

# NahamCon CTF 2022: Solutions

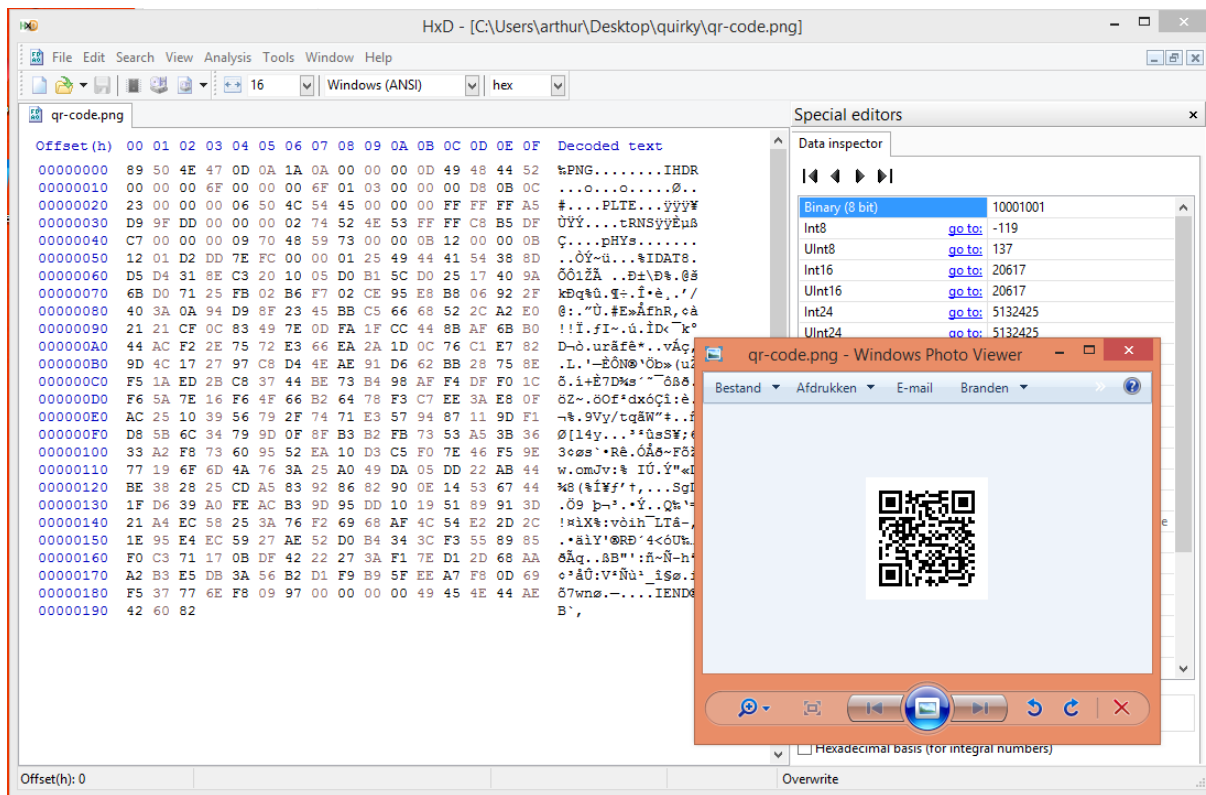
Arthur Verschaeve (hi@arthurverschaeve.be)

December 2022

## 1 Warmups

### 1.1 Quirky

The given text file looks like bytes with escape characters (\x). The first few bytes are 89 50 4e 47, which is the magic number of PNG images. I removed the escape characters and saved the bytes as a binary .png file, using the HxD freeware hex editor. The resulting image looks like a QR code.



There are a lot of utilities to read QR codes. I usually like zbar, but there are websites out there that can do it without an installation procedure too.

Reading the code yields the correct flag: `flag{b8e2a32f5ae629dcfb1ac210d1f0c032}`

### 1.2 Jurassic Park

This is a classic `robots.txt` challenge. I opened the given website, found out there's not much interesting in the source code, so decided to check out its `robots.txt` file. This revealed a directory, which had an open directory listing with a `flag.txt` file. That file contains `flag{c2145f65df7f5895822eb249e25028fa}`.

## 2 Cryptography

### 2.1 XORROX

The operations inside the given script can easily be reversed, because the XOR-operation itself is reversible: Applying the same XOR operation with the original to encrypted data will decrypt it. I did exactly that, copying part of the given loops, and pasting in the given values for `xorrox` and `enc`. After some simplifications (unused loop variables, for example), the script looks like this:

```
xorrox=[1, 209, 108, 239, 4, 55, 34, 174, 79, 117, 8, 222, 123, 99, 184, 202,
    ↪ 95, 255, 175, 138, 150, 28, 183, 6, 168, 43, 205, 105, 92, 250, 28, 80,
    ↪ 31, 201, 46, 20, 50, 56]
enc=[26, 188, 220, 228, 144, 1, 36, 185, 214, 11, 25, 178, 145, 47, 237, 70,
    ↪ 244, 149, 98, 20, 46, 187, 207, 136, 154, 231, 131, 193, 84, 148, 212,
    ↪ 126, 126, 226, 211, 10, 20, 119]

key = [1]
flag = ""

for i in range(len(enc)):
    k = xorrox[i]
    for j in range(i, 0, -1):
        k ^= key[j]
    key.append(k)
    flag += chr(enc[i] ^ k)

print(flag)
```

Executing this yields the following result:

```
ag{21571dd4764a52121d94deea22214402}
```

The first characters, "fl", are missing here because this reverse script doesn't perfectly imitate the start of the script, but that didn't matter, as we knew it was a `flag{...}`-formatted flag.

The flag is `flag{21571dd4764a52121d94deea22214402}`.

### 2.2 Unimod

The main (and only) operation in the given script is `chr((ord(c) + k) % 0xFFFD)`, with `k` a randomly chosen integer between 0 and 0xFFFD. This can be reversed using a subtraction (keeping the modulo), and for finding the value of `k` a loop can be used. The following script finds the flag:

```
output = open('out', 'r').read()
solution = ""

for k in range(0xFFFD):
    for c in output:
        solution += chr((ord(c) - k) % 0xFFFD)

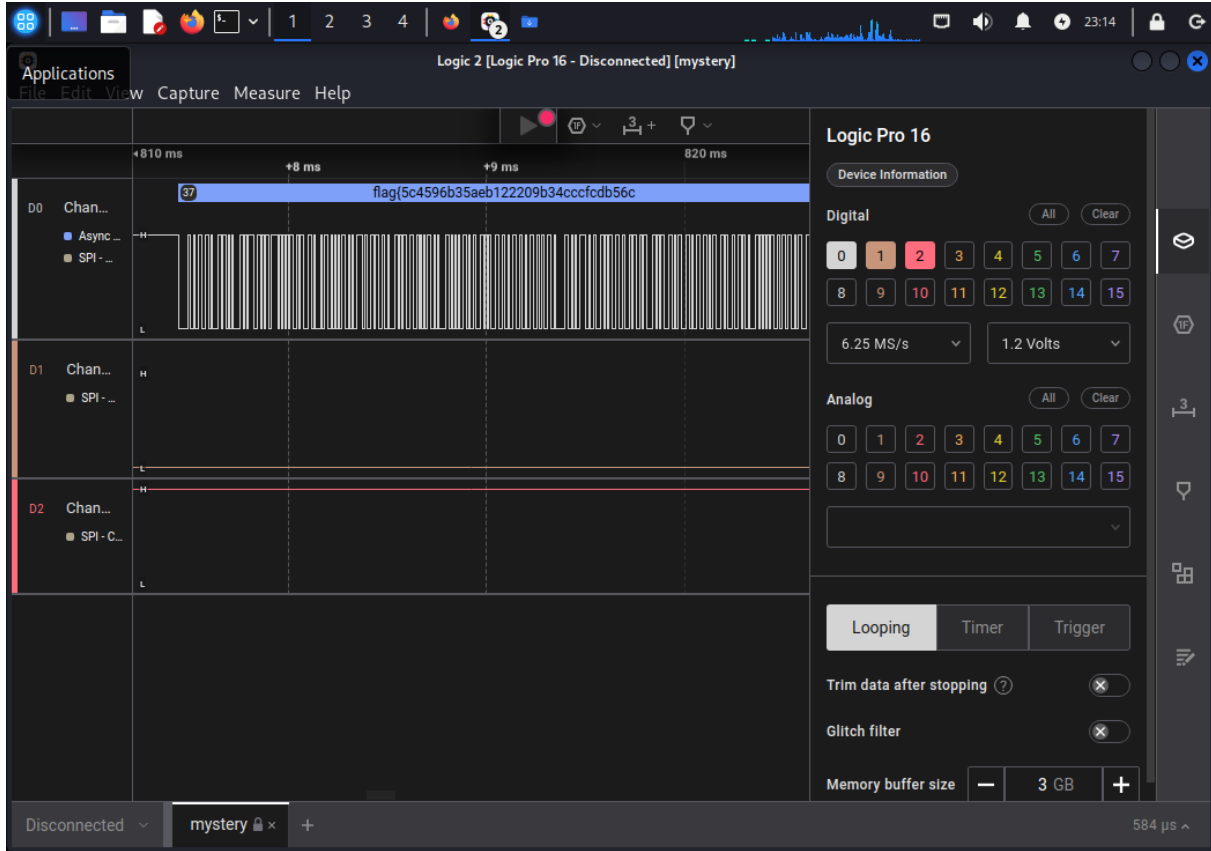
    if "flag" in solution:
        print(solution)
        solution = ""
    else:
        solution = ""
```

The flag is `flag{4e68d16a61bc2ea72d5f971344e84f11}`.

## 3 Hardware

### 3.1 Cereal

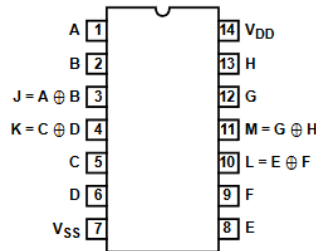
I didn't know what the given file was at all. Some searching for the `.sal`-extension didn't yield much results either, but there was one forum question on something that seemed like hardware design tooling<sup>1</sup>. I installed this *SALAE* - *Logic 2* tool and indeed, it could open the file. The electrical signal contained in the what was apparently a capture file contained the flag:



The flag is `flag{5c4596b35aeb122209b34ccfcdb56c1}`.

### 3.2 Dweeno

I decided to not verify using the actual picture, and assume the sketch PDF was accurate. The *4070* refers to XOR-gates, I found the datasheet for that component<sup>2</sup>. The following figure inside is all I really needed:



Meaning the following relations follow from the given sketch (where  $VX$  denotes the value on port  $X$ ):

<sup>1</sup><https://discuss.saleae.com/t/utilities-for-sal-files/725>

<sup>2</sup><https://pdf1.alldatasheet.com/datasheet-pdf/view/26894/TI/CD4070.html>

- $V_{49} = V_{25} \oplus 0$
- $V_{23} = V_{47} \oplus 1$
- $V_{51} = V_{27} \oplus 1$
- $V_{29} = V_{53} \oplus 0$

The first few lines of code within the code file indicate which ports are inputs and which are outputs. Continuing on the previously established relationships:

- $out1 = in1 \oplus 0$
- $out2 = in2 \oplus 1$
- $out3 = in3 \oplus 0$
- $out4 = in4 \oplus 1$

The program reads the bytes in the flag, and handles them 4 bits at a time through this electrical scheme. In other words, both the nibbles of the given bytes are XORed by  $0b0101$ . We can reverse this using a straightforward script:

```
given_file = open("output.txt", "r").read().splitlines()

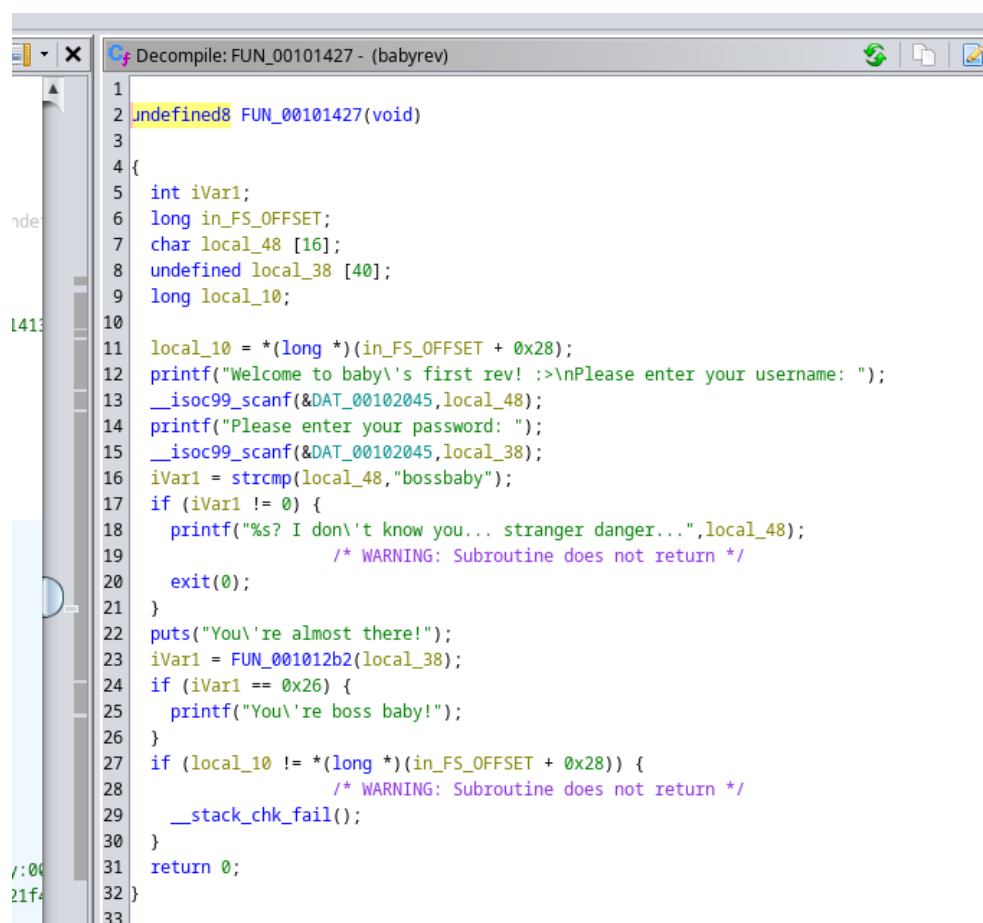
for line in given_file:
    part1 = int(line[0:4], 2) ^ 0b0101
    part2 = int(line[4:8], 2) ^ 0b0101
    result = (part1 << 4) + part2
    print(chr(result), end='')
```

The flag is `flag{a16b8027cf374b115f7c3e2f622d84bc}`.

## 4 Reverse Engineering

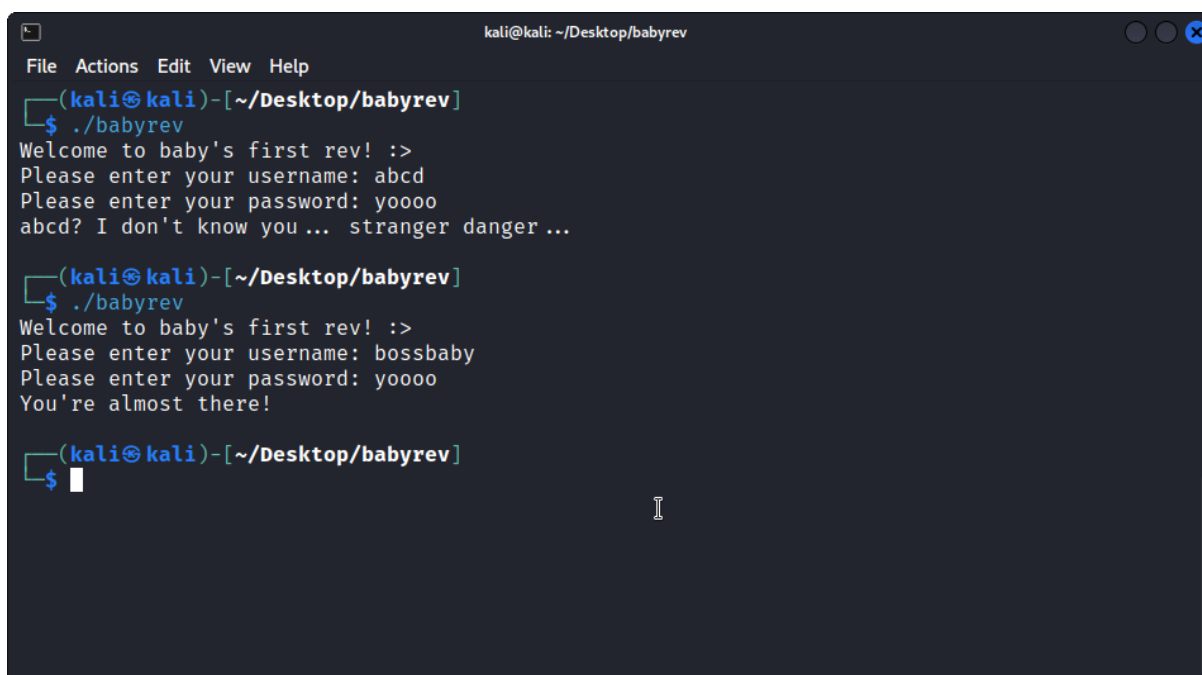
### 4.1 Babyrev

After establishing that the given file is a standard executable, I opened it in the Ghidra reverse engineering tool. The main function is the following.



```
1
2 undefined8 FUN_00101427(void)
3
4 {
5     int iVar1;
6     long in_FS_OFFSET;
7     char local_48 [16];
8     undefined local_38 [40];
9     long local_10;
10
11     local_10 = *(long *)(in_FS_OFFSET + 0x28);
12     printf("Welcome to baby's first rev! :>\nPlease enter your username: ");
13     __isoc99_scanf(&DAT_00102045,local_48);
14     printf("Please enter your password: ");
15     __isoc99_scanf(&DAT_00102045,local_38);
16     iVar1 = strcmp(local_48,"bossbaby");
17     if (iVar1 != 0) {
18         printf("%s? I don't know you... stranger danger...",local_48);
19         /* WARNING: Subroutine does not return */
20         exit(0);
21     }
22     puts("You're almost there!");
23     iVar1 = FUN_001012b2(local_38);
24     if (iVar1 == 0x26) {
25         printf("You're boss baby!");
26     }
27     if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
28         /* WARNING: Subroutine does not return */
29         __stack_chk_fail();
30     }
31     return 0;
32 }
33
```

It appears to check for a username first, which must be equal to "bossbaby", and then do some kind of procedure to check the password. The procedure that checks the password must return 0x26. I figured this password must be the flag, as those have always been of length 38. There does not seem to be anything dangerous in the file, so I executed it on my VM to confirm my first suspicions:



```
kali@kali: ~/Desktop/babyrev
File Actions Edit View Help
(kali@kali)~[~/Desktop/babyrev]
$ ./babyrev
Welcome to baby's first rev! :>
Please enter your username: abcd
Please enter your password: yoooo
abcd? I don't know you... stranger danger...

(kali@kali)~[~/Desktop/babyrev]
$ ./babyrev
Welcome to baby's first rev! :>
Please enter your username: bossbaby
Please enter your password: yoooo
You're almost there!

(kali@kali)~[~/Desktop/babyrev]
$
```

Now, investigating the procedure that checks the password, the following lines seem essential. It is a length indeed,

but the right length is only returned if some encoding of the password matches data hard coded at &DAT\_00104020. The alignment seems to be four bits, which was also confirmed with a manual look at the data there in memory.

```

44  *(undefined8 *)((long)psVar5 + iVar1 + -8) = 0x1013fb;
45  sVar4 = strlen(pcVar2);
46  if (sVar4 <= uVar3) break;
47  if (*(int *)&DAT_00104020 + (long)i * 4 == *(int *)(local_38 + (long)i * 4)) {
48      length_of_password = length_of_password + 1;
49  }
50  i = i + 1;
51  }
52  if (local_30 != *(long *)(in_FS_OFFSET + 0x28)) {
53      /* WARNING: Subroutine does not return */

```

The procedure which generates this encoding (local\_38) is the following. It seems like a relatively simple calculation, which we can easily match in a Python script.

```

C: Decompiler: encode_password - (babyrev)
1
2 long encode_password(char *given_password, long addr_encrypted_pw)
3
4 {
5     size_t sVar1;
6     int local_1c;
7
8     local_1c = 0;
9     while( true ) {
10        sVar1 = strlen(given_password);
11        if (sVar1 <= (ulong)(long)local_1c) break;
12        *(int *)(addr_encrypted_pw + (long)local_1c * 4) =
13            local_1c * local_1c +
14            ((int)given_password[local_1c] << ((char)local_1c + (char)(local_1c / 7) * -7 & 0x1f));
15        local_1c = local_1c + 1;
16    }
17    return addr_encrypted_pw;
18 }
19

```

Copying both the data we found and the calculation itself into Python, and creating a loop to try values of the flag byte-per-byte, we get the following. This script successfully generates the flag.

```

raw_data = b'\x66\x00\x00\x00\xd9\x00\x00\x00\x88\x01\x00\x00\x41\x03\x00\x00\x
→ xc0\x07\x00\x00\xf9\x06\x00\x00\xa4\x18\x00\x00\x95\x00\x00\x00\x0a\x01\x
→ x00\x00\xd5\x01\x00\x00x7c\x03\x00\x00\xa9\x03\x00\x00\xb0\x07\x00\x00\x
→ x69\x19\x00\x00x27\x01\x00\x00\xa3\x01\x00\x00xc4\x01\x00\x00\xb9\x02\x
→ x00\x00x54\x07\x00\x00x89\x08\x00\x00x50\x0f\x00\x00xf0\x01\x00\x00\x
→ x54\x02\x00\x00xd9\x02\x00\x00x58\x05\x00\x00x71\x05\x00\x00x24\x09\x
→ x00\x00x19\x10\x00\x00x42\x03\x00\x00xad\x03\x00\x00x08\x05\x00\x00\x
→ xe9\x06\x00\x00x30\x0a\x00\x00xe1\x10\x00\x00x84\x12\x00\x00\x00\x05\x
→ x00\x00xd2\x05\x00\x00x4d\x07\x00\x00'

flag = b""

for i in range(38):
    for c in range(256):
        calculated_value = i*i + (c << (i + (i // 7) * (-7) & 0x1f))
        encrypted_value = int.from_bytes(raw_data[i*4 : (i+1)*4], "little")
        if calculated_value == encrypted_value:
            flag += bytes([c])

print(flag)

```

The flag is `flag{7bdeac39cca13a97782c04522aece87a}`.

## 5 Mobile

### 5.1 Mobilize (50 pts)

I used `apktools` to decode the given APK file, and found the flag within `res/values/strings.xml`.

```
...
<string name="exposed_dropdown_menu_content_description">Show dropdown menu</
  ↳ string>
<string name="fab_transformation_scrim_behavior">com.google.android.material.
  ↳ transformation.FabTransformationScrimBehavior</string>
<string name="fab_transformation_sheet_behavior">com.google.android.material.
  ↳ transformation.FabTransformationSheetBehavior</string>
<string name="flag">flag{e2e7fd4a43e93ea679d38561fa982682}</string>
<string name="hide_bottom_view_on_scroll_behavior">com.google.android.material.
  ↳ behavior.HideBottomViewOnScrollBehavior</string>
<string name="icon_content_description">Dialog Icon</string>
<string name="item_view_role_description">Tab</string>
...
```

An alternative without the need to unpack the APK file could be using `strings` on the APK file:

```
$ strings mobilize.apk | grep flag{
```

I verified, and this would have worked too. The flag is `flag{e2e7fd4a43e93ea679d38561fa982682}`.

## 6 Keeber (OSINT)

### 6.1 Keeber 1

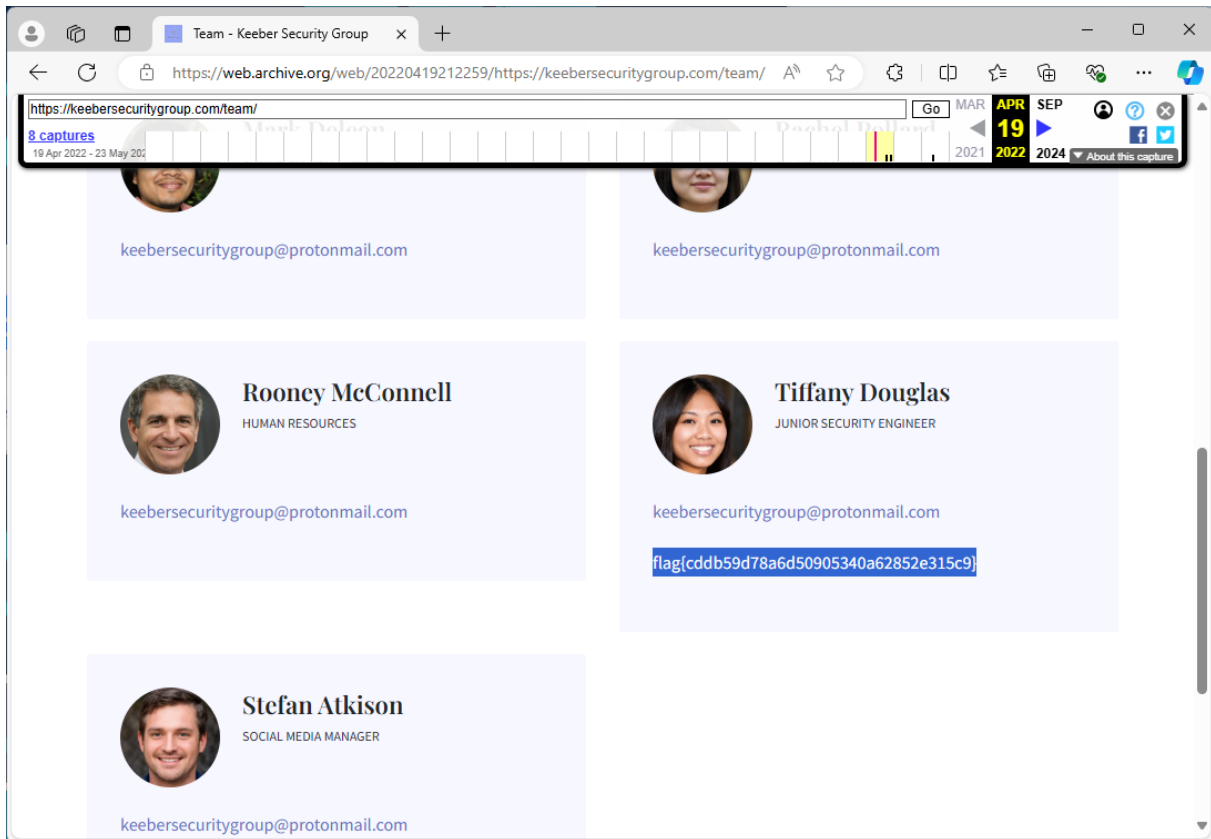
I assumed I was gonna have to investigate `keerbersecuritygroup.com`, it was the only website that showed up on Google that looked somewhat relevant. I used *WHOIS* to find out who registered the domain. The flag showed up in the Registrant Name line:

```
$ whois keerbersecuritygroup.com
  Domain Name: KEEBERSECURITYGROUP.COM
  Registry Domain ID: 2689392646_DOMAIN_COM-VRSN
  Registrar WHOIS Server: whois.name.com
  Registrar URL: http://www.name.com
  Updated Date: 2022-04-15T01:52:49Z
...
  Registry Registrant ID: Not Available From Registry
    Registrant Name: flag{ef67b2243b195eba43c7dc797b75d75b} Redacted
    Registrant Organization:
    Registrant Street: 8 Apple Lane
    Registrant City: Standish
    Registrant State/Province: ME
    Registrant Postal Code: 04084
    Registrant Country: US
...
```

The flag is `flag{ef67b2243b195eba43c7dc797b75d75b}`.

## 6.2 Keeber 2

The Wayback Machine is a great way to find ex-employees. In this case, the snapshot made on April 19 reveals one:

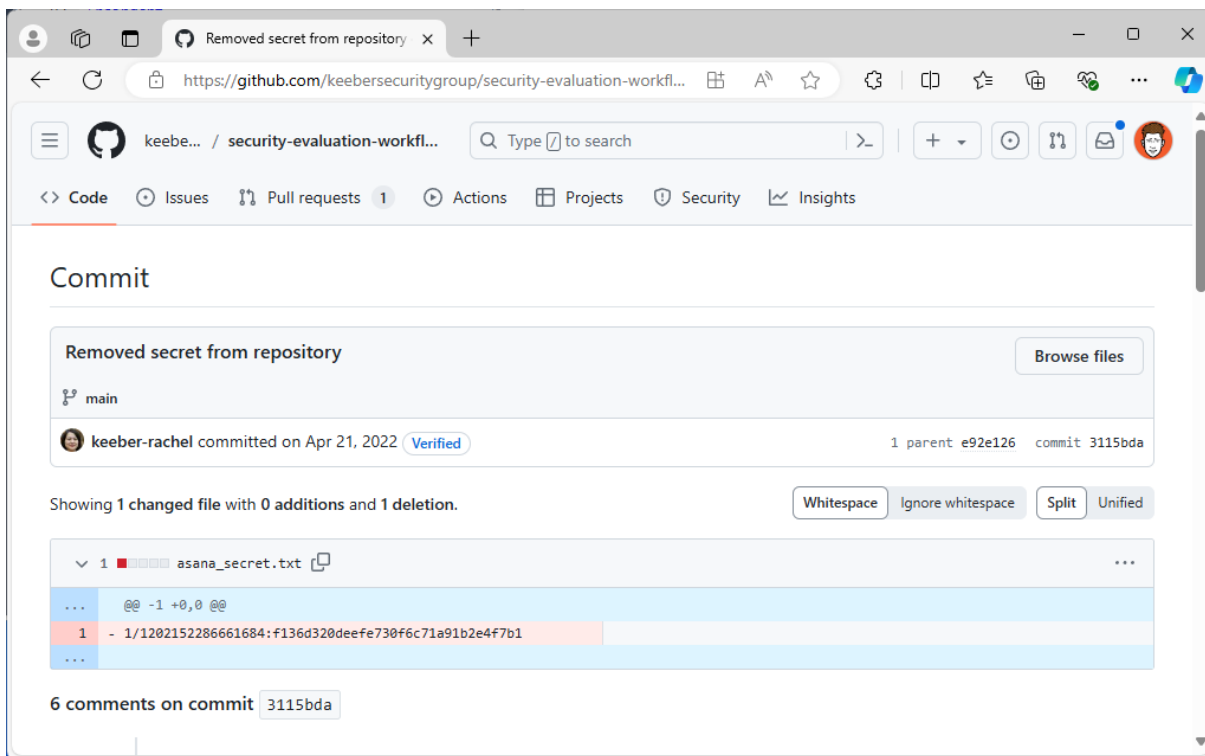


The flag is `flag{cddb59d78a6d50905340a62852e315c9}`.

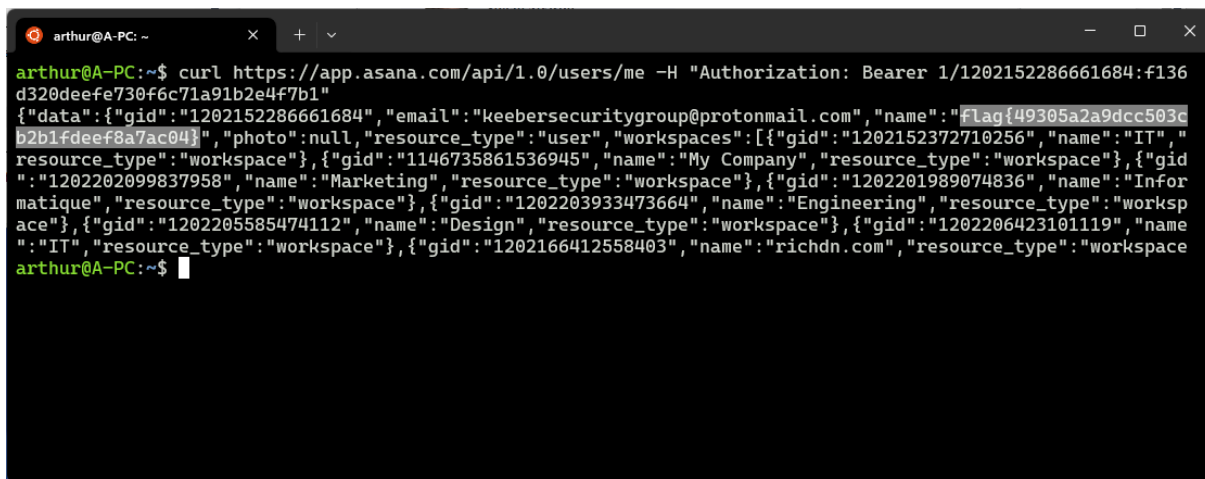
## 6.3 Keeber 3

I figured the Git commit must be one on the `@keebersecuritygroup` repositories on GitHub. The `security-evaluation-workflow` repository has a `.gitignore` file containing a reference to `asana_secret.txt`, and the following commit specifically removing that file:





The Asana platform has a documentation page on personal access tokens<sup>3</sup>, and this looks like one. The page contains an example `curl`-command to access the API, which I tried using the token. The response contained the flag.



The flag is `flag{49305a2a9dcc503cb2b1fdeef8a7ac04}`.

## 6.4 Keeber 5

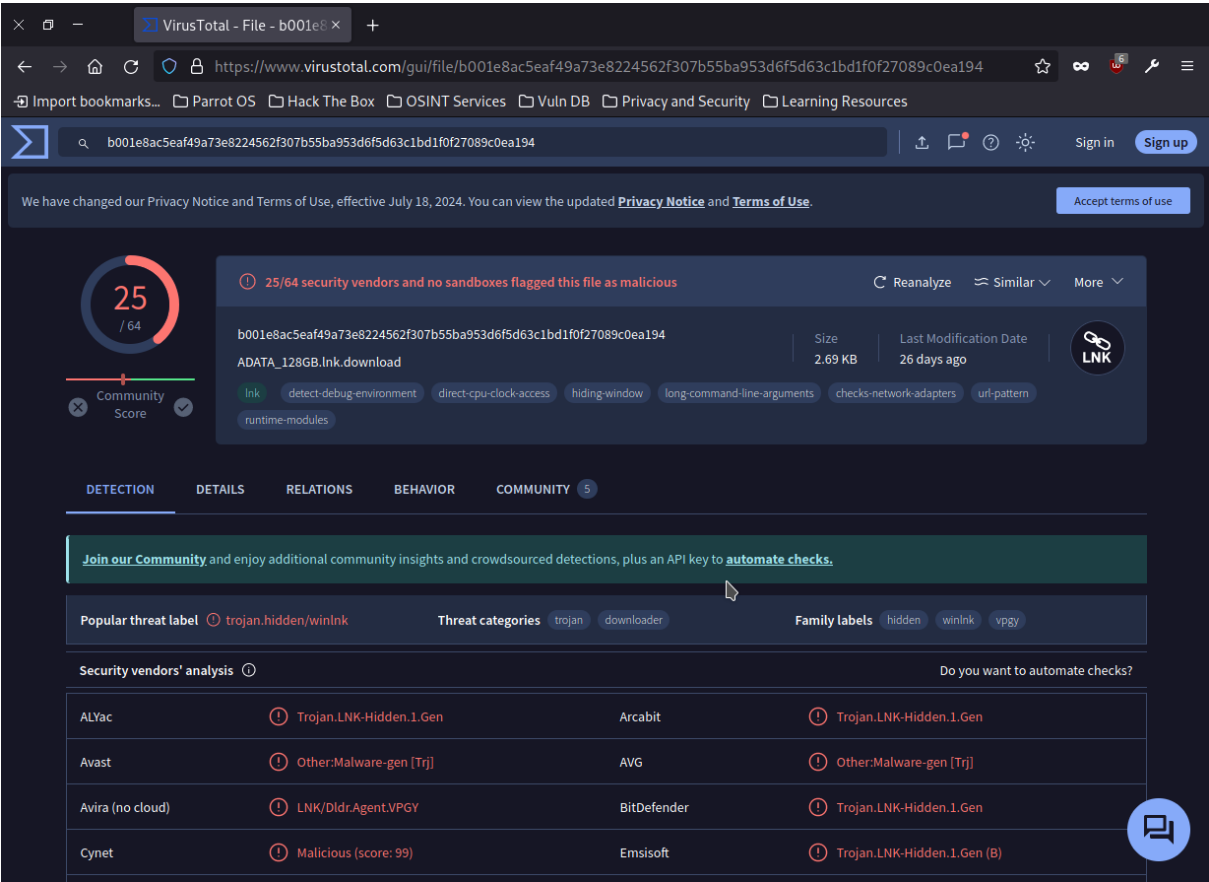
The challenge text tells me the e-mail address I'm looking for is going to be in one of the commit histories. I could have accessed this on GitHub directly, but a search command seemed easier than reading everything myself. I did so using `git log | grep "flag"`, which immediately yielded the flag in the *security-evaluation-workflow* repository. That's `flag{2c90416c24a91a9e1eb18168697e8ff5}`.

<sup>3</sup><https://developers.asana.com/docs/personal-access-token>

# 7 Malware

## 7.1 USB Drive

I first verified whether the given file was known malware. I used VirusTotal for this, which immediately yielded some results:



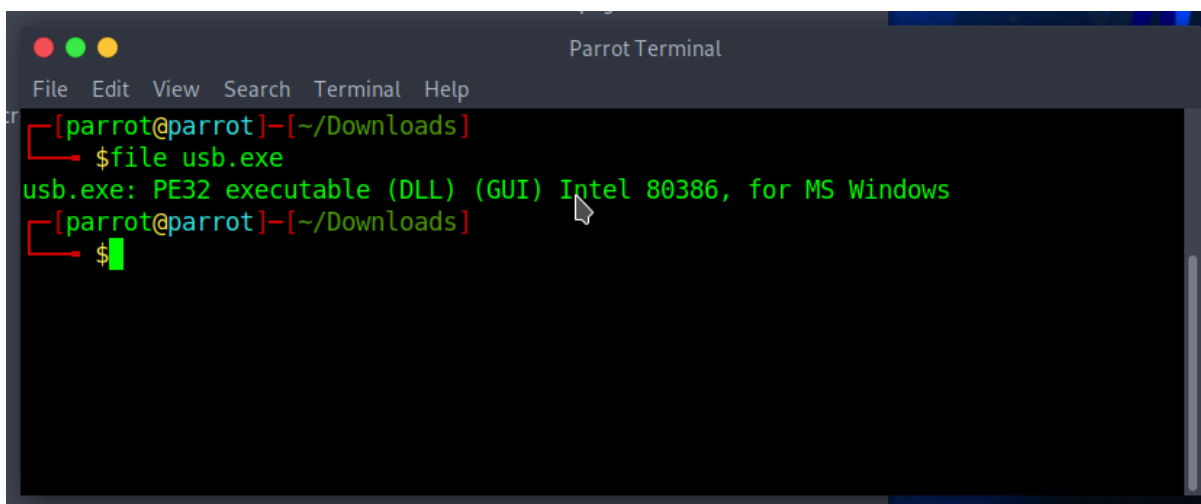
However, from these results, it was hard to check what exactly the file was doing, let alone find a flag. Using the file utility, I found out it's a Windows Shortcut file: this was already suggested by the .lnk extension.

```
Parrot Terminal
File Edit View Search Terminal Help
[parrot@parrot]--[~/Downloads]
$ file ADATA_128GB.lnk
ADATA_128GB.lnk: MS Windows shortcut, Item id list present, Points to a file or
directory, Has Description string, Has command line arguments, Icon number=30, A
rchive, ctime=Sun Nov 21 03:24:03 2010, mtime=Sun Nov 21 03:24:03 2010, atime=Su
n Nov 21 03:24:03 2010, length=302592, window=hiddenormalshowminimized
[parrot@parrot]--[~/Downloads]
$
```

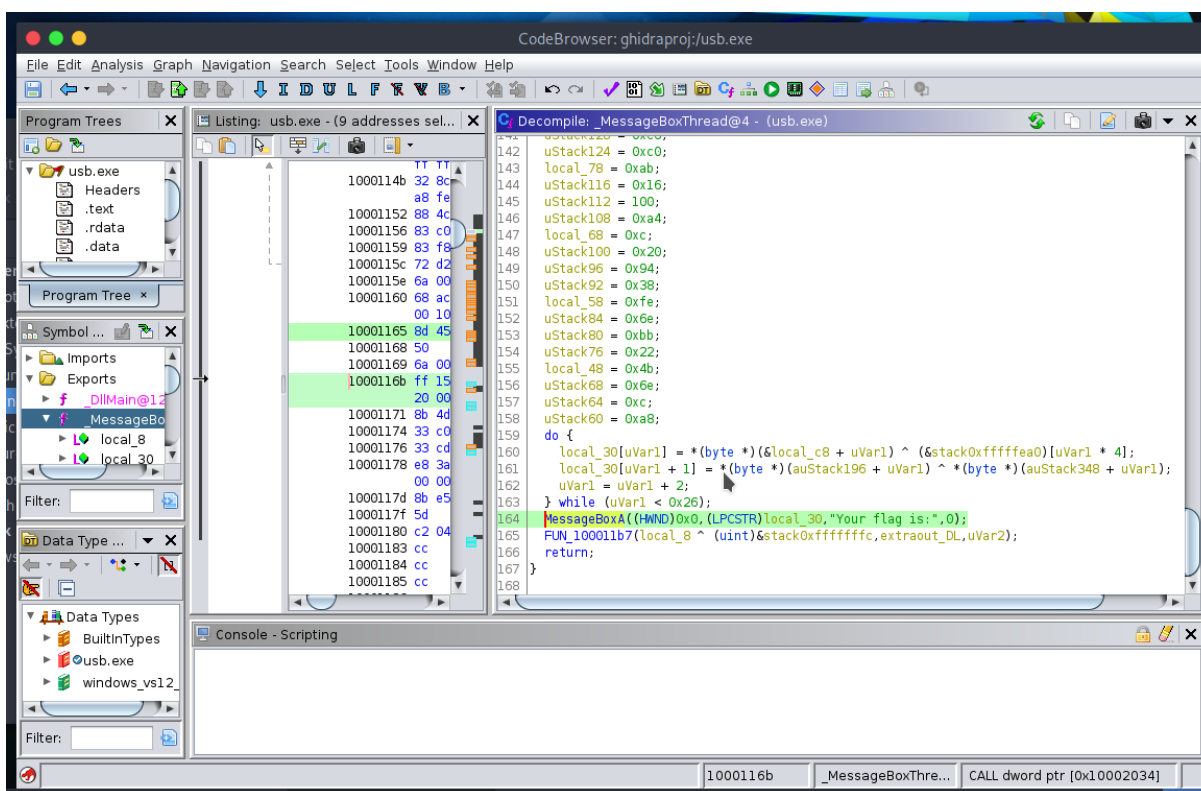
There exists a tool called `lnkinfo` which shows all the information in a `.lnk`-file. Calling it resulted in a lot of whitespace, which was probably an obfuscation attempt, but also a suspicious url: <https://tinyurl.com/a7ba6ma>.

```
Parrot Terminal
File Edit View Search Terminal Help
n
/V/R CMD<https://
Icon location : inyurl.com/a7ba6ma
Link target identifier:
Shell item list
Number of items : 5
Shell item: 1
Item type : Root folder
Class type indicator : 0x1f (Root folder)
Shell folder identifier : 20d04fe0-3aea-1069-a2d8-08002b30309d
Shell folder name : My Computer
```

This url contained plain text, clearly some kind of encoding. My first guess was Base64 (which is popular in CTF challenges), but it turned out to be Base32. Decoding it resulted in an executable:



Which I dropped into the Ghidra reverse engineering tool, to get an insight into what the program does. It appears to print the flag:



The loop with XOR operations on the preceding lines are probably meant to decode the flag first. I could write code to simulate those operations, which would take some time but would definitely work. Another option would be to execute the script: if it really only prints something, that is supposed to be safe. I went with a third option: I used emulation, specifically [speakeasy](https://github.com/mandiant/speakeasy)<sup>4</sup>.

<sup>4</sup><https://github.com/mandiant/speakeasy>

```
kali@kali: ~  
File Actions Edit View Help  
(kali@kali)-[~]  
$ speakeasy -t usb.exe  
* exec: dll_entry.DLL_PROCESS_ATTACH  
0x10001662: 'KERNEL32.GetSystemTimeAsFileTime(0x12fffc8)' → None  
0x10001671: 'KERNEL32.GetCurrentThreadId()' → 0x434  
0x1000167a: 'KERNEL32.GetCurrentProcessId()' → 0x420  
0x10001687: 'KERNEL32.QueryPerformanceCounter(0x12fffc0)' → 0x1  
0x10001c13: 'KERNEL32.IsProcessorFeaturePresent("PF_XMMI64_INSTRUCTIONS_AVAILABLE")' → 0x1  
0x100018cf: 'api-ms-win-crt-runtime-l1-1-0._initialize_onexit_table(0x10003364)' → 0x0  
0x100018de: 'api-ms-win-crt-runtime-l1-1-0._initialize_onexit_table(0x10003370)' → 0x0  
0x100016ed: 'KERNEL32.InitializeSListHead(0x10003340)' → None  
0x10001283: 'api-ms-win-crt-runtime-l1-1-0._initterm_e(0x10002080, 0x10002084)' → 0x0  
0x10001c13: 'KERNEL32.IsProcessorFeaturePresent("PF_XMMI64_INSTRUCTIONS_AVAILABLE")' → 0x1  
0x100012a1: 'api-ms-win-crt-runtime-l1-1-0._initterm(0x10002078, 0x1000207c)' → 0x0  
0x100011ae: 'KERNEL32.CreateThread(0x0, 0x0, 0x10001000, 0x0, 0x0, 0x0)' → 0x220  
* exec: export._DllMain@12  
* exec: export._MessageBoxThread@4  
0x10001171: 'USER32.MessageBoxA(0x0, "flag{0af2873a74cfa957ccb90cef814cfe3d}", "Your flag is :", 0x0)' → 0x2  
* exec: thread  
0x10001171: 'USER32.MessageBoxA(0x0, "flag{0af2873a74cfa957ccb90cef814cfe3d}>|", "Your flag is:", 0x0)' → 0x2  
* Finished emulating  
(kali@kali)-[~]  
$
```

The flag is flag{0af2873a74cfa957ccb90cef814cfe3d}.