

# CSCI 476: Computer Security

Network Security: DNS Cache Poisoning (Part 2)

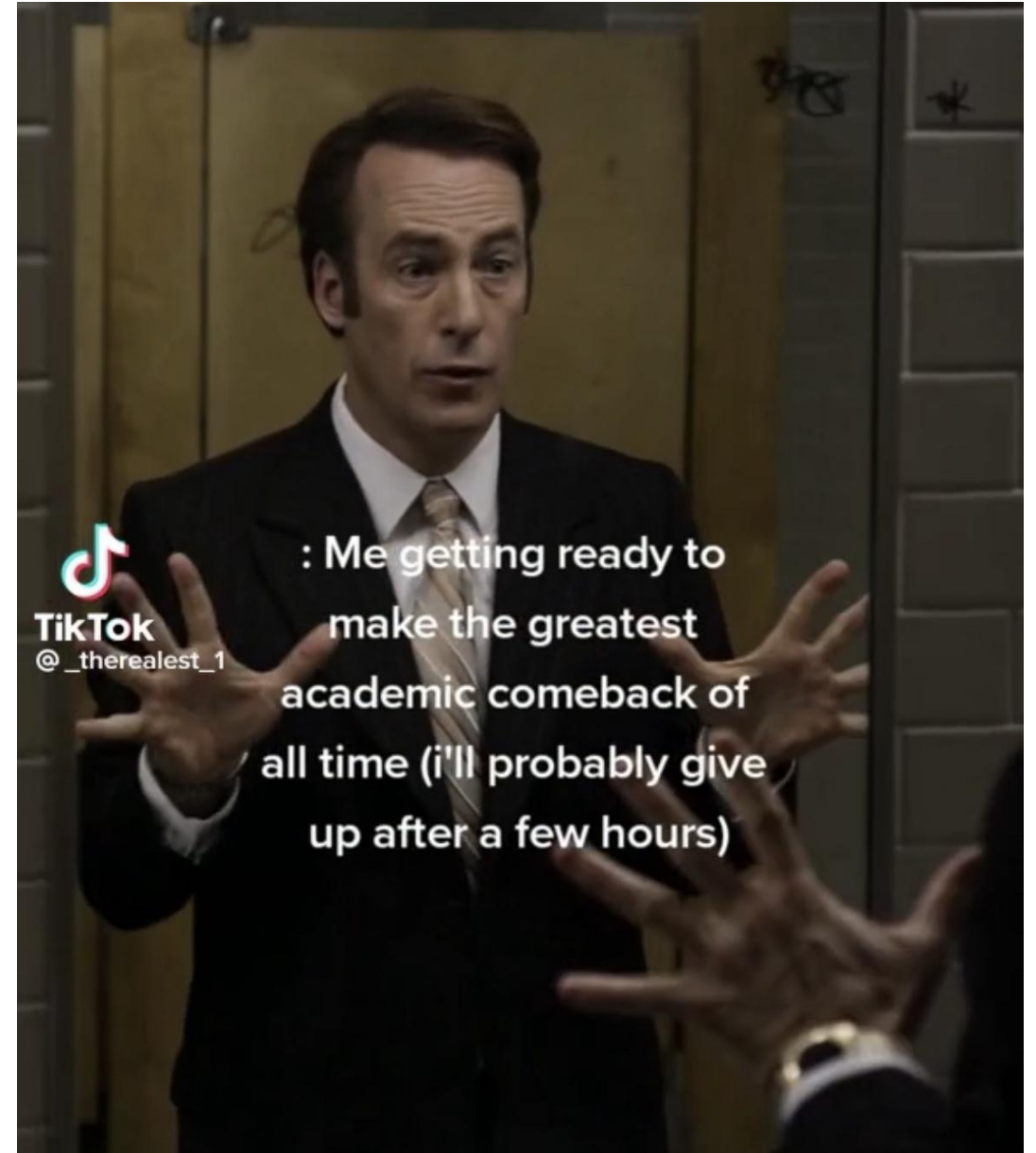
Reese Pearsall  
Spring 2023

## Announcements

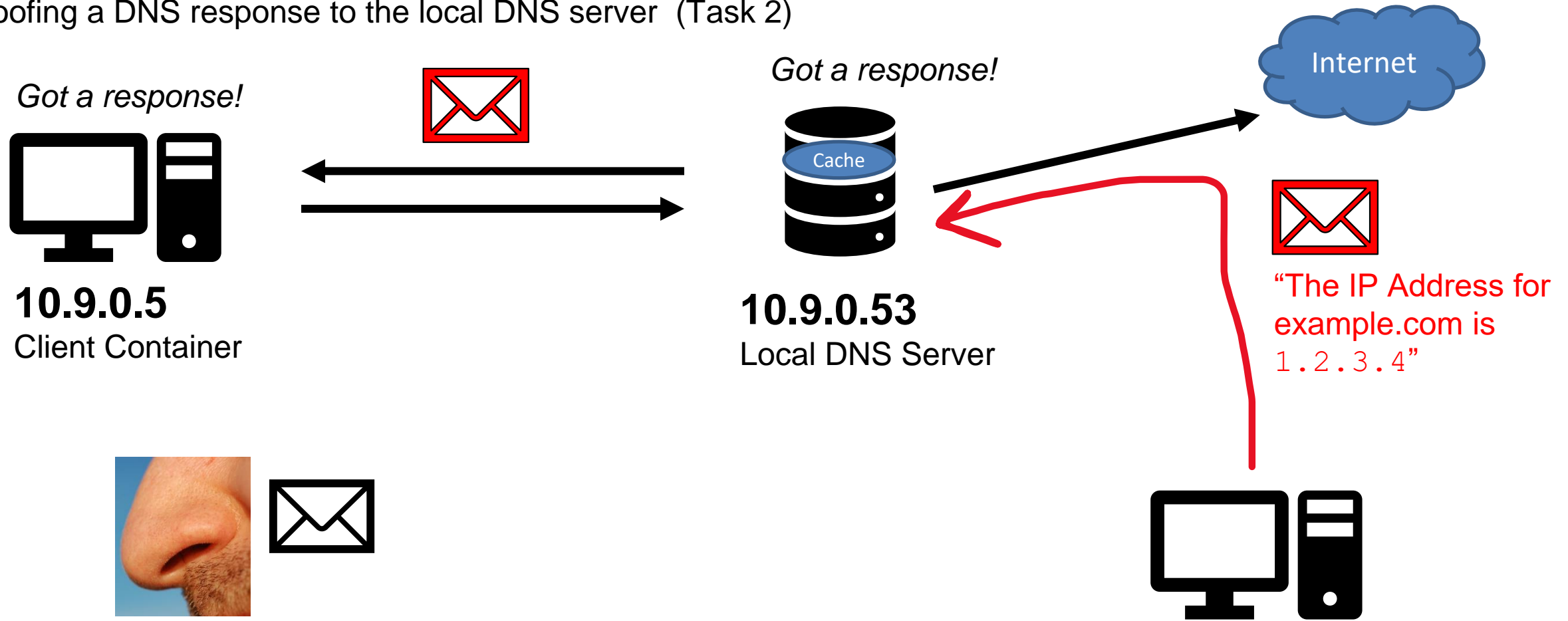
No Class on Friday

Lab 7 (DNS Attacks) due Monday 4/10

Next Monday's lecture will be asynchronous  
(don't come to class)



## 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

Attacker VM (10.9.0.1)

```
^C[03/29/23]seed@VM:~/.../07_dns_attacks$ sudo python3
spoofer.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

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
spoofer.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”

Local DNS Server (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

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

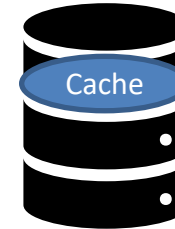
;; QUESTION SECTION:
;www.example.com.                IN      A

;; 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**

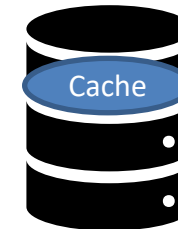
We have “poisoned” this DNS server



# Spoofing a DNS Response packet to the LOCAL DNS Server

## Local DNS Server (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**

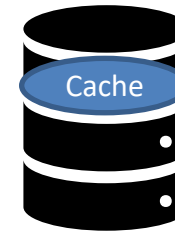
**Type A Records:** IPv4 Addresses. Ie. the IP Address for [www.example.com](http://www.example.com) is 1.2.3.4

**Type NS Records:** Authoritative DNS Servers for a domain. Ie. the Authoritative DNS Server for [www.example.com](http://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**

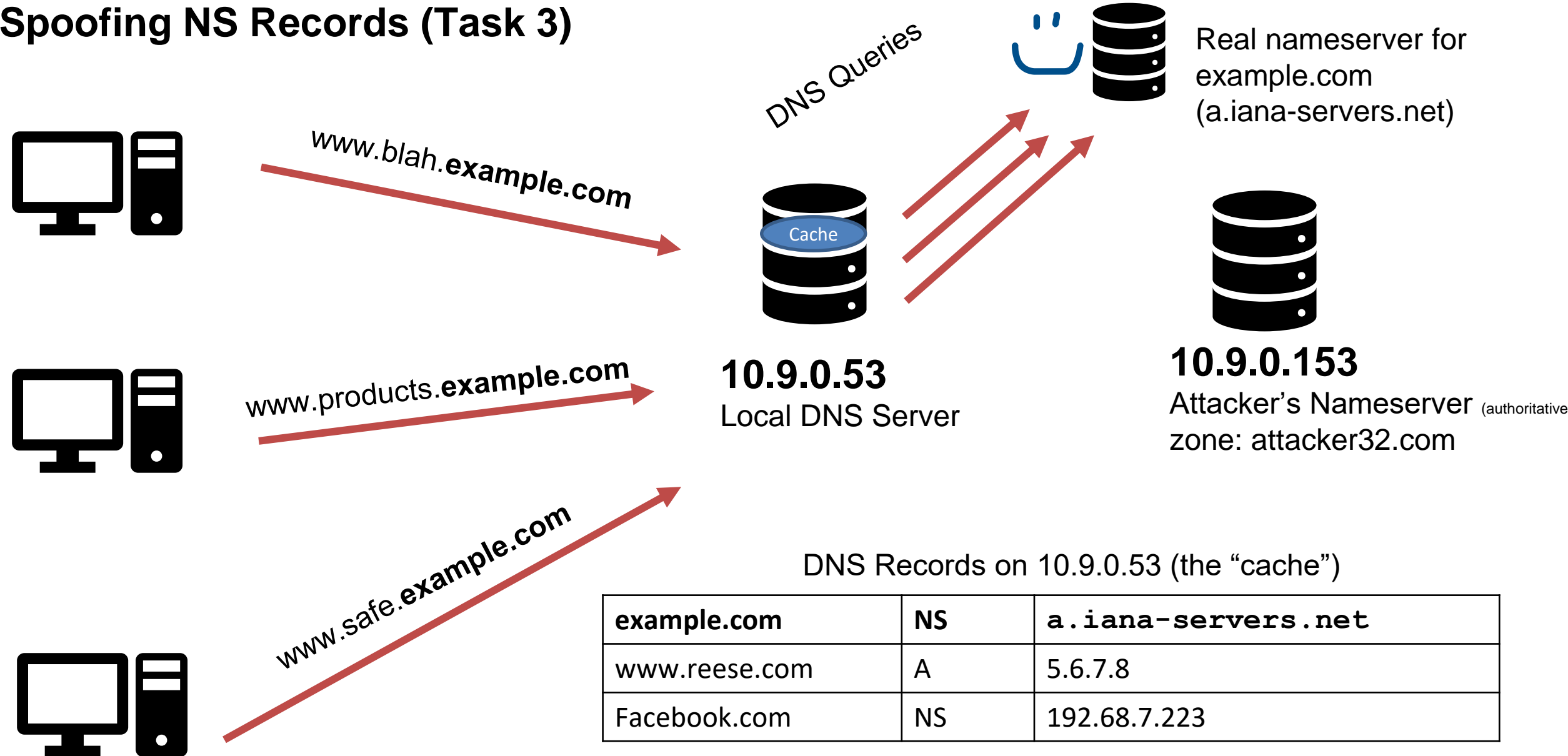
**Type A Records:** IPv4 Addresses. Ie. the IP Address for [www.example.com](http://www.example.com) is 1.2.3.4

**Type NS Records:** Authoritative DNS Servers for a domain. Ie. the Authoritative DNS Server for [www.example.com](http://www.example.com) is a.iana-servers.net

Our next task will be to poison a local DNS cache with **NS type records**.

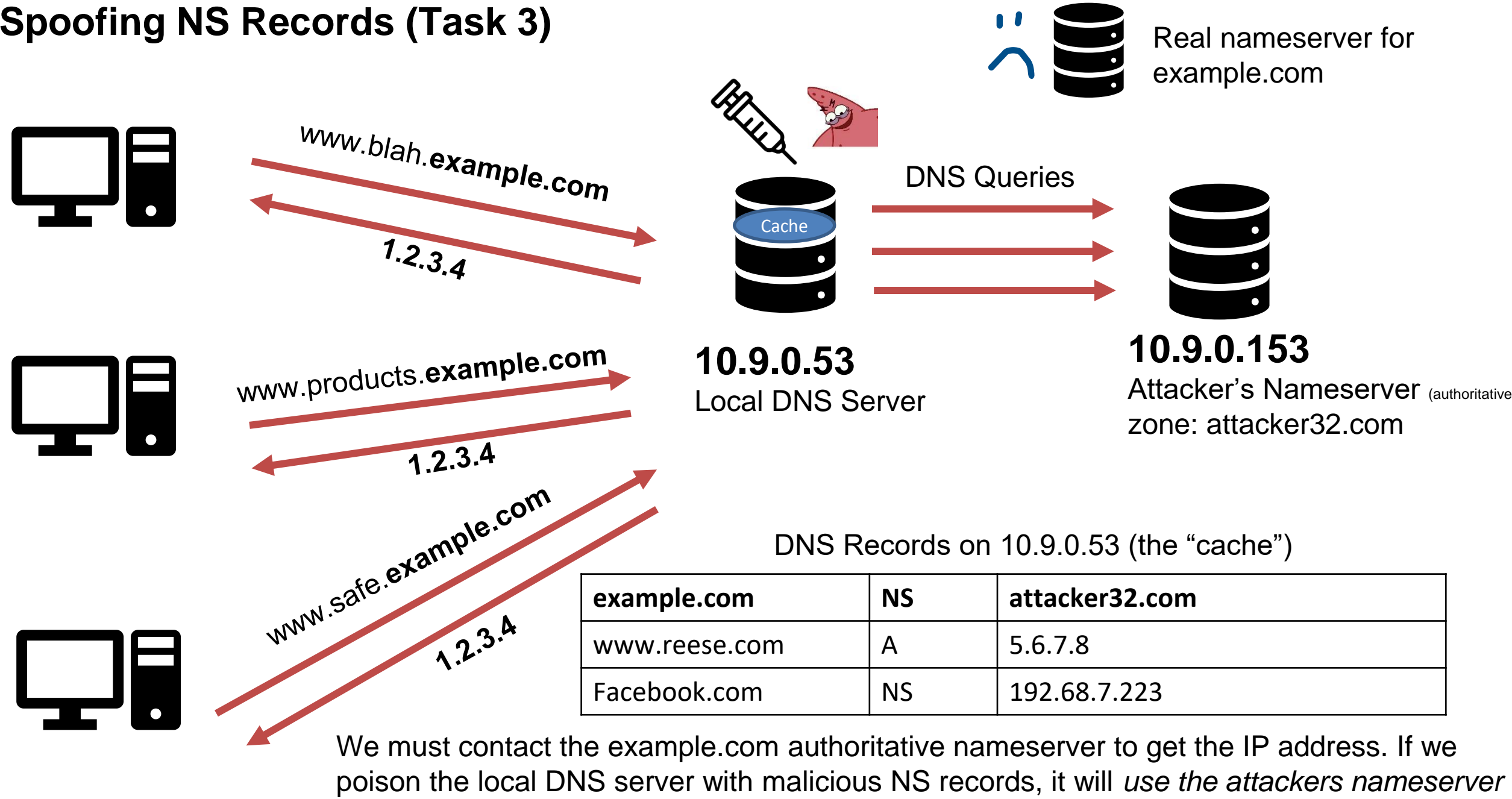
→ Visitor that want to access any webpage in the domain example.com will use the attackers nameserver

# Spoofing NS Records (Task 3)





# Spoofing NS Records (Task 3)



**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)**

## 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)

## 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 Server (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 spoof_ns.py 10.9.0.53
```

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

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

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

## Local DNS Server (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 spoof_ns.py 10.9.0.53
```

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

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

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

## Local DNS Server (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

5. Check the cache on the local DNS server

```
root@e8f13d4a656e:/# rndc dumpdb -cache  
root@e8f13d4a656e:/# cat /var/cache/bind/dump.db | grep example  
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?

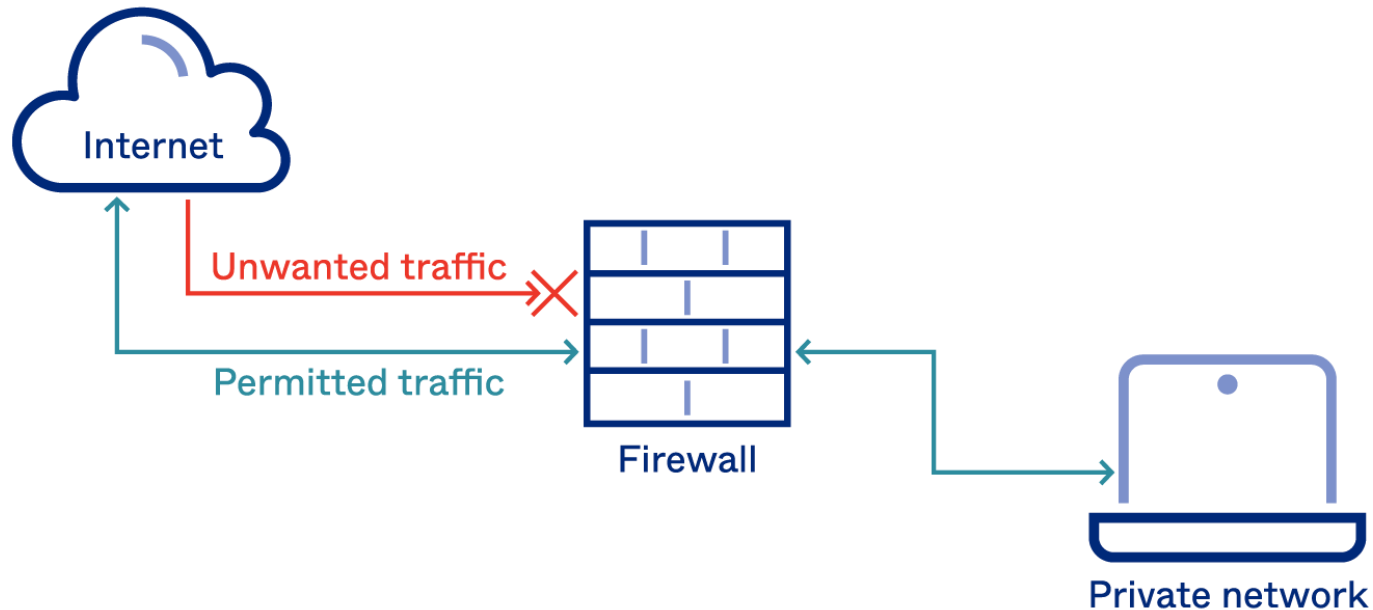
- Coming soon <sup>TM</sup>

# Lab 7



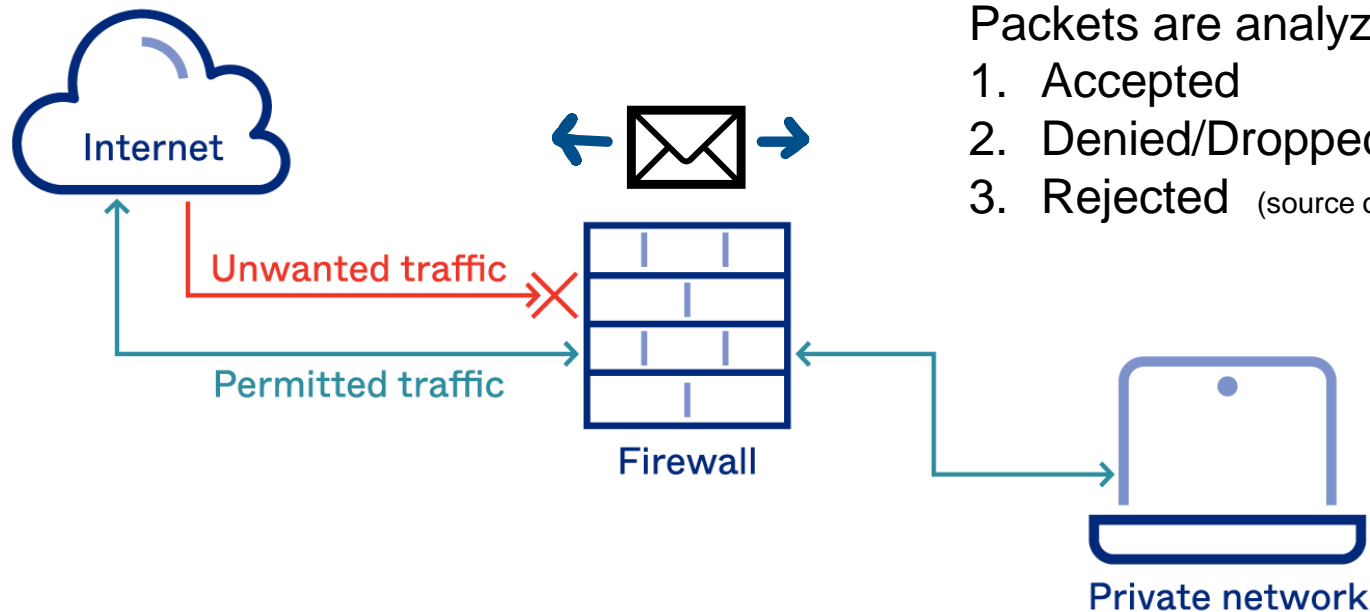
A **firewall** is a part of a computer system or network that is designed to stop unauthorized traffic from one network to another.

- 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



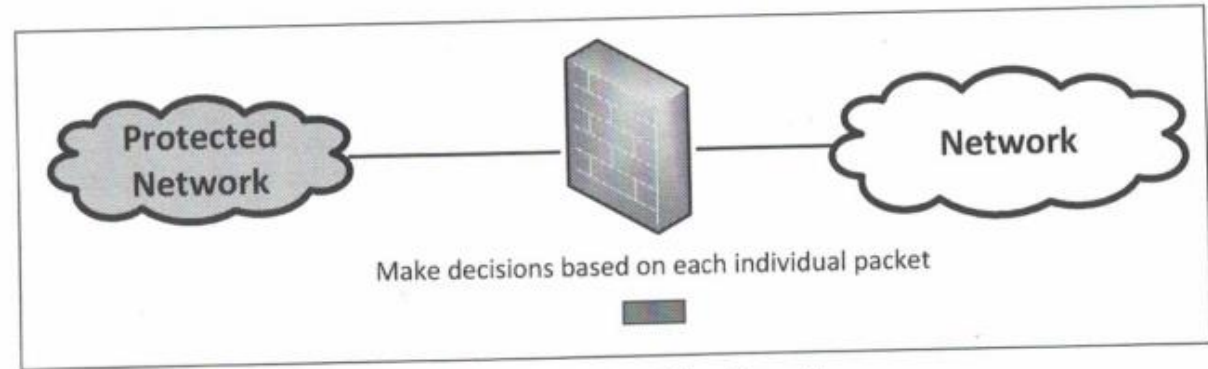
Packets are analyzed by the Firewall, and are:

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

# Three types of firewalls

## 1. Packet Filter

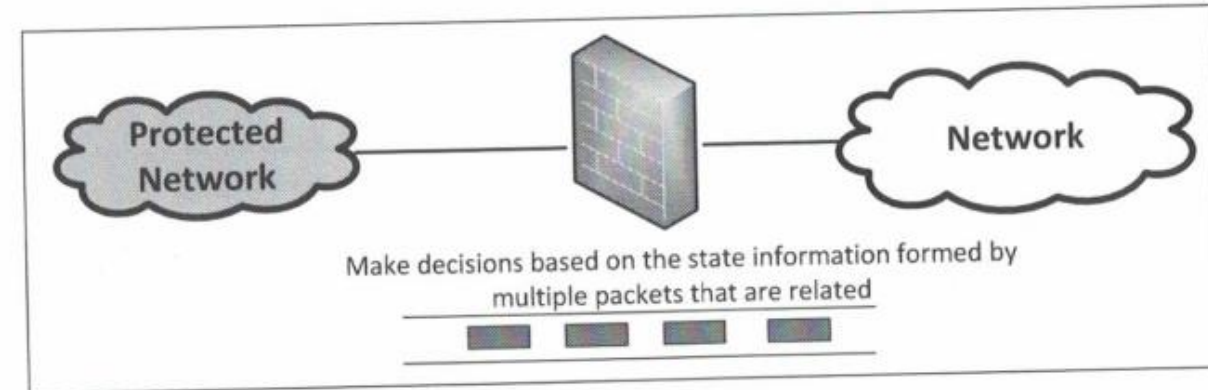
→ Makes decisions based on information within packet (IP address, port #s, etc)



(A) Packet Filter Firewall

## 2. Stateful Firewall

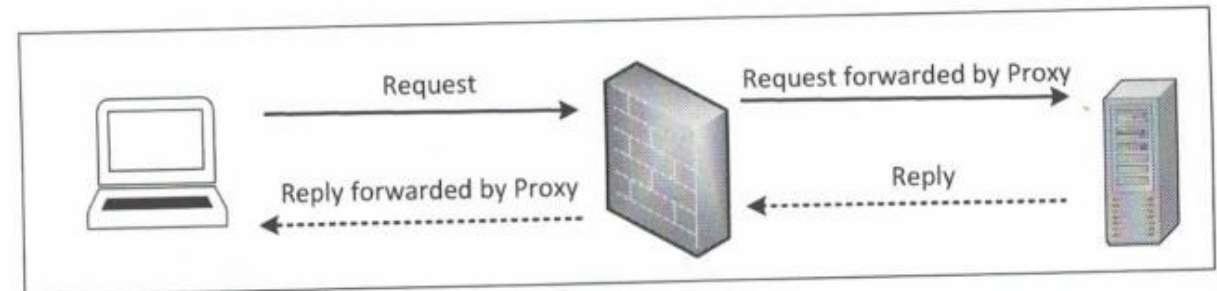
→ Makes decisions based “sessions” and streams of related packets



(B) Stateful Firewall

## 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



(C) Application/Proxy Firewall

# 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)

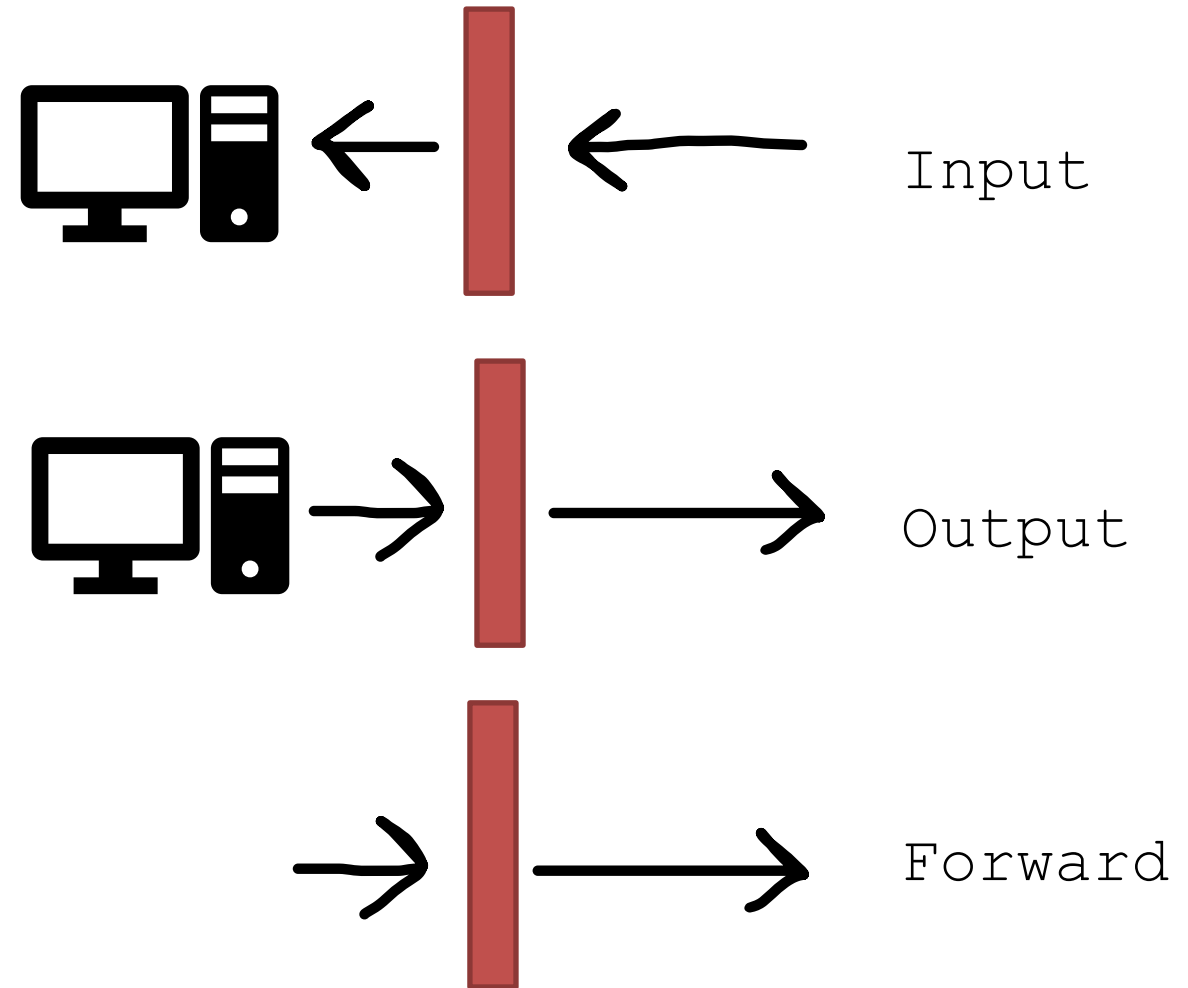
Table Name	Purpose
<code>filter</code>	Packet Filtering
<code>nat</code>	Modifying source or destination network address
<code>mangle</code>	Packet content modification



We will only focus on the filter table

## Three types of chains:

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



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

```
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 **Appended**  
to the `input` chain, which  
means it's a rule for  
incoming traffic

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
```



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           tcp dpt:
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)