Funnel Write-up

Prepared by: amra, C4rm3l0

# Introduction

It is a fairly common practice for developers to release the first version of their products on internal networks for testing and debugging. By doing so, they make sure that any potential security risks are **confined** and can only be accessed by "trusted" internal machines. Moreover, some well known applications, like [Redis](https://redis.io/docs/management/security/) or databases are designed to operate securely only on internal/trusted networks and never get exposed over the Internet.

This is indeed a secure practice, but it is based on the hypothesis that the internal network is uncompromised. If a machine that has access to the internal network gets compromised it is possible to access these instances using tunneling.

The definition of according to the [Wikipedia](https://en.wikipedia.org/wiki/Tunneling_protocol) page is:

tunneling

In computer networks, a tunneling protocol is a communication protocol which allows for the movement of data from one network to another, by exploiting **encapsulation**. It involves allowing private network communications to be sent across a public network (such as the Internet) through a process called encapsulation.

[...]

The tunneling protocol works by using the data **portion** of a packet (the payload) to carry the packets that actually provide the service. Tunneling uses a layered protocol model such as those of the OSI or TCP/IP protocol suite, but usually violates the layering when using the payload to carry a service not normally provided by the network. Typically, the delivery protocol operates at an equal or higher level in the layered model than the payload protocol.

According to the definition of tunneling, one can use it to access resources that are available only to internal networks. To create/facilitate such tunnels, an appropriate application should be used. The most known one is SSH . According to [Wikipedia](https://en.wikipedia.org/wiki/Secure_Shell):

The Secure Shell Protocol (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network. Its most **notable** applications are remote login and command-line execution.

The protocol is vastly used for maintaining and accessing remote systems in a secure and encrypted

SSH

way. But, it also offers the possibility to create tunnels that operate over the SSH protocol. More specifically, offers various types of tunnels. Before we start exploring these types we have to clarify some basics on

SSH

how the SSH protocol works.

First of all, the machine that initiates the connection is called the and the machine that receives the

client

connections is called the server . The client, has to authenticate to the server in order for the connection to succeed. After the connection is initiated, we have a valid SSH session and the client is able to interact with the server via a shell. The main thing to point out here, is that the data that gets transported through this session can be of any type. This is exactly what allows us to create SSH tunnels within an existing valid SSH session.

The first type of tunneling we are going to take a look is called Local port forwarding . When local port forwarding is used, a separate tunnel is created inside the existing valid SSH session that forwards network traffic from a local port on the client's machine over to the remote server's port. Under the **hood**, SSH allocates a socket listener on the client on the given port. When a connection is made to this port, the connection is forwarded over the existing SSH session over to the remote server's port.

The second type of tunneling is called Remote port forwarding , also known as Reverse Tunneling and as one can imagine it is exactly the opposite operation of a Local port forwarding tunnel . Again, after a successful SSH connection, a separate tunnel is created which SSH uses to redirect incoming traffic to the server's port back to the client. Internally, SSH allocates a socket listener on the server on the given port.

When a connection is made to this port, the connection is forwarded over the existing SSH session over to the local client's port.

The third type of tunneling is called Dynamic port forwarding . **The main issue with both local and remote forwarding is that a local and a remote port have to be defined prior to the creation of the tunnel.** To address this issue, one can use dynamic tunneling . Dynamic tunneling, allows the users to specify just one port that will forward the incoming traffic from the client to the server dynamically. The usage of

dynamic tunneling relies upon the the SOCKS5 protocol. The definition of the [Wikipedia](https://en.wikipedia.org/wiki/SOCKS) is the following:

SOCKS

protocol according to

SOCKS is an Internet protocol that exchanges network packets between a client and server through a proxy server. SOCKS5 optionally provides authentication so only authorized users may access a server. Practically, a SOCKS server proxies TCP connections to an arbitrary IP address, **and provides a means for UDP packets to be forwarded.**

So, what is happenning internaly is that SSH turns into a proxy that proxies connections from the

SOCKS5

client through the server. Tunneling can be a tricky topic to wrap your head around, which is why a hands- on approach like this Box is especially useful in understanding the concept and applying it in future scenarios.

Now that we have covered the basics of tunneling, let's see how it solves real life problems that may occur. Suppose that you are working remotely, and you want to access a database that is only available on your company's internal network. To make the example more specific, let's say you wanted to access a PostgreSQL database that is is often used by businesses and organizations to store, manage, and retrieve data that is critical to their operations. PostgreSQL, also known as Postgres, is a powerful and open-source relational database management system (RDBMS). It is widely used for managing and storing large amounts of data due to its reliability, flexibility, and performance. Without tunneling, you would not be able to access these resources directly. However, by using tunneling, you can create a secure connection between your local machine and the internal network, allowing you to access the internal services as if you were on the network itself. This can be particularly useful for remote employees who need to access internal resources

but do not have direct access to the company's network.

# Enumeration

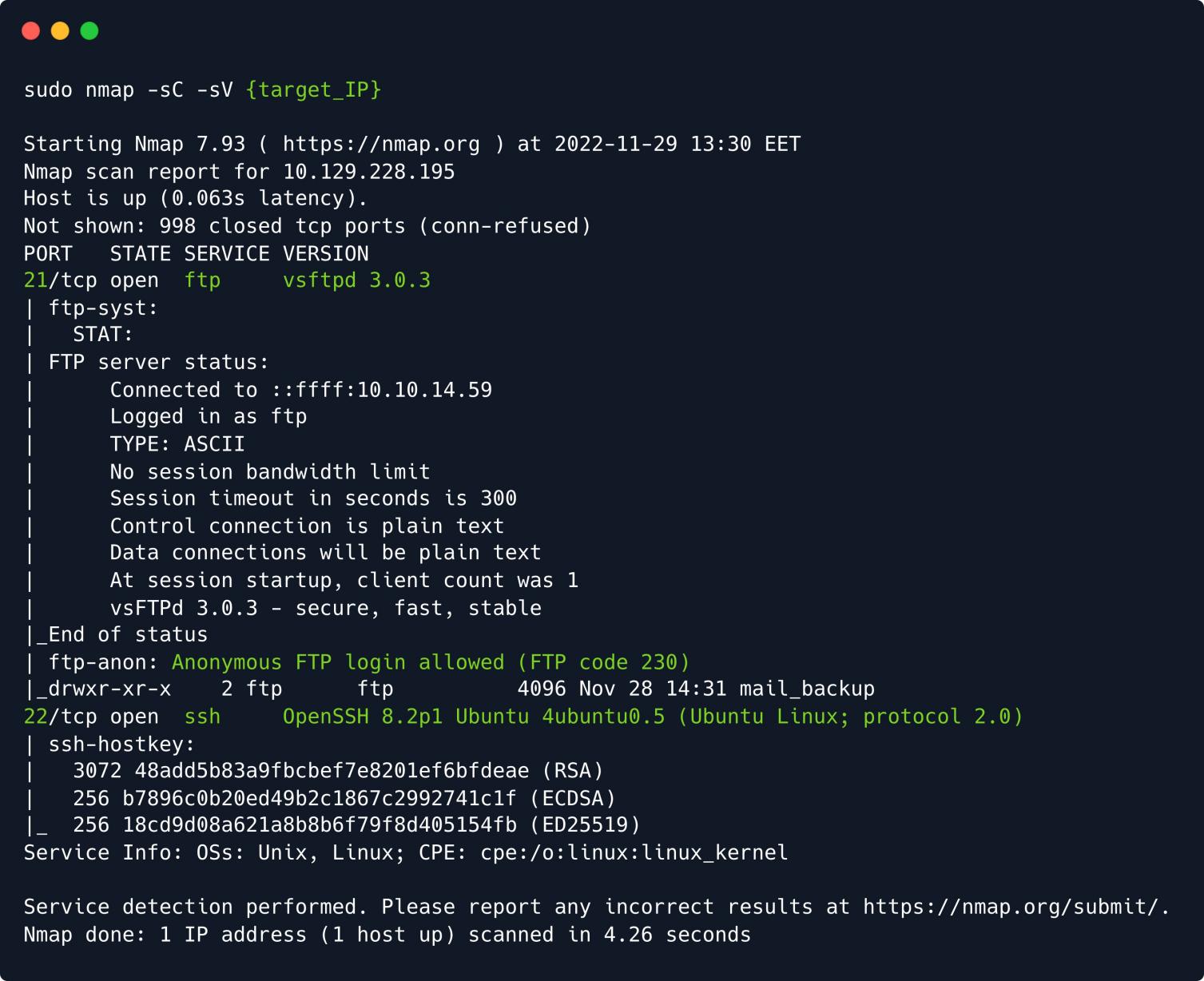
Starting with the nmap scan, we can check what ports are open and what services are running on them:

-sC: Performs a script scan using the default set of scripts. It is equivalent to --

script=default. Some of the scripts in this category are considered intrusive and should not be run against a target network without permission.

-sV: Enables version detection, which will detect what versions are running on what

port.



We find two open ports, namely port 21 , running a service called vsftpd 3.0.3 , and port 22 , running OpenSSH . The former is a service for the File Transfer Protocol - FTP , which is designed to upload, download, and transfer files from one location to another between computer systems.

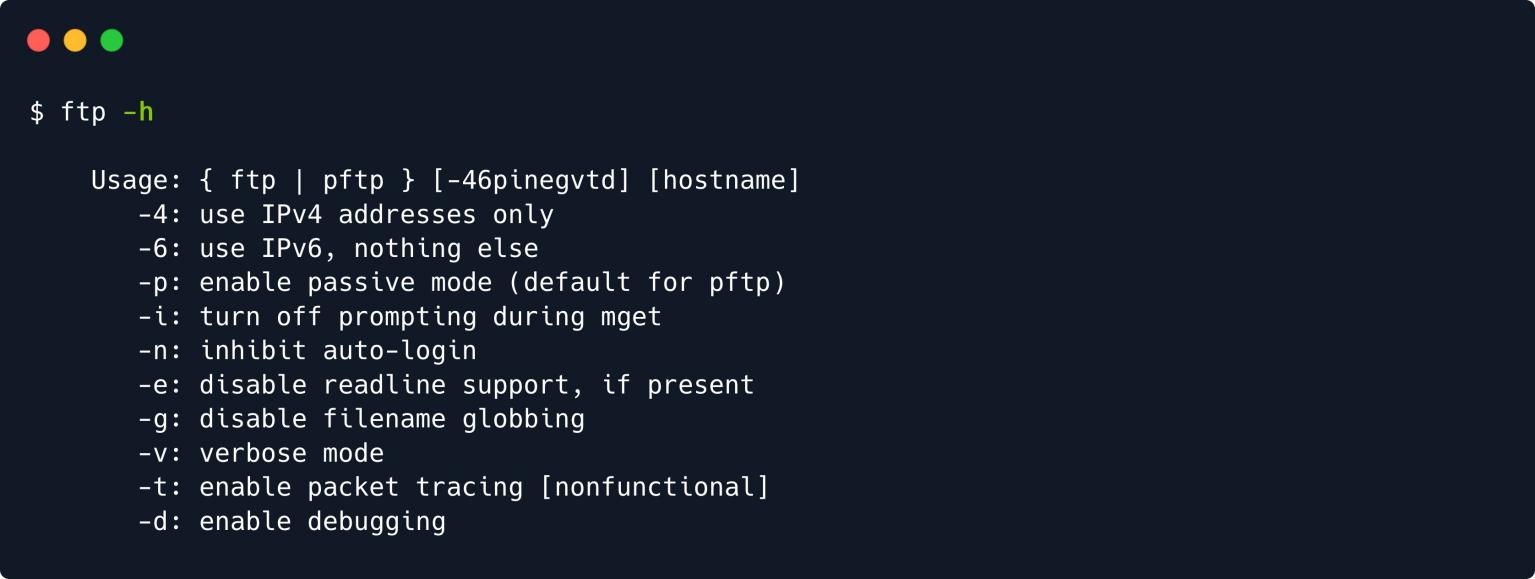
Users could connect to the FTP server anonymously if the server is configured to allow it, meaning that we could use it even if we had no valid credentials. If we look back at our nmap scan result, the FTP server is indeed configured to allow anonymous login:

ftp-anon: Anonymous FTP login allowed (FTP code 230)

If you need a **refresher**, the service on your local host.

ftp -h

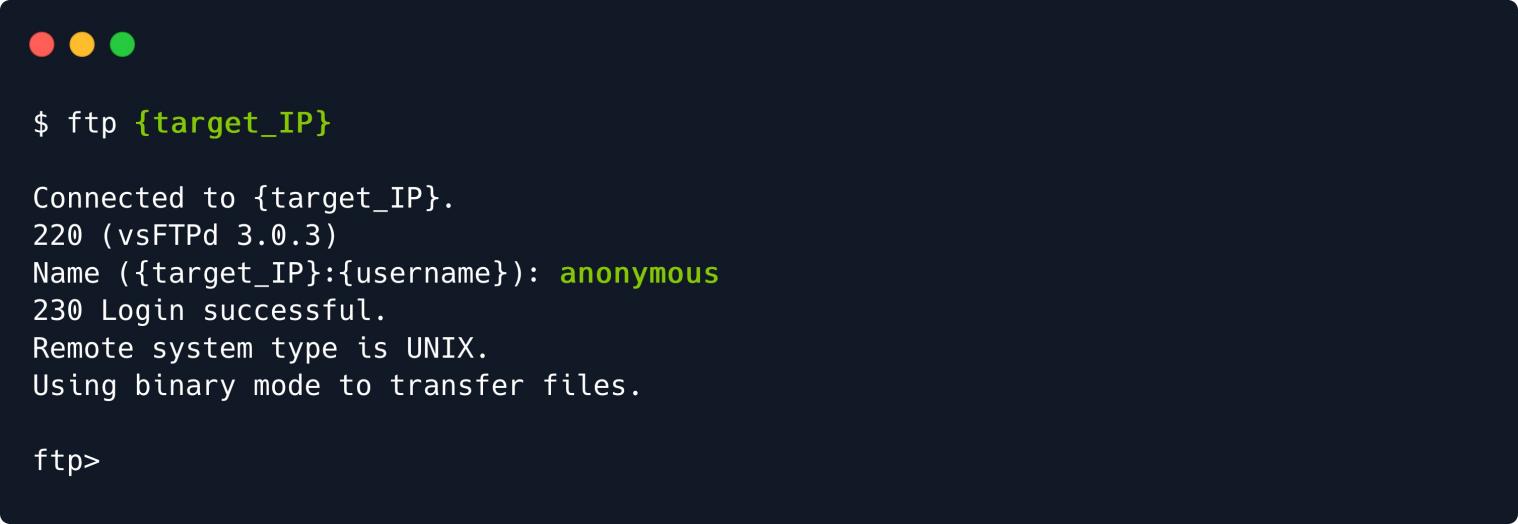
command will help you figure out the available commands for the FTP



To connect to the remote FTP server, you need to specify the target's IP address (or hostname), as displayed on the Starting Point lab page. The prompt will then ask us for our login credentials, which is where we can fill in the anonymous username. In our case, the FTP server does not request a password, and inputting the

anonymous

username proves enough for us to receive the 230 code, Login successful .



Once logged in, you can type the command to check the available commands.

help



We will use

and

to list the directories and download the files stored on the FTP server. With the

command, we can check the contents of our current directory on the remote host, and find a directory called mail\_backup .

dir

get

dir



We can use



cd

dir

to navigate inside that directory, and

once more to list its contents.

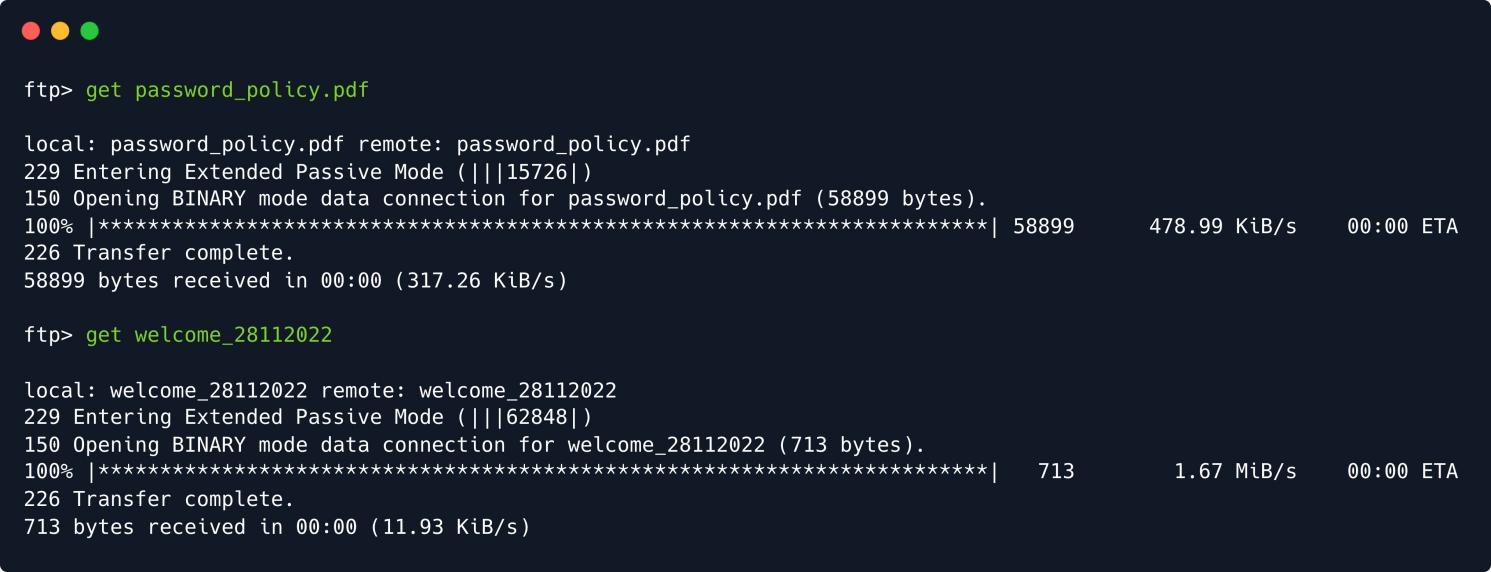


The directory listing shows that two files exist inside this folder. Both files can easily be downloaded using

the command. The FTP service will report the download status completion back to you during this

get

phase. It should not take long to have them both sitting snuggly on your attacking VM.



Termination of the FTP connection can be done by using the terminal tab to its' previous state.

command. This will return the current

exit

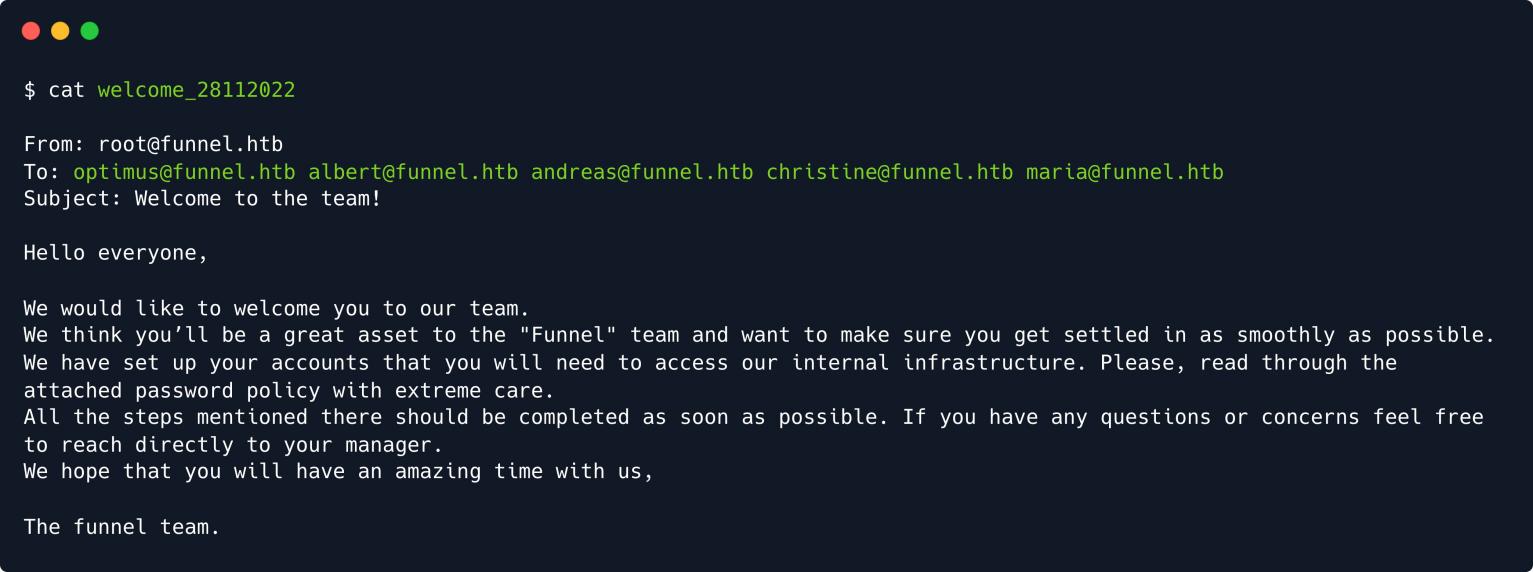


Immediately after exiting the FTP service shell, we can type in the ls command to check if our files are

cat

present in the directory we were last positioned in. We can use the filename, to read one of the files.

command, followed by the



The file appears to be an email, sent to various employees of the *Funnel* company,

welcome\_28112022

instructing them to read the attached document, presumably the other file we downloaded, and go through the steps mentioned there to gain access to their internal infrastructure. Crucially, we can see all the emails that this message is addressed to, giving us an idea of what usernames we might encounter on the target machine.

Since the other file we downloaded, namely password\_policy.pdf , is a file, we cannot use to

PDF

cat

display it, but will rather view it using the conventional way and open it with whichever document viewer is installed by default on our system. To open the current working directory in a file manager window, we can

use the command, followed by the path to the target directory. The current working directory can be

open

referred to as a single **period** . , meaning we don't have to actually write the full path.

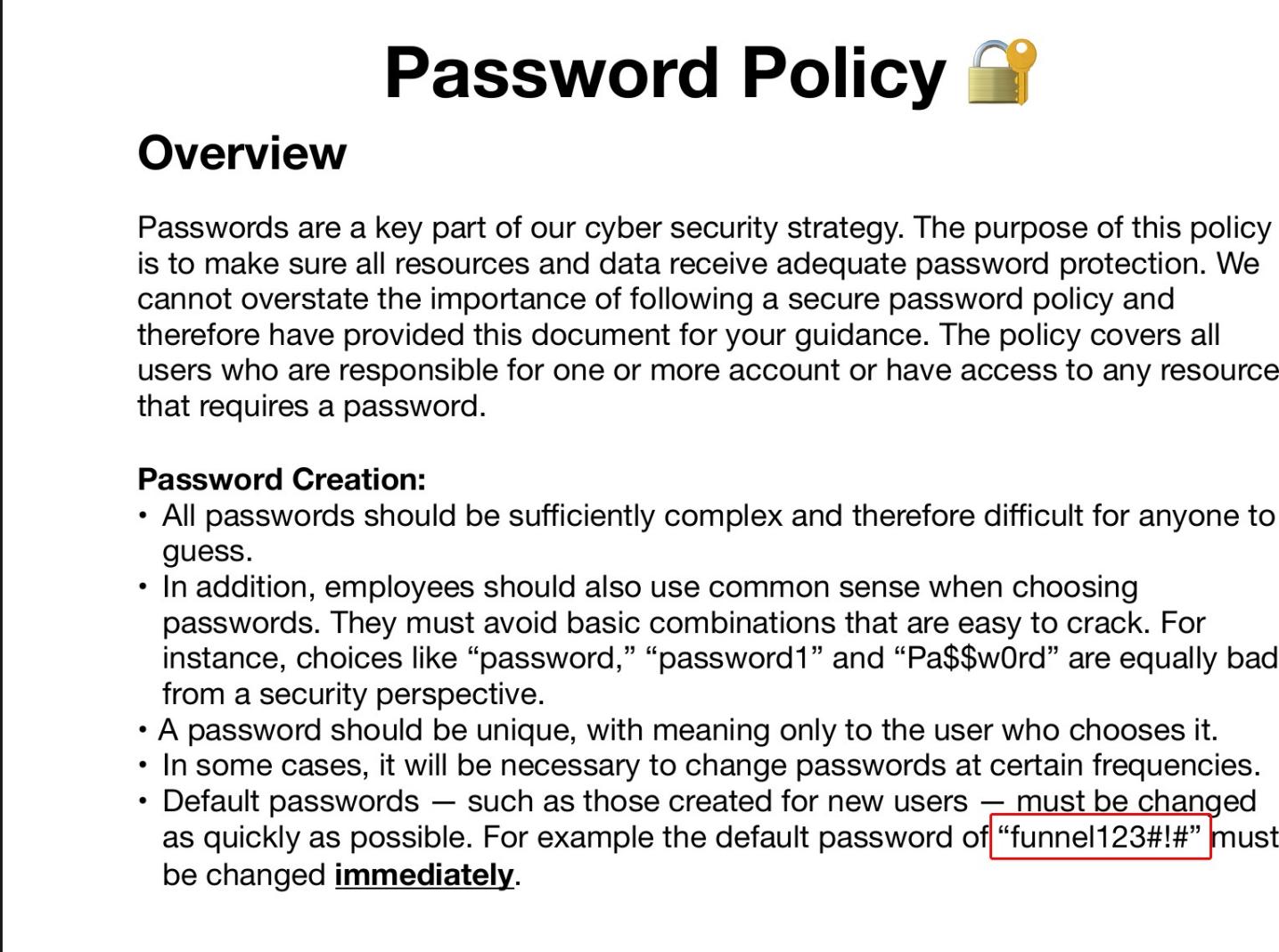
open .

我一直在好奇有什么办法能够方便地以图形化的形式打开当前终端所在目录，今天终于看到了。原来是我失散多年的open命令。

The above command will graphically display the folder, meaning we can now just double-click the and view its contents.

PDF

file



The document appears to be a **memo**, prompting employees to create a secure and complex password for their user accounts. At the end of the document, we can also find a **default** password, namely

funnel123#!# .

# Foothold

Overall, our enumeration yielded a handful of potential usernames, as well as a default password. We also

know that is running on the target machine, meaning we could attempt to bruteforce a username-

SSH

password combination, using the credentials we gathered. This type of attack is also referred to as *password spraying*, and can be automated using a tool such as Hydra .

The password spraying technique involves circumventing common countermeasures against brute-force attacks, such as the locking of the account due to too many attempts, as the same password is sprayed

across many users before another password is attempted. is preinstalled on most penetration-

Hydra

ParrotOS

testing distributions, such as following command.

and Kali Linux , but can also be manually installed using the

sudo apt-get install hydra

In order to conduct our attack, we need to create a list of usernames to try the password against. To do so, we can refer to the email we read earlier, extracting the usernames of all the addresses into a list called usernames.txt , making sure to only include the part **before** @funnel.htb .



Finally, we can now task

with executing the attack on the target machine. Using the

option, we

specify which file contains the list of usernames we will use for the attack. The option specifies that we

Hydra

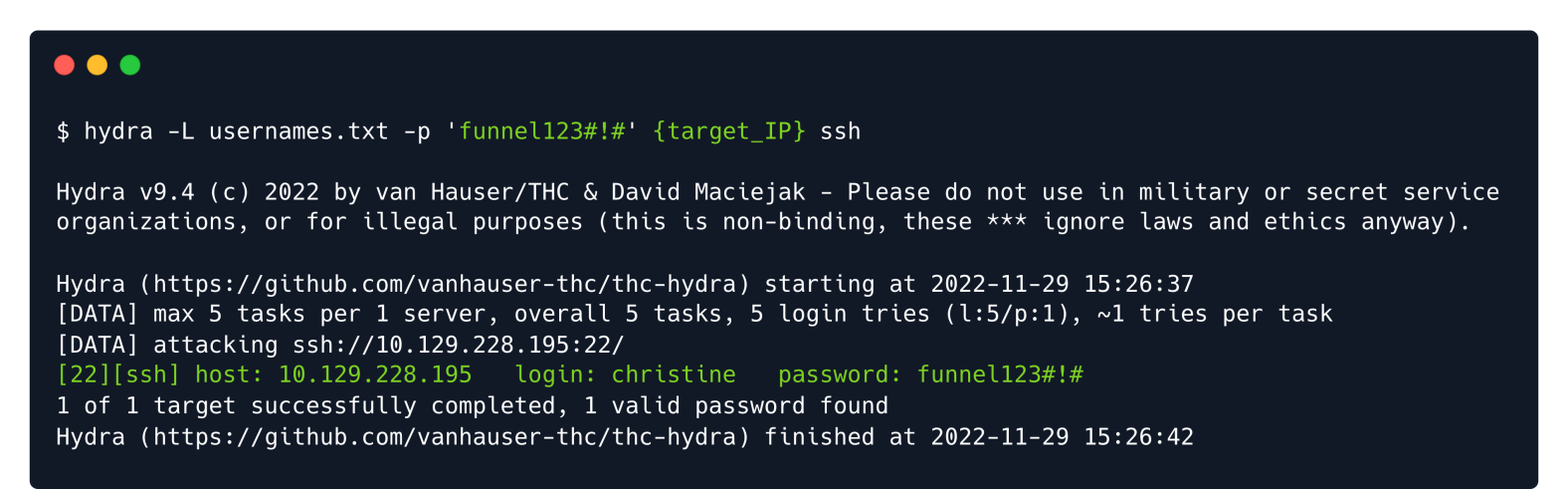


-L



-p

only want to use **one** password, instead of a password list. After the target IP address, we specify the protocol for the attack, which in this case is SSH .



hydra -L usernames.txt -p 'funnel123#!#' {target\_IP} ssh

After just a few seconds hydra gets a valid hit on the combination christine:funnel123#!# . We can now use these credentials to gain remote access to the machine, as the user christine .



## Enumeration

From this point on, we have complete access as the

user on the target machine, and can start

christine

enumerating it for potential files or services that we can explore further. A crucial command at this point in

time is the command, which stands for socket statistics , and can be used to check which ports are



ss

listening locally on a given machine.

-l: Display only listening sockets.

-t: Display TCP sockets.

-n: Do not try to resolve service names.

ss -tln



The output reveals a handful of information; we will analyse it bit-by-bit. The first column indicates the *state*



-l

that the socket is in; since we specified the connection. Moving along horizontally, the

Recv-Q

Send-Q

flag, we will only see sockets that are actively *listening* for a column is not of much concern at this point, it simply

displays the number of queued *received* packets for that given port; does the same but for the

amount of *sent* packets. The crucial column is the fourth, which displays the local address on which a service

listens, as well as its port. is synonymous with localhost , and essentially means that the

127.0.0.1

specified port is **only** listening **locally** on the machine and cannot be accessed externally. This also explains

why we did not discover such ports in our initial scan. On the other hand, the addresses 0.0.0.0 , \* ,

Nmap

and indicate that a port is listening on **all** intefaces, meaning that it is accessible externally, as well as

[::]

FTP

SSH

locally, which is why we were able to detect both the port 22 .



-n

service on port 21 , as well as the

service on

Among these open ports, one particularly sticks out, namely port 5432 . Running flag will show the default service that is presumably running on the respective port.



ss

again **without** the



In this case, the default service that runs on

TCP

port

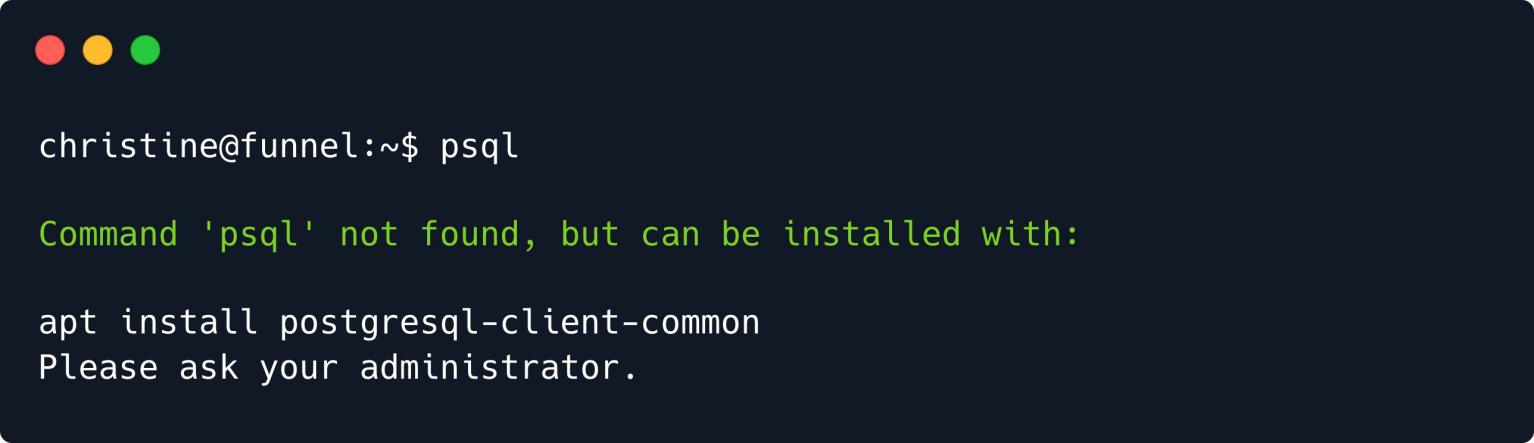
is PostgreSQL , which is a database

management system: creating, modifying, and updating databases, changing and adding data, and more. can typically be interacted with using a command-line tool called psql , however, attempting

5432

PostgreSQL

to run this command on the target machine shows that the tool is not installed.



Seeing as we do not have administrative privileges, we now find ourselves at a bit of a crossroad. The service which most likely has the flag is hidden locally on the target machine, and the tool to access that service is not installed. While there are some potential **workarounds（解决办法）** involving uploading static binaries onto the target machine, an easier way to bypass this roadblock is by a practice called *port-forwarding*, or *tunneling*, using SSH .

## Tunneling

While the theory surrounding tunneling has been broadly covered in the introduction of this document, we will now dive into the praxis; it is now time to get our hands dirty and start digging.

As stated, there are multiple options to take at this point when it comes to the actual port forwarding, but

we will opt for **local** port forwarding (you can find the version in this document's appendix.)

dynamic

To use local port forwarding with SSH , you can use the

ssh

command with the

option, followed by the

local port, remote host and port, and the remote server. For example, the following command will



-L

SSH

1234

localhost

forward traffic from the local port on port 22 :

to the remote server remote.example.com 's

interface

ssh -L 1234:localhost:22 [user@remote.example.com](mailto:user@remote.example.com)

When you run this command, the

SSH

client will establish a secure connection to the remote

server,

and it will **listen** for incoming connections on the **local** port 1234 . When a client connects to the **local** port,

SSH

the client will **forward** the connection to the **remote** server on port 22 . This allows the **local** client to

SSH

access services on the **remote** server as if they were running on the **local** machine.

也就是说，我现在访问本地的1234端口，这个流量会被转发到远程服务器的22端口。

In the scenario we are currently facing, we want to forward traffic from **any** given local port, for instance

1234 , to the port on which PostgreSQL is listening, namely 5432 , on the remote server. We therefore

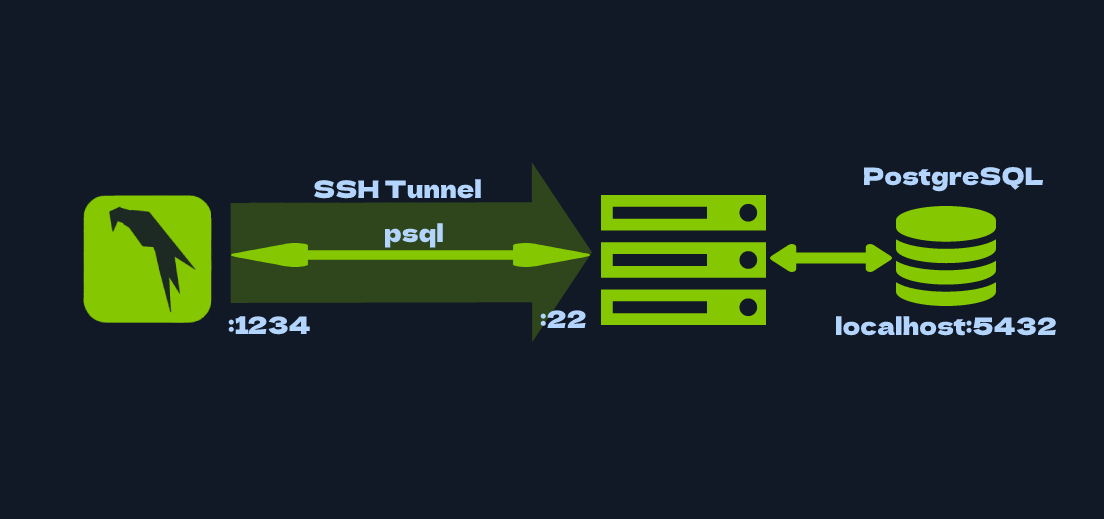
1234

5432

specify port

to the **left** of localhost , and

to the **right**, indicating the target port.



ssh -L 1234:localhost:5432 christine@{target\_IP}



As a side-note, we may elect to just establish a tunnel to the target, without actually opening a full-on



-f



-N

shell on the target system. To do so, we can use the

and

flags, which a) send the command to

the shell's background right before executing it remotely, and b) tells commands remotely.

SSH

not to execute any

After entering christine 's password, we can see that we have a shell on the target system once more,

however, under its hood, has opened up a socket on our local machine on port 1234 , to which we can

SSH

now direct traffic that we want forwarded to port on the target machine. We can see this new socket

5432

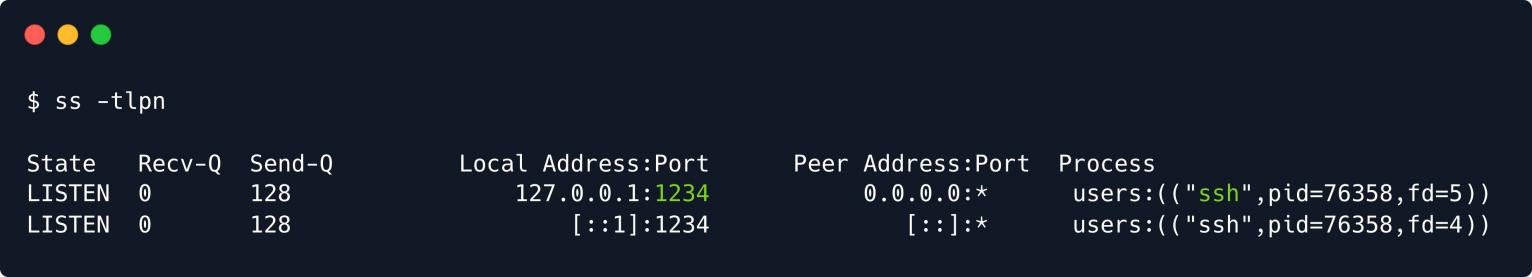
by running again, but this time on our local machine, using a different shell than the one we used to



ss

establish the tunnel.

ss -tlpn



In order to interact with the remote service, we must first install locally on our system. This can be

psql

done easily using the default package manager (on most pentesting distros), apt .

sudo apt update && sudo apt install psql

Using our installation of psql , we can now interact with the service running **locally** on the

PostgreSQL

localhost



-h

target machine. We make sure to specify

1234



-p

using the

option, as we are targeting the tunnel we

created earlier with SSH , as well as port on.

with the

option, which is the port the tunnel is listening

psql -U christine -h localhost -p 1234



Once again, we are prompted for a password, which turns out to be the default password funnel123#!# .

PostgreSQL

We have successfully tunnelled ourselves through to the remote with the various databases and tables on the system.

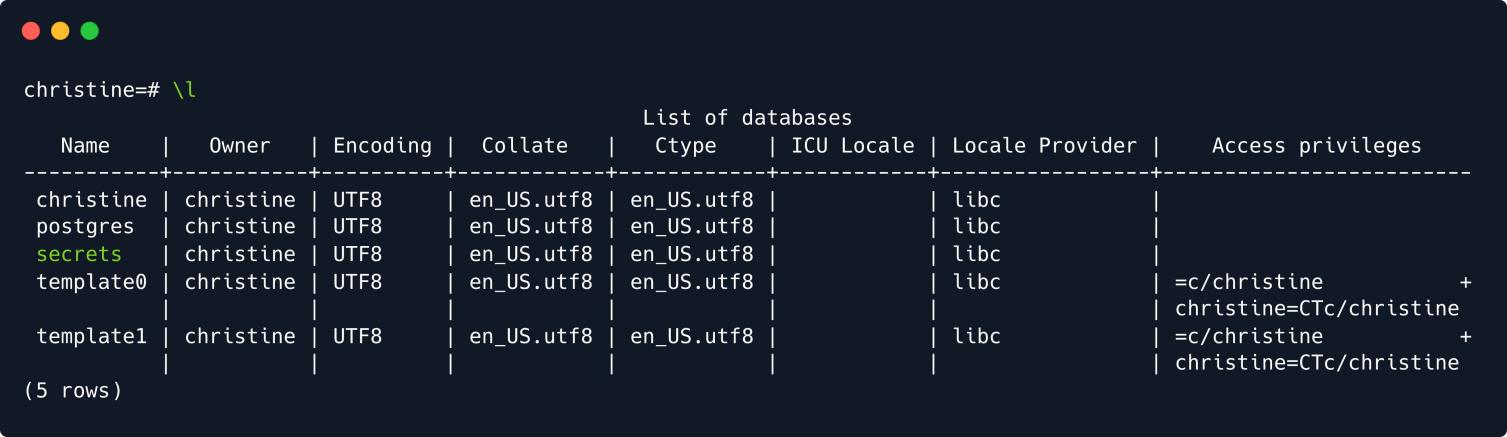
service, and can now interact

In order to list the existing databases, we can execute the command, short for \list .



\l

\l



Five rows are returned, including a database with the **ominous** name secrets . Using the short for \connect , we can select a database and proceed to interact with its tables.



\c

command,

\c secrets



Finally, we can list the database's tables using the command, and dump its contents using the

\dt

conventional SQL query.

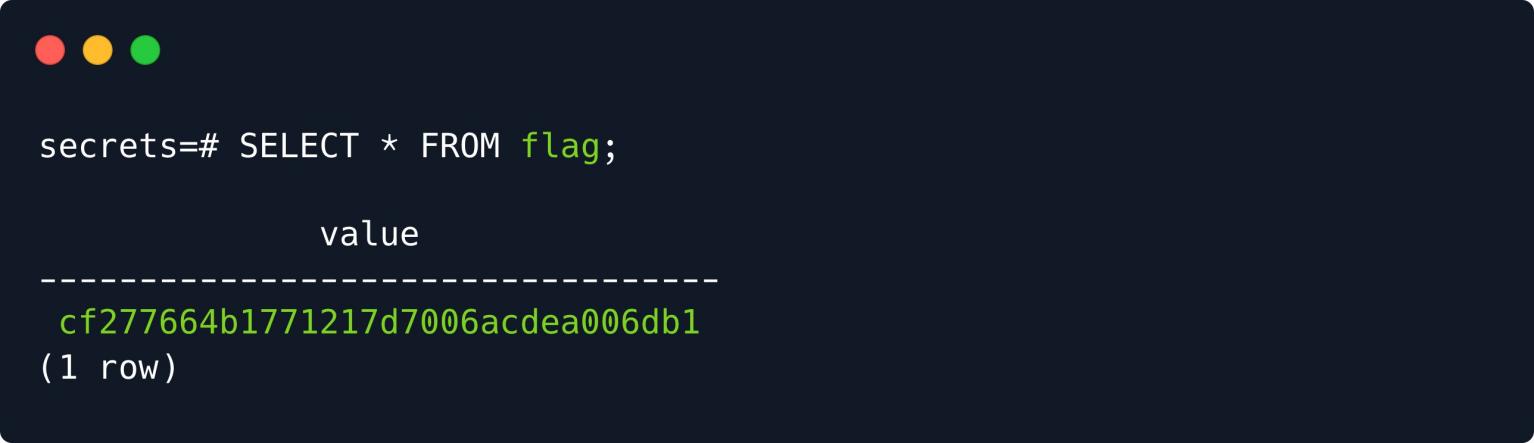
SELECT

\dt

Dt应该是用来描述这个数据库的模式的？Name列应该是表名



SELECT \* FROM flag;



With the collection of the sought flag, this target can be wrapped up. Congratulations!

# Appendix

## Dynamic Port Forwarding

Instead of *local* port forwarding, we could have also opted for *dynamic* port forwarding, again using SSH . Unlike local port forwarding and remote port forwarding, which use a **specific** local and remote port (earlier

we used and 5432 , for instance), dynamic port forwarding uses a **single** local port and **dynamically**

1234

assigns remote ports for each connection.

To use dynamic port forwarding with SSH, you can use the

command with the

ssh

option, followed by



-D

the local port, the remote host and port, and the remote SSH server. For example, the following command will forward traffic from the local port 1234 to the remote server on port 5432, where the PostgreSQL server is running:

ssh -D 1234 christine@{target\_IP}

Again, we can use the



-f

and

flags so we don't actually

into the box, and can instead

continue using that shell locally.



-N

SSH

As you can see, this time around we specify a **single** local port to which we will direct all the traffic needing

forwarding. If we now try running the same command as before, we will get an error.

psql



That is because this time around we did not specify a **target** port for our traffic to be directed to, meaning is just sending traffic into the established **local** socket on port 1234 , but never reaches the

psql

PostgreSQL

service on the target machine.

To make use of dynamic port forwarding, a tool such as is especially useful. In summary and

proxychains

as the name implies, can be used to tunnel a connection through multiple proxies; a use case

proxychains

for this could be increasing anonymity, as the origin of a connection would be significantly more difficult to trace. In our case, we would only tunnel through **one** such "proxy"; the target machine.

The tool is pre-installed on most pentesting distributions (such as and Kali Linux ) and is highly

ParrotOS

customisable, featuring an array of strategies for tunneling, which can be tampered with in its configuration

file /etc/proxychains4.conf .

The minimal changes that we have to make to the file for to:

proxychains

to work in our current use case is

1. Ensure that commented out)

strict\_chain

random\_chain

socks5

is **not** commented out; ( dynamic\_chain and

should be

1. At the very bottom of the file, under [ProxyList] , we specify the that we used for our tunnel

(or socks4 ) host and port

In our case, it would look something like this, as our tunnel is listening at localhost:1234 .

<SNIP>

[ProxyList]

# add proxy here ... # meanwile

# defaults set to "tor" #socks4 127.0.0.1 9050

socks5 127.0.0.1 1234

Having configured

proxychains

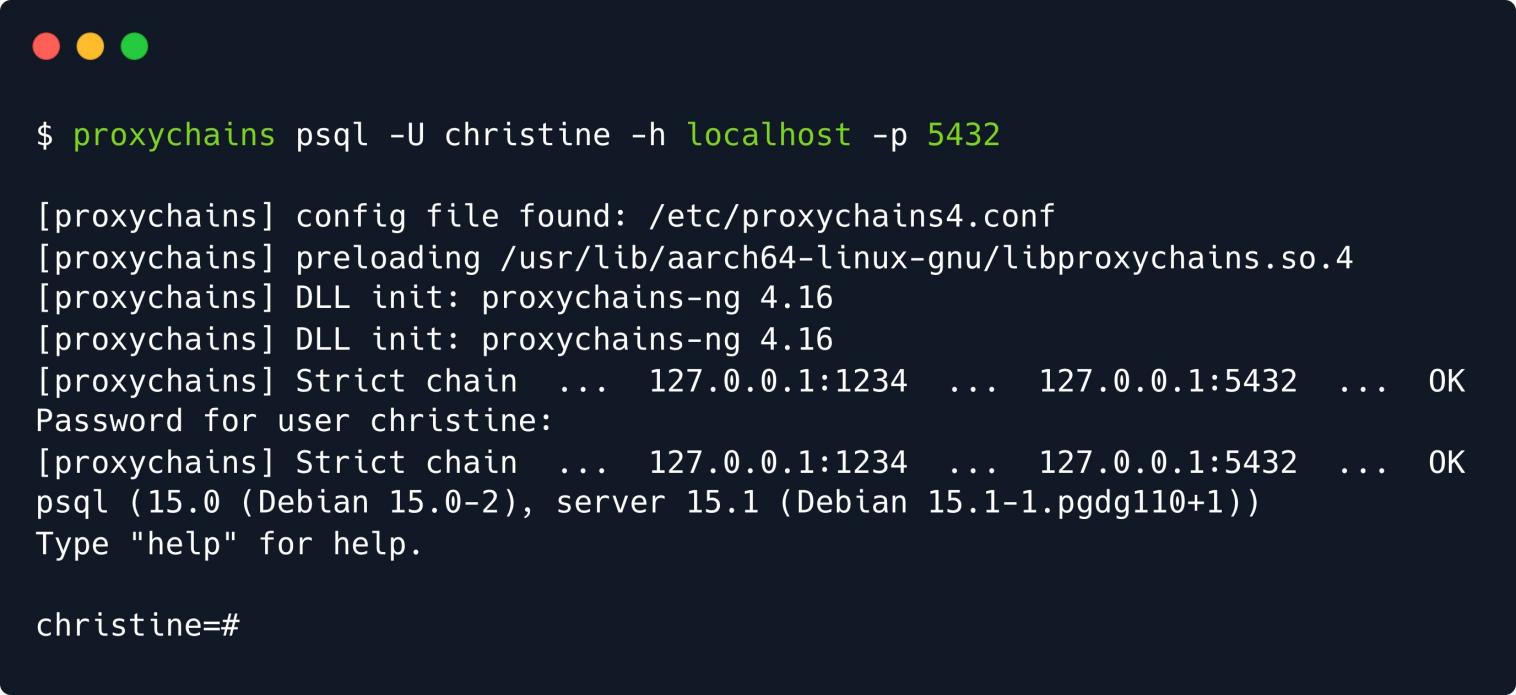
correctly, we can now connect to the

service on the target, as

if we were on the target machine ourselves! This is done by prefixing whatever command we want to run with proxychains , like so:

PostgreSQL

proxychains psql -U christine -h localhost -p 5432



Proxychains can produce an unusual amount of output, but don't be intimidated by it, it is just verbose in showing you whether a certain connection to a proxy worked or not.

This should hopefully demonstrate the beauty of dynamic port forwarding, as we can specify the target port

freely and in accord with each command we want to run. If we wanted to a webserver on port 80 , for

cURL

instance, during local port forwarding we would have to run the tunneling command all over again and

change up the target port. Here, we can simply prefix our command with proxychains , and access

cURL

the webserver as if we were on the target machine ourselves; no need for any extra specification- hence,

**dynamic**.