet live in an era of infinitely scalable resolution—piques your interest yet is presented as a small thumbnail. *Why are they called thumbnails, anyway?*

Despite the clear instructions not to do so, you resolve to click, tap, press enter, or otherwise engage with the image. After all, you have been conditioned to expect that such an action will yield a higher-quality image through some opaque and clearly incomprehensible process.

Yet the image now appearing before your eyes is not the same image that you clicked on. Curses! What is this sorcery? Have I been fooled? Is this alien technology? *Did someone hack Reddit?*

The first time I came across this technique was a few years ago on a post on 4chan. Despite the fact that the image was not just lewd but downright unsavory to my taste, I have to admit I spent quite some time analysing exactly what was going on in detail. I have since seen this trick used a few times here and there, and indeed I've even used a variant of it myself in a CTF challenge. Thanks go to my friend `@Miluda` for giving me permission to use her art in this article's examples.

So, do tell, what is going on? It all has to do with the PNG format. Like most image formats, PNG images carry metadata. That metadata includes information about how the image, and in particular color information, is itself encoded. The PNG format can specify how RGB values map to how much light comes out of the pixels on your screen in several ways, but one of the simplest is the `'gAMA'` chunk which specifies the gamma value of the image, γ.

Intuitively, you'd think that a pixel with 50% brightness would be encoded as a 0.5 value (or about `0x7f`, in an 8-bit format), but that is not the case. Due to a series of historical circumstances and practical coincidences too long-winded to be worth going into, pixel brightness values are not linear. Instead, they are stored as the brightness value raised to a power γ. The most common default is $\gamma = 0.4545$. When the image is displayed, the pixels are raised to the inverse gamma, 2.2, to obtain the linear brightness value.[64] This is typically done by your monitor. Thus, 50% brightness is actually encoded as 0.73, or `0xba`. PNG images can specify an alternate γ value, and your PNG decoder is responsible for converting it to the correct display gamma.

Like every other optional feature of every other file format, whether this is actually implemented is anyone's guess. As it turns out, most web browsers implement it properly, and most image processing libraries do not. Many websites use these to create thumbnails: Reddit, 4chan, Imgur, Google Docs. We can use this to our advantage.

Take one source image and darken it (map its brightness range to 0%..80%). Take the other source image, and lighten it (map its brightness range to 80%..100%). The two images now occupy distinct portions of the brightness gamut. Now, for every `2x2` group of pixels, take 3 pixels of the darker image and 1 pixel of the lighter image. Finally, encode the result as a PNG and apply the gAMA PNG tag, using an extreme value such as γ=0.0227. (Twenty times lower than the default γ=0.4545.)

[64] Most computers these days use, or at least claim to support, the sRGB colorspace, which doesn't actually use a pure gamma function for a bunch of technical reasons. But it approximates $\gamma = 2.2$, so we're rolling with that.

We can do this easily enough with ImageMagick:

```
$ size=$(convert "$high" -format "%wx%h" info:)
$ convert \( "$low" -alpha off +level 0%,80% \) \
      \( "$high" -alpha off +level 80%,100% \) \
      -size $size pattern:gray25 -composite    \
      -set gamma 0.022727                      \
      -define png:include-chunk=none,gAMA      \
      "$output"
```

When viewed without the specified gamma correction, all of the lighter pixels (25% of the image) approach white and the overall image looks like a washed out version of the darker source image (75% of the image). The 2×2 pixel pattern disappears when the image is downscaled to less than half of its original dimensions (if the scaler is any good anyway). When the gamma correction *is* applied to the original image, however, all the darker pixels are crushed to black, and now the lighter pixels span most of the brightness spectrum, revealing the lighter image as a grid of bright pixels against a black background. If the image is displayed at `1:1` pixel scale, it will look quite clean. Scales between 100% and 50% typically result in moiré artifacts, because most scalers cheat. Scaling down usually darkens the image, because most scalers also don't do gamma-correct scaling.[65]

$\gamma = 0.4545$

$\gamma = 0.0227$

This approach is the one I've seen used so far, and it is easy to achieve using the Levels tool in GIMP, but we can do better. The second image is much too dark: we're mapping the image to a linear brightness range, but then applying a very much non-linear gamma correction. Also, in the first image, we can see a "halo" of the second image, since the information is actually there. We can fix these issues.

Let's use ImageMagick again. First we'll apply a true gamma adjustment to the high source image. The `-gamma` operation in ImageMagick performs an adjustment by the inverse of the supplied value, so to apply an adjustment of $\gamma = 1/20$ we'll pass in 20. We'll also slightly increase its brightness, to ensure that after gamma adjustment the pixels are close enough to white:

```
$ convert "$high" -alpha off +level 3.5%,100% \
                  -gamma 20 high_gamma.png
```

This effectively maps the image range to $0.035^{0.05} = 0.846..1.0$, but with a non-linear gamma curve. Next, because the low image will appear washed out, we'll apply a gamma of 0.8, then darken it to 77% of its original brightness. $0.77^{20} = 0.005$, which is dark enough to not be noticeable. We're keeping this in a variable to chain later.

```
$ low_gamma="-alpha off -gamma 0.8 +level 0%,77%"
```

Now let's compensate for the halo caused by the high image. For every 2x2 output pixels, we'd like an average color of:

$$v = 3/4v_{low} + 1/4$$

That is, as if the high image was completely white. What we actually have is:

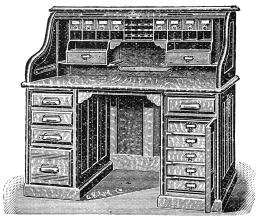
$$v = 3/4v'_{low} + 1/4v_{high}$$

Solving for v'_{low} gives:

$$v'_{low} = v_{low} - 1/3v_{high} + 1/3$$

We can implement this in ImageMagick using `-compose Mathematics`:

```
$ convert \( "$low" $low_gamma \) high_gamma.png \
          -compose Mathematics                   \
          -define compose:args='0,-0.33,1,0.33'  \
          -composite low_adjusted.png
```

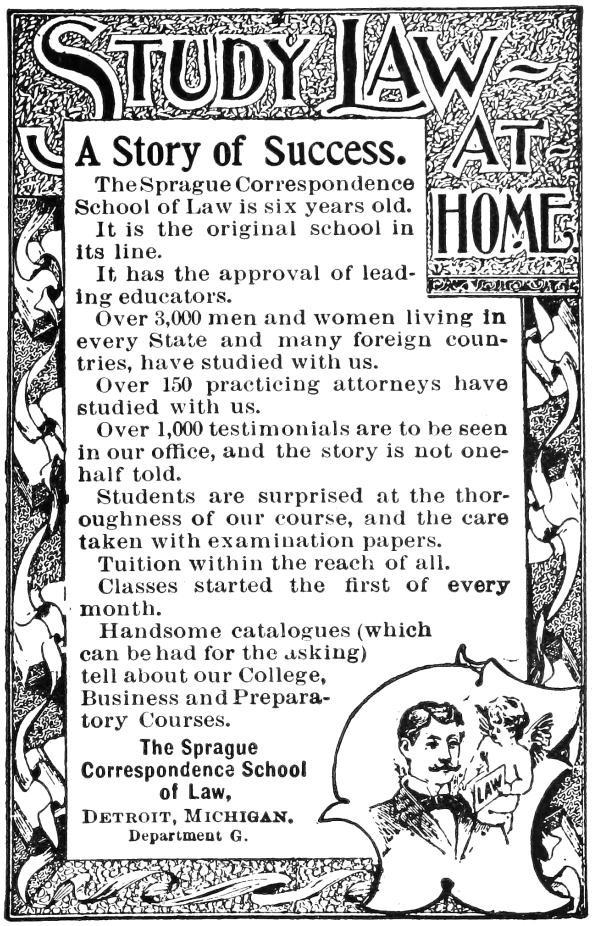


[65] Note that gamma-correct scaling is orthogonal to the gamma trick used here. A simple black-and-white checkerboard *should* be downscaled to a solid 0.73 gray (half the photons, or 50% brightness, at $\gamma = 0.4545$), but most scalers just average it down to 0.5, which is wrong. GIMP is one of the few apps that does gamma-correct scaling these days. Isn't gamma fun?

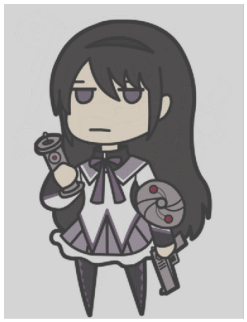
There will be some slight edge effects, due to aliasing issues between the chosen pixels from both images, but this will remove any blatant solid halo areas. This correction assumes that the thumbnail scaler does not perform gamma-correct scaling,[65] which is the common case. This means it is incorrect if the output image is viewed at 1:1 scale (the halo will be visible), but once scaled down it will disappear. In order to cater for gamma-correct scalers (or 1:1 viewing), we'd have to perform the adjustment in a linear colorspace.

Finally, we just compose both images together with a pattern as before:

```
$ convert low_adjusted.png high_gamma.png \
      -size $size pattern:gray25          \
      -composite -set gamma 0.022727      \
      -define png:include-chunk=none,gAMA \
      "$output"
```



The result is much better.



$\gamma = 0.4545$

$\gamma = 0.0227$

The previous images in this article have been filtered (2×2 box blur) to remove the high-frequency pixel pattern, in order to approximate how they would visually appear in a browser context without relying on the specific scaling/resampling behavior of your PDF renderer. In fact, the filtering method varies: gamma-naive for simulating thumbnailing, gamma-aware for simulating the true response at 1:1 scale. For your amusement, here are the raw images. Their appearance will depend on exactly what kind of filtering, scaling, or other processing is applied when the PDF is rasterized. Feel free to play with your zoom setting.

$\gamma = 0.4545$

$\gamma = 0.0227$

Yup, it's 2017 and most software still can't up/downscale images properly. Now don't get me started on the bane that is non-premultiplied alpha, but that's a topic for another day.

15:14 Laphroaig's Home for Unwanted Polyglots and 0day

from the desk of Pastor Manul Laphroaig,
International Church of the Weird Machines

Dearest neighbor,

If you enjoyed reading this little tract, I have some good news and a polite request for you.

Thanks to the fine folks at No Starch Press, our 768 page Book of PoC||GTFO is sailing on its merry way across the Pacific ocean![66] It includes full color file format illustrations by Ange Albertini, as well as every article from our first nine releases on thin paper with gold trim, faux leather binding, and a ribbon to keep your place. Each article has been revised, indexed, and cross referenced.

But today I'm writing to ask for your offering. Not an offering of money, but on offering of writing. Send me your proofs of concept!

Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. 8-bit ASCII is also acceptable if generated on TempleOS. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Buzzfeed listicle. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

Like an email, we expect informal language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

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

[66] Preorders accepted at `http://nostarch.com/gtfo`