CMSC414 Computer and Network Security

Final Review

Yizheng Chen | University of Maryland
surrealyz.github.io

May 7, 2026

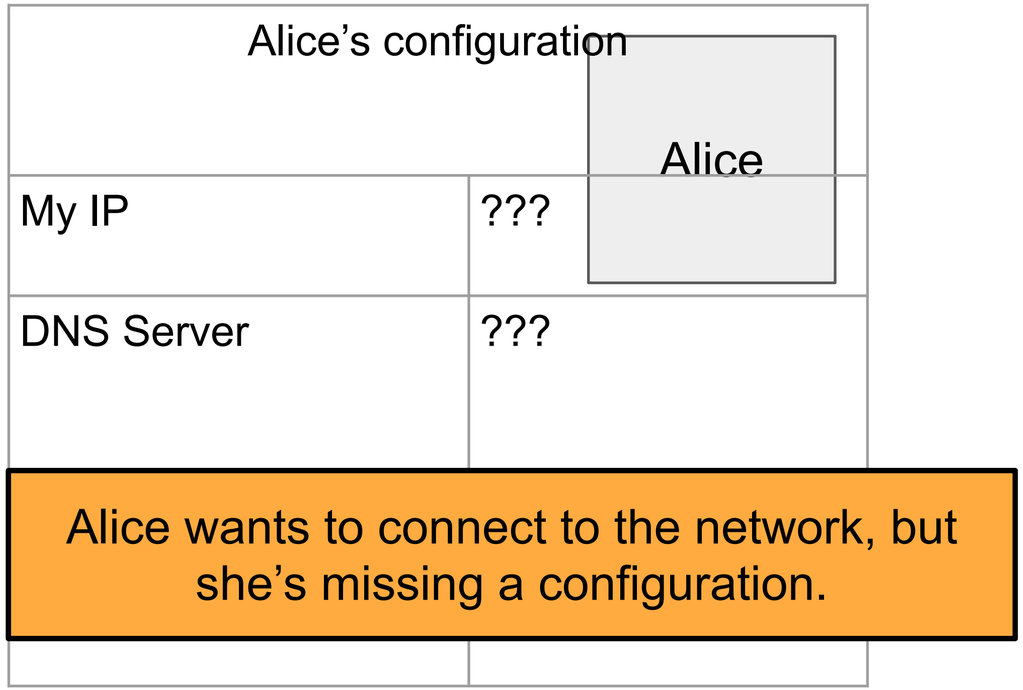
DHCP: Initial Network Configuration

- To connect to a network, a user needs:
  - An IP address so that other people can contact the user
  - The IP address of the DNS server to look up IPs of domain names
  - The IP address of the router (gateway) to contact machines outside of the LAN

- The first time a user connects, they don’t have this information yet
  - The user also doesn’t know who to ask for this information

- **DHCP** gives the user a configuration when they first join the network

Dynamic Host Configuration Protocol (DHCP)



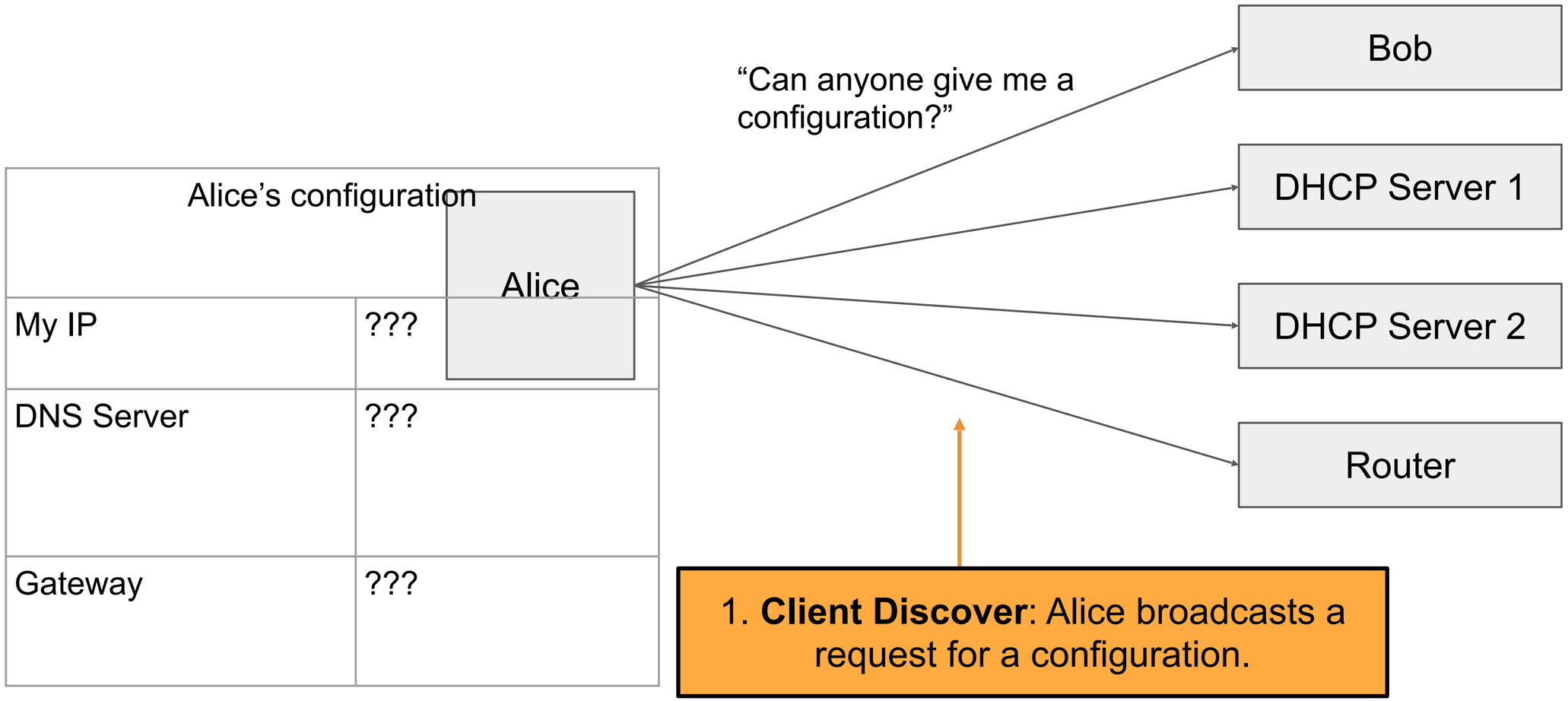
Bob

DHCP Server 1

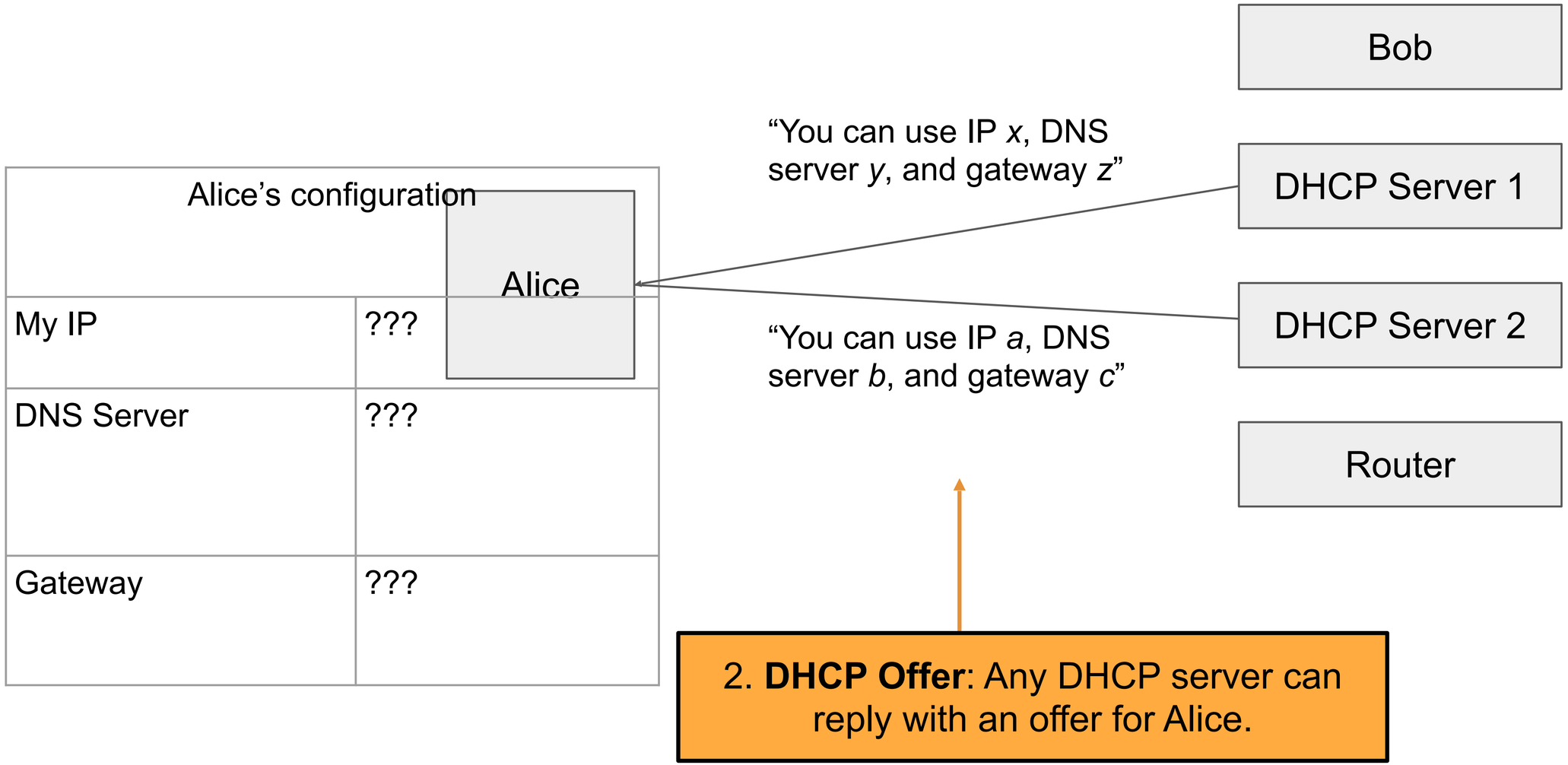
DHCP Server 2

Router

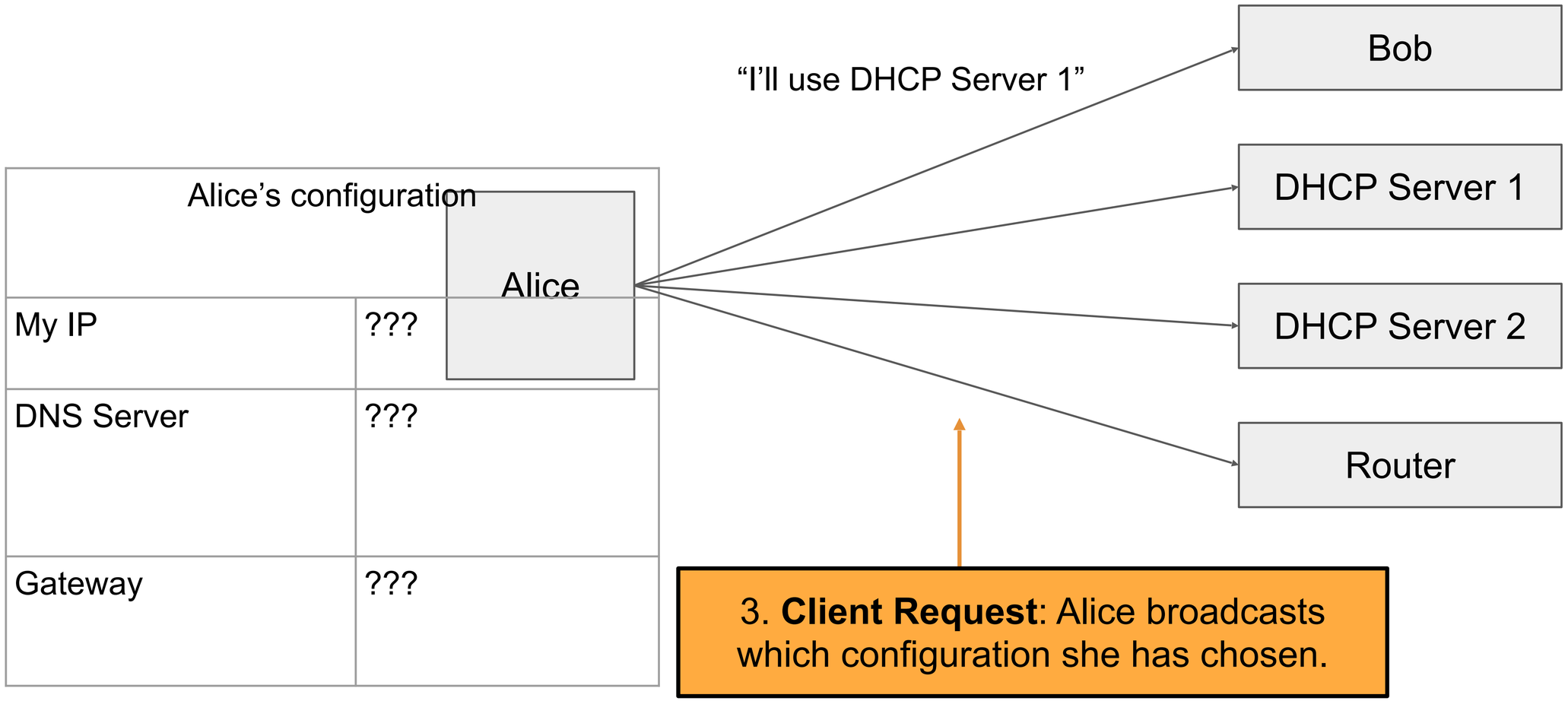
Dynamic Host Configuration Protocol (DHCP)



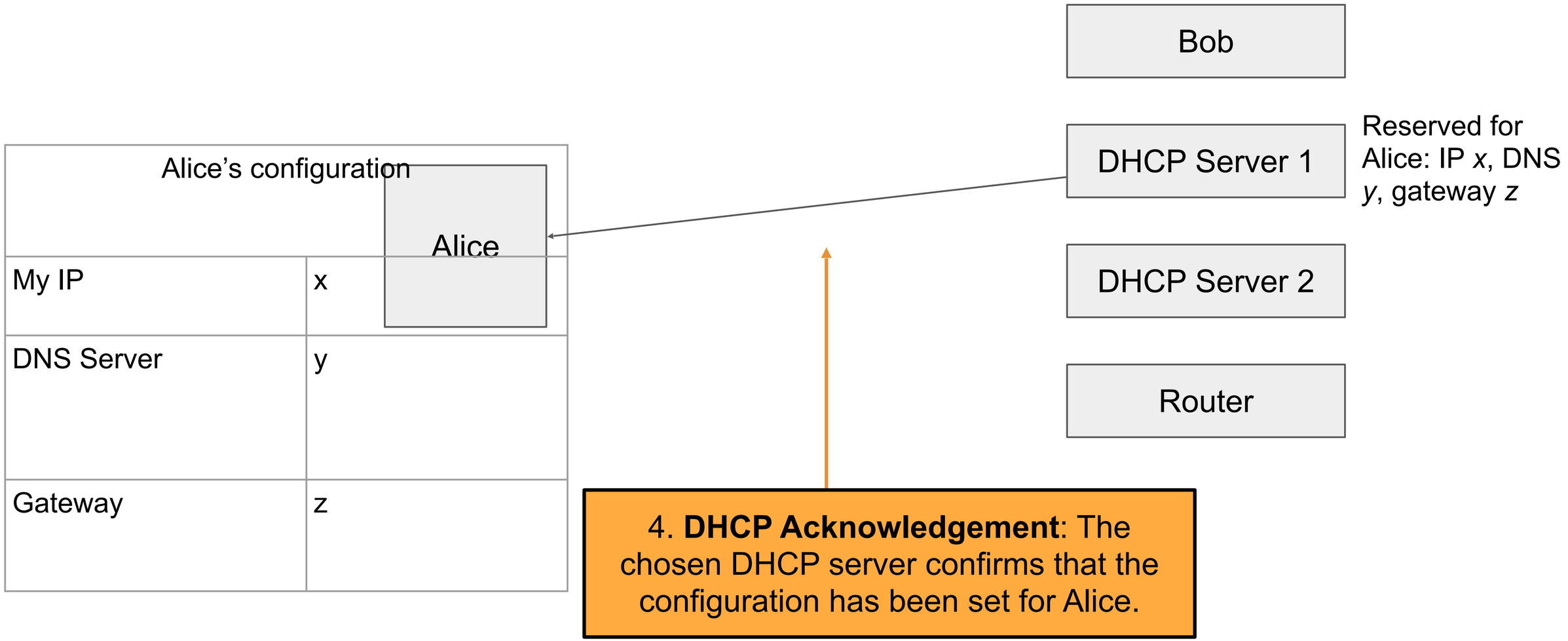
Dynamic Host Configuration Protocol (DHCP)



Dynamic Host Configuration Protocol (DHCP)



Dynamic Host Configuration Protocol (DHCP)



Steps of the DHCP Handshake

1. **Client Discover:** The client broadcasts a request for a configuration
2. **DHCP Offer**: Any DHCP server can respond with a configuration offer
    - Usually only one DHCP server responds
    - The offer includes an IP address for the client, the DNS server's IP address, and the (gateway) router's IP address
    - The offer also has an expiration time (how long the user can use this configuration)
3. **Client Request**: The client broadcasts which configuration it has chosen
    - If multiple DHCP servers made offers, the ones that were not chosen discard their offer
    - The chosen DHCP server gives the offer to the client
4. **DHCP Acknowledgement**: The chosen server confirms that its configuration has been given to the client

DHCP Attacks

- Alice has no way of verifying the DHCP response
  - Spoofing: Any attacker on the network can claim to have a configuration
- Alice usually expects only one DHCP server to respond, so she will accept the first response
  - **Race condition**: As long as the attacker responds faster, Alice will accept the attacker's response
- DHCP attacks require Mallory to be in the same LAN as Alice
- DHCP attacks let Mallory become a man-in-the-middle (MITM) attacker
  - Mallory claims the gateway router's address is Mallory's address
    - When Alice sends a message to the rest of the Internet, she actually sends it to Mallory
    - Mallory can modify the message before sending it to its destination
  - Mallory can also claim the DNS server's address is Mallory's address

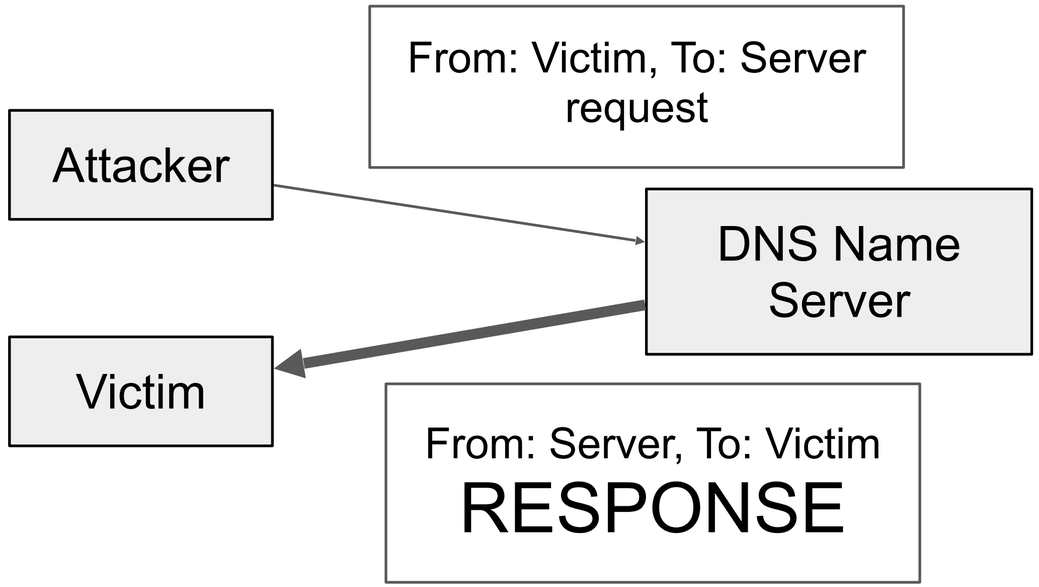
ARP and DHCP

- The attacks on ARP and DHCP are very similar
  - **Spoofing**: The attacker claims to have an answer
  - **Race condition**: The requester accepts the first response. As long as the attacker's response arrives first, it is accepted

- Main vulnerabilities
  - **Broadcast protocols**: Requests are sent to everyone on the LAN, so the attacker can see every request
  - **No trust anchor**: There is no way to verify that responses are legitimate

Amplified Denial-of-Service

- **Amplified denial-of-service**: Use an amplifier to overwhelm the target more effectively
  - Idea: Some services send a large response when sent a small request
  - Spoofing a small request that appears to come from the victim results in a large amount of data sent to the victim
  - Example: DNS amplification
    - Requests contain only the question
    - Responses contain answer records, authority records, and additional records



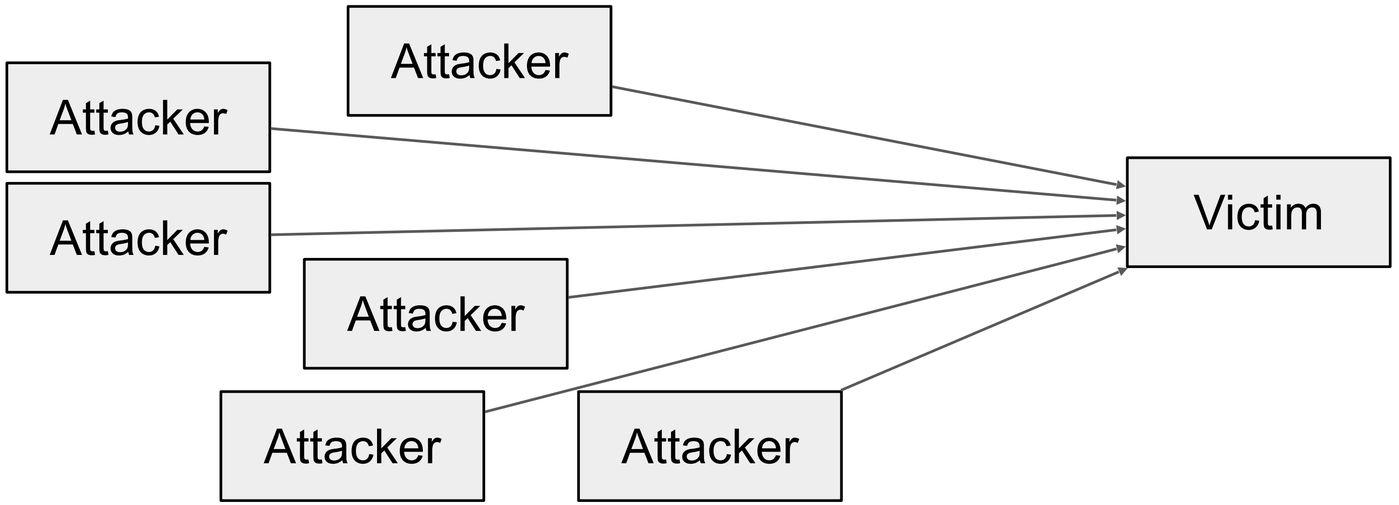
Amplified Denial-of-Service

- Benefits:
  - The attacker’s identity is concealed because the packets come from the amplification server
  - The attacker is able to overwhelm more bandwidth with relatively little bandwidth
    - Amplification servers often have massive bandwidths to support large numbers of users

- Drawbacks:
  - Requires blind spoofing capability
    - Cannot work over TCP, since TCP spoofing is assumed to be hard, only UDP protocols

Distributed Denial-of-Service (DDoS)

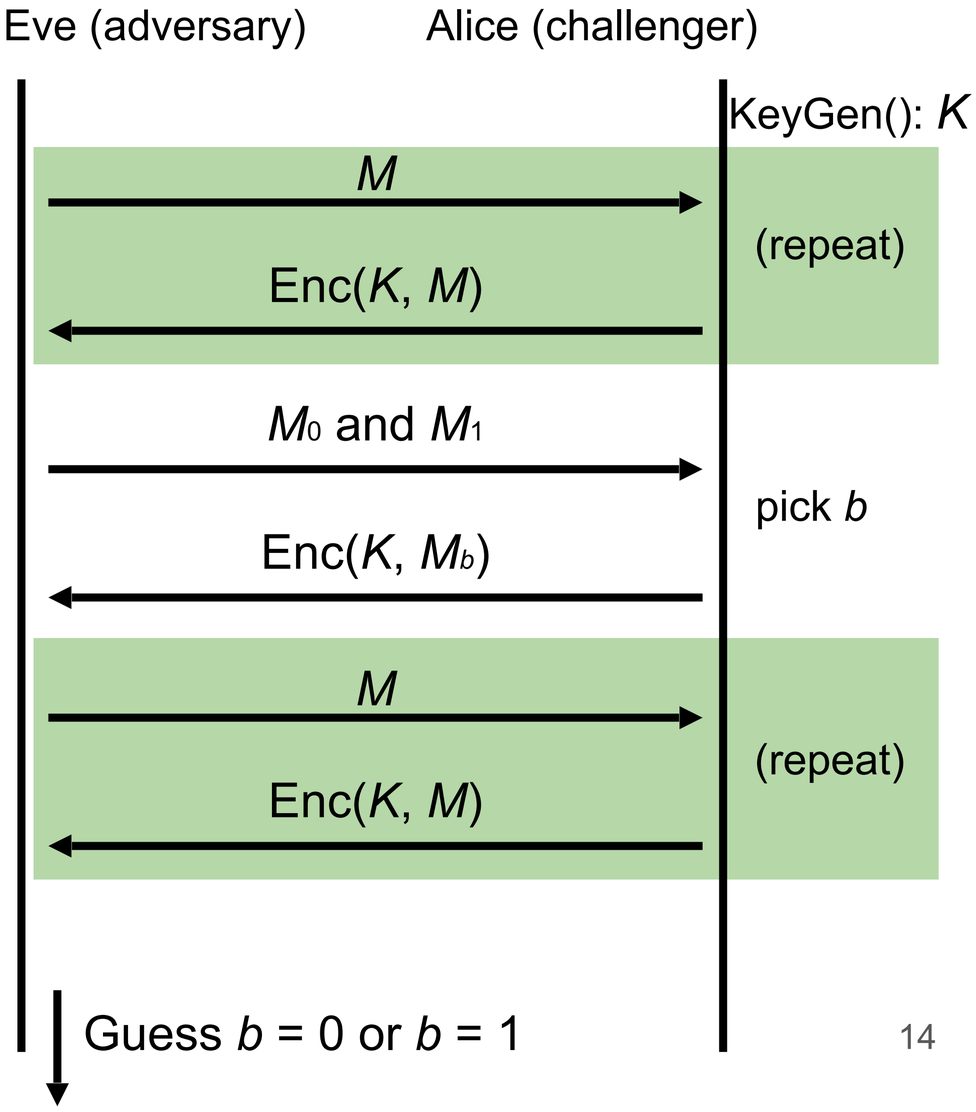
- **Distributed denial-of-service** (**DDoS**): Use multiple systems to overwhelm the target system
  - Controlling many systems gives the attacker a huge amount of bandwidth
  - Sending packets from many sources makes it hard for packet filters to distinguish DDoS traffic from normal traffic
  - **Botnet**: A collection of compromised computers controlled by one attacker
    - The attacker can tell all the computers on the botnet to flood a given target



IND-CPA (indistinguishability under chosen plaintext attack)

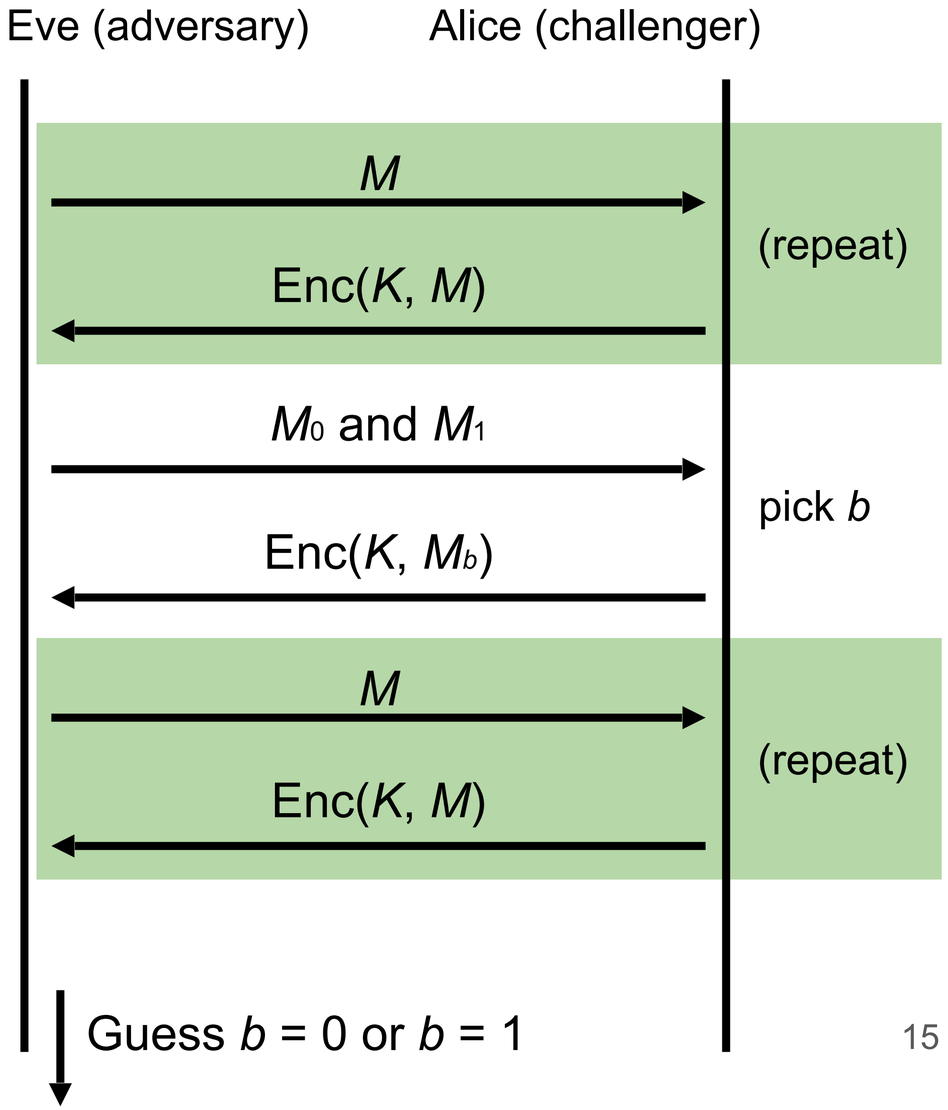
1. Eve may choose plaintexts to send to Alice and receives their ciphertexts
2. Eve issues a pair of plaintexts M_0 and M_1 to Alice
3. Alice randomly chooses either M_0 or M_1 to encrypt and sends the encryption back
   - Alice does not tell Eve which one was encrypted!
4. Eve may again choose plaintexts to send to Alice and receives their ciphertexts
5. Eventually, Eve outputs a guess as to whether encrypted M_0 or M_1

- An encryption scheme is IND-CPA secure if for all polynomial time attackers Eve:
  - Eve can win with probability ≤ 1/2 + Ɛ, where Ɛ is *negligible.*



Are Block Ciphers IND-CPA Secure?

- Consider the following adversary:
  - Eve sends two different messages M_0 and M_1
  - Eve receives either $E_K(M_0)$ or $E_K(M_1)$
  - Eve requests the encryption of M_0 again
  - Strategy: If the encryption of M_0 matches what she received, guess b = 0. Else, guess b = 1.
- Eve can win the IND-CPA game with probability 1!
  - Block ciphers are not IND-CPA secure because they are deterministic



Issues with Block Ciphers

- Block ciphers are not IND-CPA secure, because they're deterministic
  - A scheme is **deterministic** if the same input always produces the same output
  - No deterministic scheme can be IND-CPA secure because the adversary can always tell if the same message was encrypted twice
- Block ciphers can only encrypt messages of a fixed size
  - For example, AES can only encrypt-decrypt 128-bit messages
  - What if we want to encrypt something longer than 128 bits?
- To address these problems, we'll add **modes of operation** that use block ciphers as a building block!

Three Main Goals of Cryptography

- In cryptography, there are three common properties that we want on our data
- **Confidentiality**: An adversary cannot read our messages.
- **Integrity**: An adversary cannot change our messages without being detected.
- **Authenticity**: I can prove that this message came from the person who claims to have written it.

Authenticity vs Authentication

- **Authenticity**: I can prove that this message came from the person who claims to have written it.
- **Authentication**: verification of identity (are you who you say you are). Examples include username/password and biometrics.

Digital signature properties

Authenticity

Bob can prove that a message signed by Alice is truly from Alice (even without a *pairwise* key)

Integrity

Bob can prove that no one has tampered with a signed message

Non-repudiation

Once Alice signs a message, she cannot subsequently claim she did *not* sign that message

Do *handwritten* signatures at the end of a letter have these properties?

Authenticity

Bob can prove that a message signed by Alice is truly from Alice (even without a *private key*)

Would require unforgeable handwritten signatures. This is the one property they sort of get

Integrity

Bob can prove that no one has tampered with a signed message

Would require having a signature that depended on each part in the body of the letter

Non-repudiation

Once Alice signs a message, she cannot subsequently claim that she did not sign that message

Would require both of the above (unforgeable signature that depends on each part of letter)

Return Oriented Programming (ROP)

Instead of executing an existing function,

execute different pieces of assembly instructions.

ROP Example

- Execute pieces of assembly code in a chain, among many returns
  - They form the functionality that the attacker wants
- What is a Gadget
- How to chain two gadgets together
- How to start executing the first gadget

ROP Gadget

- Gadget: A small set of assembly instructions that already exist in memory
  - Gadgets usually end in a **`ret`** instruction
  - Gadgets are usually **not** full functions

```
foo:
            ...
<foo+7>   addl $4, %esp
<foo+10>  xorl %eax, %ebx
<foo+12>  ret
```

```
bar:
            ...
<bar+22>  andl $1, %edx
<bar+25>  movl $1, %eax
<bar+30>  ret
```

How to chain two gadgets together

- Supposed our goal is:
  - **`movl $1, %eax`**
  - **`xorl %eax, %ebx`**

```
foo:
            ...
<foo+7>   addl $4, %esp
<foo+10>  xorl %eax, %ebx
<foo+12>  ret
```

```
bar:
            ...
<bar+22>  andl $1, %edx
<bar+25>  movl $1, %eax
<bar+30>  ret
```

What to do about ret?

- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`

```
foo:
            ...
<foo+7>   addl $4, %esp
<foo+10>  xorl %eax, %ebx
<foo+12>  ret
```

```
bar:
            ...
<bar+22>  andl $1, %edx
<bar+25>  movl $1, %eax
<bar+30>  ret
```

What to do about ret?

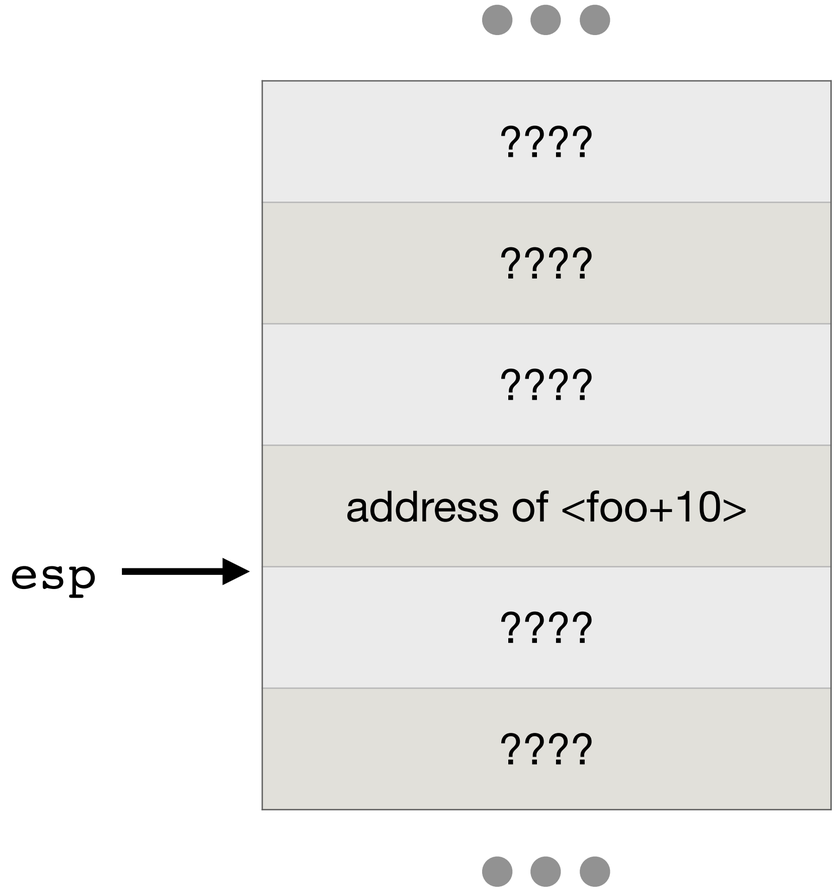
- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`

ret: pop %eip

Put `<foo+10>` on the stack before we do ret

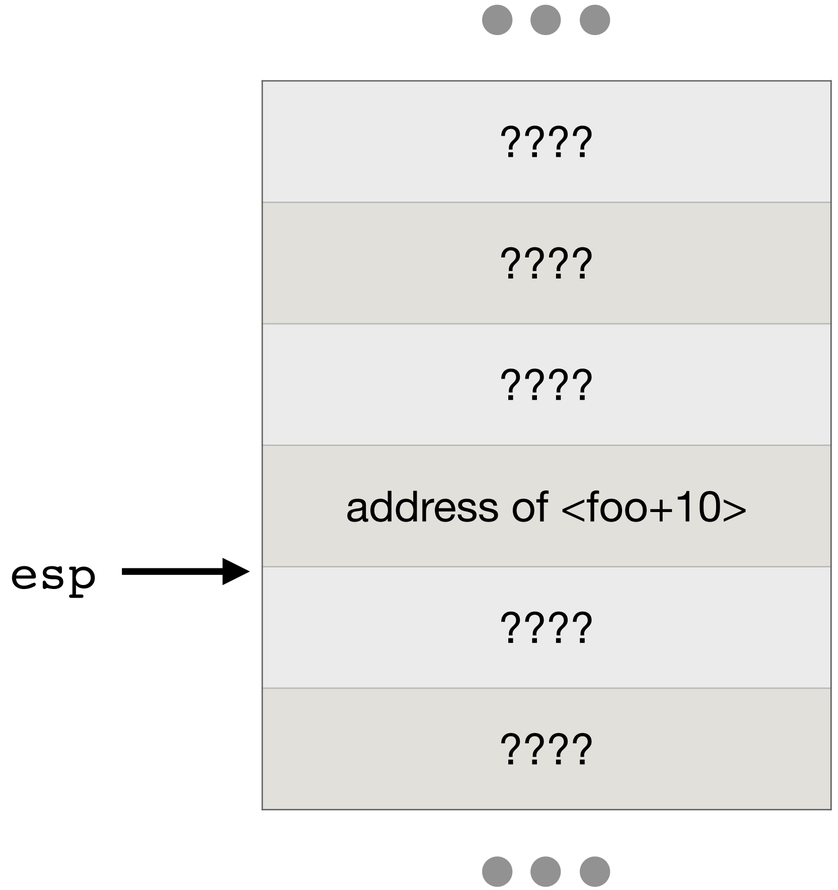
How to chain two gadgets together?

- The following two gadgets allow us to do
  - **<bar+25> movl $1, %eax**
  - **<bar+30> ret**
  - **<foo+10> xorl %eax, %ebx**
  - **<foo+12> ret**



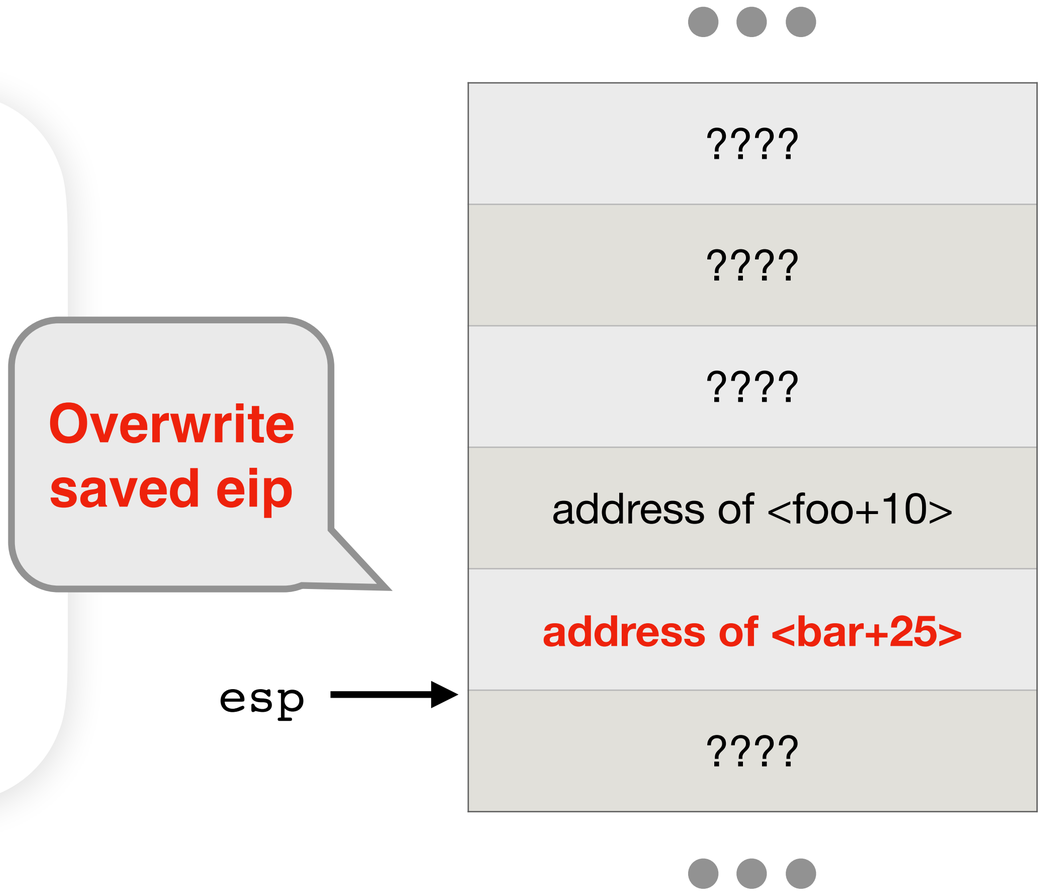
How to start executing?

- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`



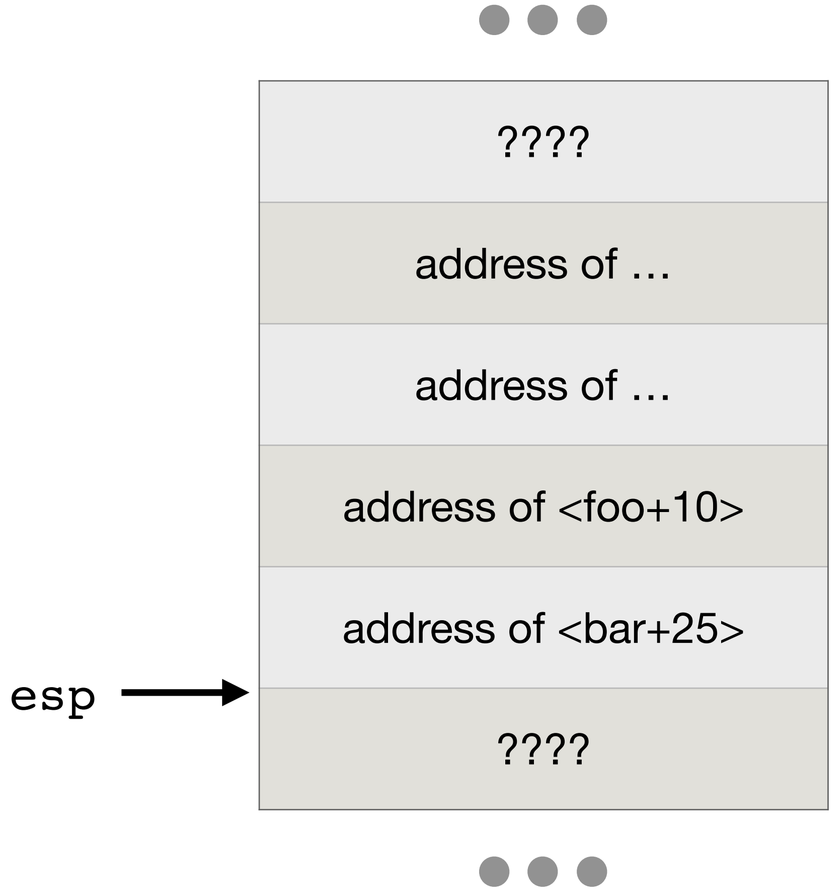
How to start executing?

- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`



ROP

- If we have many gadgets
  - **<bar+25> movl $1, %eax**
  - **<bar+30> ret**
  - **<foo+10> xorl %eax, %ebx**
  - **<foo+12> ret**
  - **<…> …**
  - **<…> ret**
  - **<…> …**
  - **<…> ret**
  - …



ROP

- Gadget: A small set of assembly instructions that already exist in memory
  - Gadgets usually end in a **`ret`** instruction
  - Gadgets are usually **not** full functions
- ROP strategy: We write a chain of return addresses starting at the RIP to achieve the behavior we want
  - Each return address points to a gadget
  - The gadget executes its instructions and ends with a ret instruction
  - The ret instruction jumps to the address of the next gadget on the stack

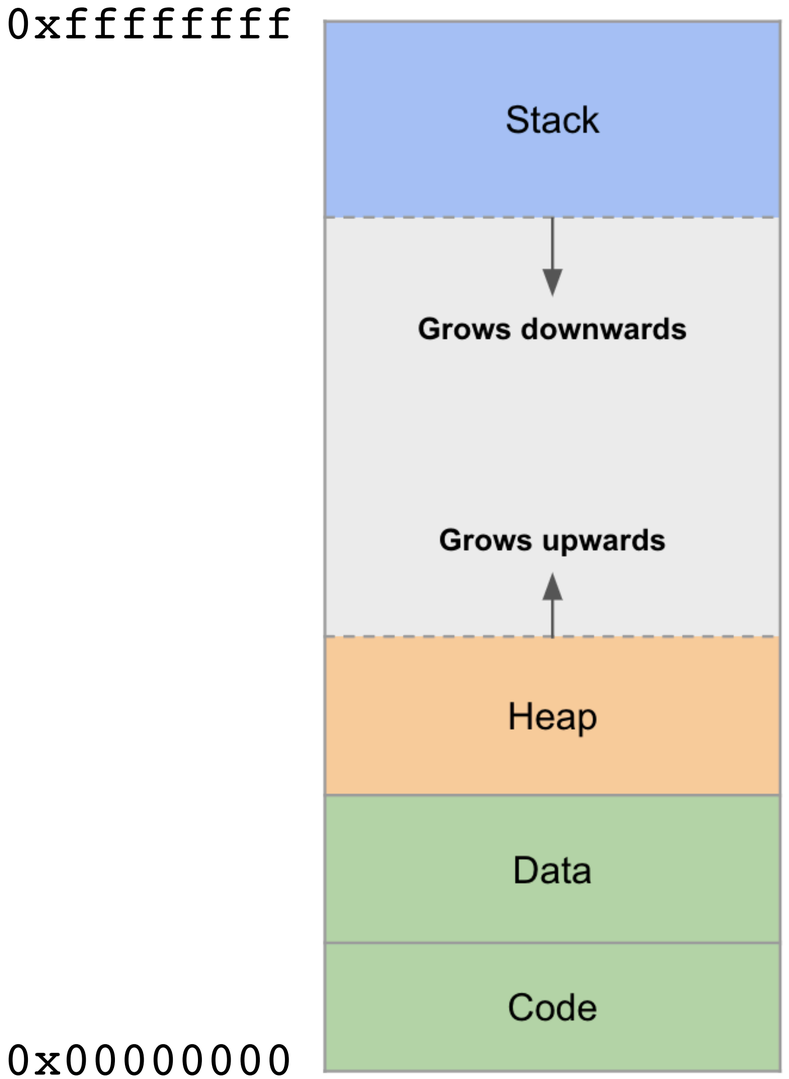
ROP

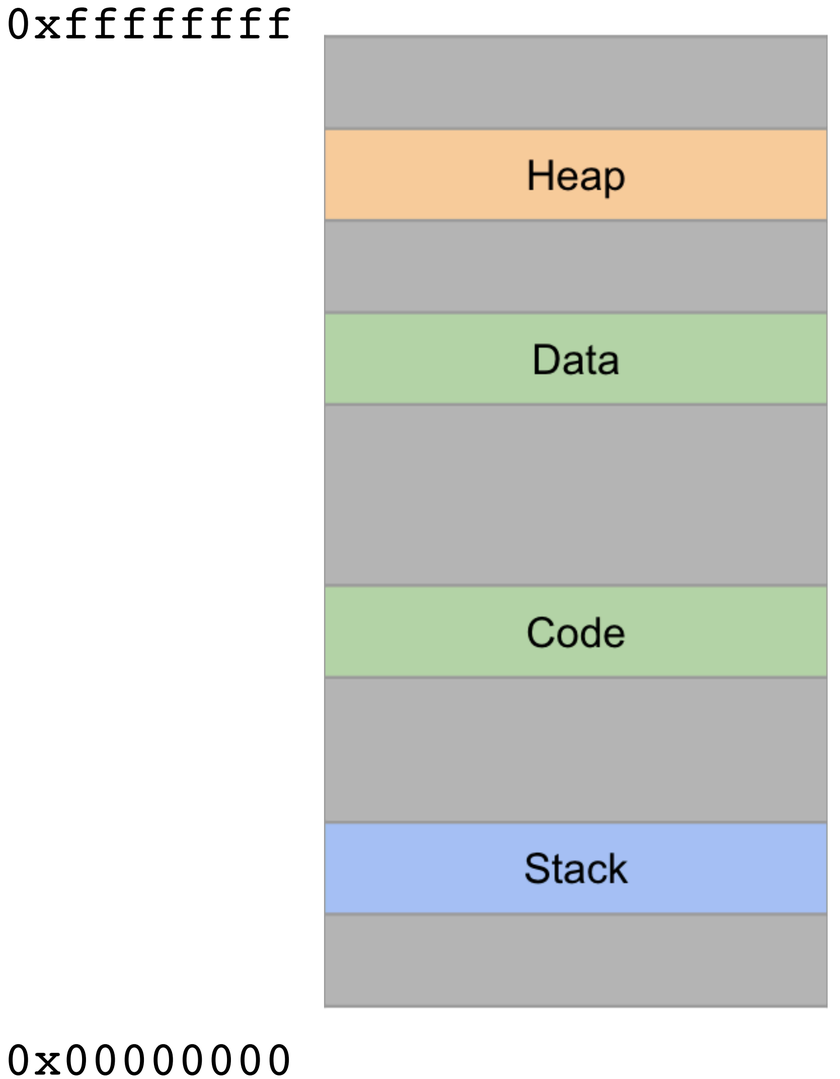
- If the code base is big enough (imports enough libraries), there are usually enough gadgets in memory for you to run any shellcode you want
- **ROP compilers** can automatically generate a ROP chain for you based on a target binary and desired malicious code!
- Non-executable pages is not a huge issue for attackers nowadays
  - Having writable and executable pages makes an attacker's life easier, but not *that* much easier

Address Space Layout Randomization

- Goal: make it hard for attackers to place shell code on the stack, on the heap, or find out the address of the code
- Randomize the addresses of code, data, heap, stack
- Theoretically, very hard to know the addresses, so we can mitigate the attacks

Address Space Layout Randomization



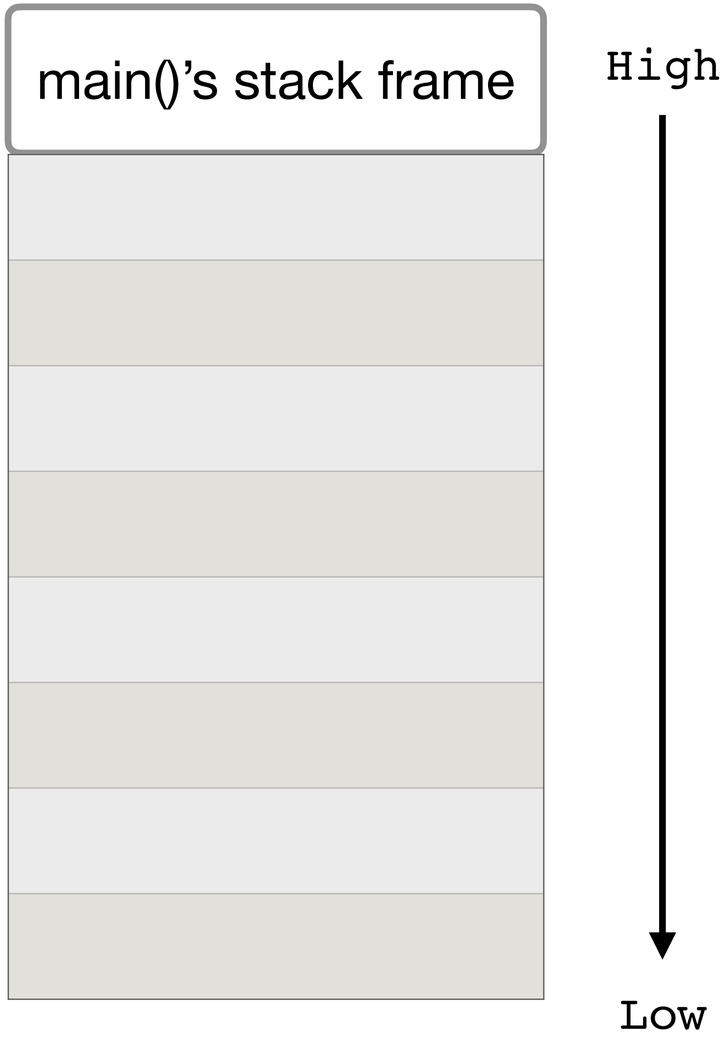


Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead
- However…
- Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)
  - Leak the address of a pointer, whose address relative to your shellcode is known (stack pointer, RIP)
  - Guess the address of your shellcode: Brute-force

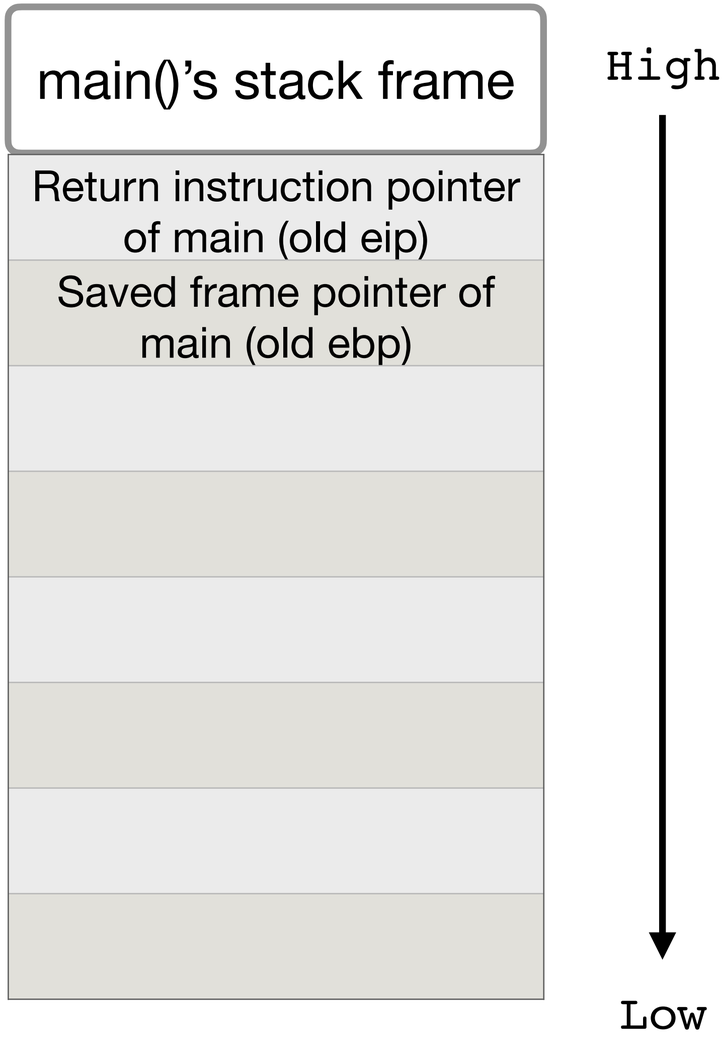
printf

```
void main() {
    not_vulnerable();
}

void not_vulnerable() {
    printf("x val: %d, y val:
%d, z val: %d\n", x, y, z);
}
```



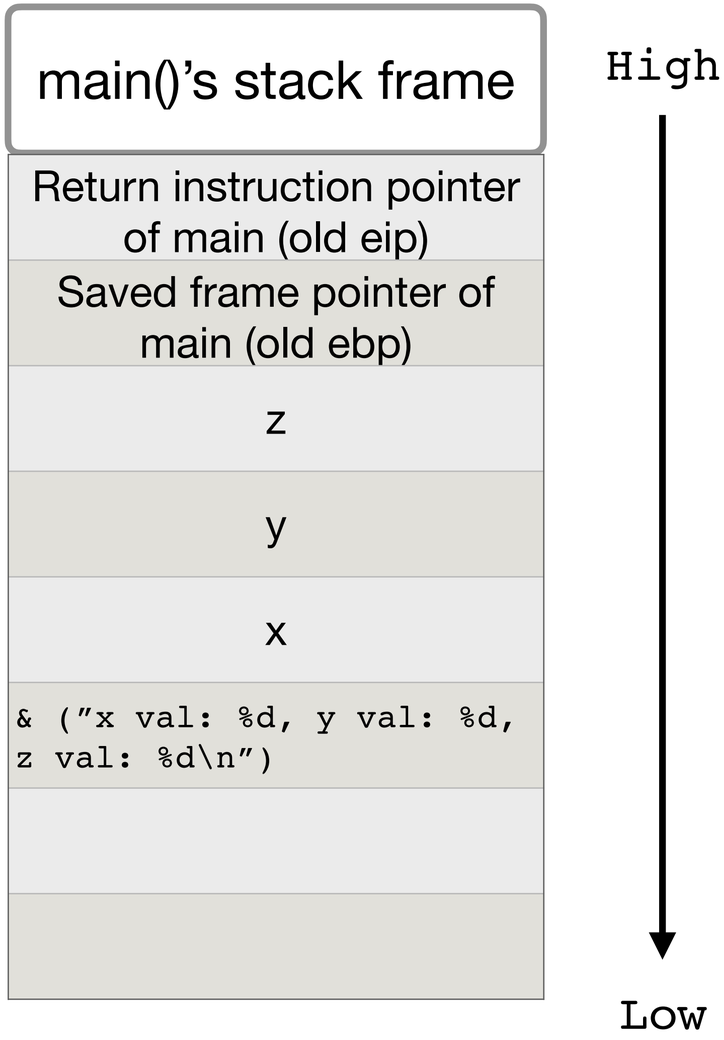
printf

```
void main() {
    not_vulnerable();
}

void not_vulnerable() {
    printf("x val: %d, y val:
%d, z val: %d\n", x, y, z);
}
```



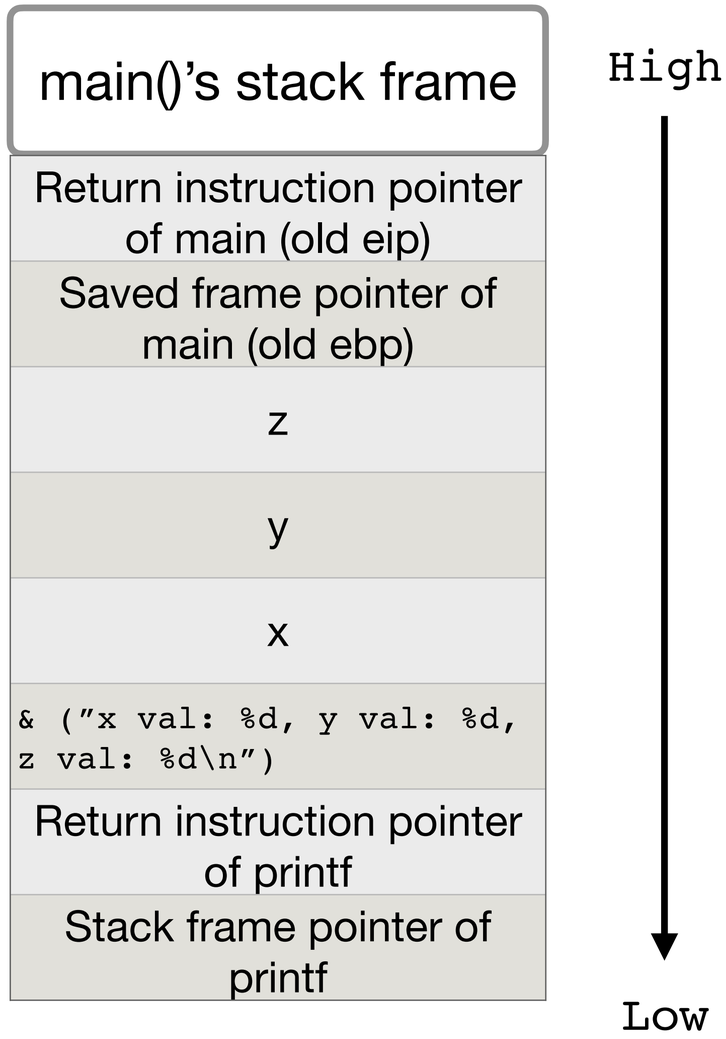
printf

```
void main() {
    not_vulnerable();
}

void not_vulnerable() {
    printf("x val: %d, y val:
%d, z val: %d\n", x, y, z);
}
```



printf

```
void main() {
    not_vulnerable();
}

void not_vulnerable() {
    printf("x val: %d, y val:
%d, z val: %d\n", x, y, z);
}
```

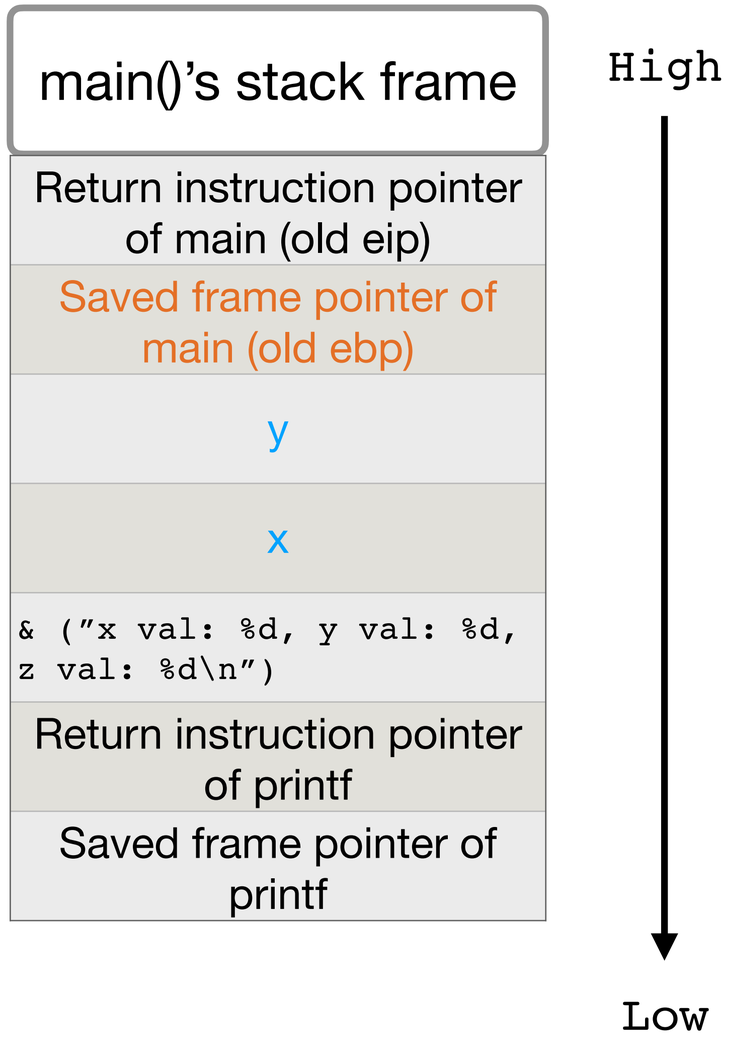


The format string “x val: … %d\n” controls the behavior of printf

Internal pointer in printf looks for content on the stack

Format String Vulnerability

```
void main() {
    vulnerable();
}

void vulnerable() {
    printf("x val: %d, y val:
%d, z val: %d\n", x, y);
}
```



The format string "x val: … %d\n" controls the behavior of printf

Internal pointer in printf looks for content on the stack

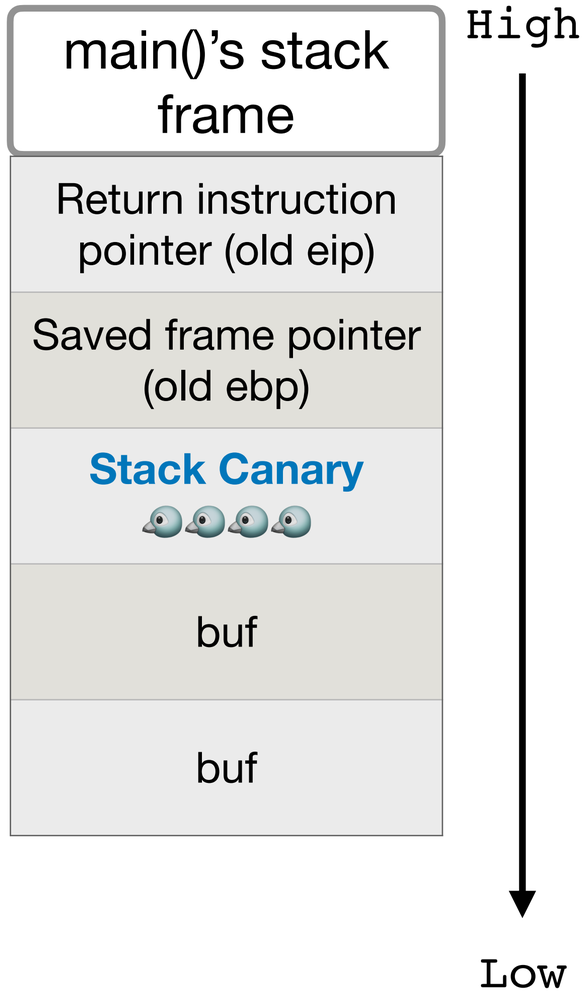
Other Formats

- %s → Treat the argument as an address and print the string at that address up until the first null byte
- %n → Treat the argument as an address and write the number of characters that have been printed so far to that address
- %c → Treat the argument as a value and print it out as a character
- %x → Print the variable as unsigned hexadecimal integer
- %[b]u → Print out [b] bytes starting from the argument

Format string vulnerability: the attacker can learn any value stored in memory and can take control of your program.

Stack Canaries

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



The attack will have to overwrite the **stack canary**

Subverting Stack Canaries

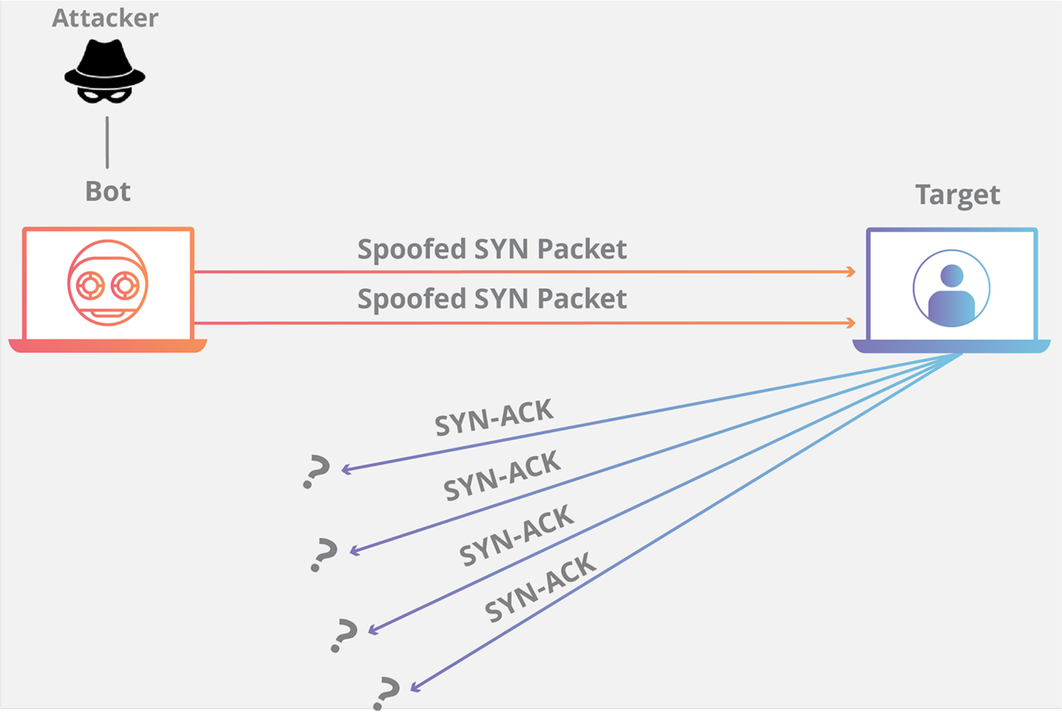
- **Leak** the value of the canary: Overwrite the canary with itself
- **Bypass** the value of the canary: Use a random write, not a sequential write
- **Guess** the value of the canary: Brute-force

Cross-Site Scripting (XSS)

- **Cross-site scripting (XSS)**: Injecting JavaScript into websites that are viewed by other users
  - Cross-site scripting subverts the same-origin policy
- Two main types of XSS
  - Stored XSS
  - Reflected XSS

TCP SYN Food Attacks

1. The attacker sends a high volume of SYN packets to the targeted server, often with spoofed IP addresses.
2. The server then responds to each one of the connection requests and leaves an open port ready to receive the response.
3. While the server waits for the final ACK packet, which never arrives, the attacker continues to send more SYN packets. The arrival of each new SYN packet causes the server to temporarily maintain a new open port connection for a certain length of time, and once all the available ports have been utilized the server is unable to function normally.



Same-Origin Policy: Definition

- **Same-origin policy:** A rule that prevents one website from tampering with other unrelated websites
  - Enforced by the web browser
  - Prevents a malicious website from tampering with behavior on other websites

Same-Origin Policy

- Every webpage has an origin defined by its URL with three parts:
  - Protocol: The protocol in the URL
  - Domain: The domain in the URL's location
  - Port: The port in the URL's location
    - If no port is specified, the default is 80 for HTTP and 443 for HTTPS
- https://www.example.com:443/image.png
- http://example.com/files/image.png 80 (default port)

Same-Origin Policy

- Two webpages have the same origin if and only if the protocol, domain, and port of the URL all match exactly.

First Webpage	Second Webpage	Same Origin?
http://www.example.com	https://www.example.com	Protocol mismatch
http://www.example.com	http://example.com	Domain mismatch
http://www.example.com[:80]	http://www.example.com:8000	Port mismatch

Same-Origin Policy

- Two websites with different origins cannot interact with each other
  - Example: If `example.com` embeds `evil.com`, the inner frame cannot interact with the outer frame, and the outer frame cannot interact with the inner-frame
- Rule enforced by the browser

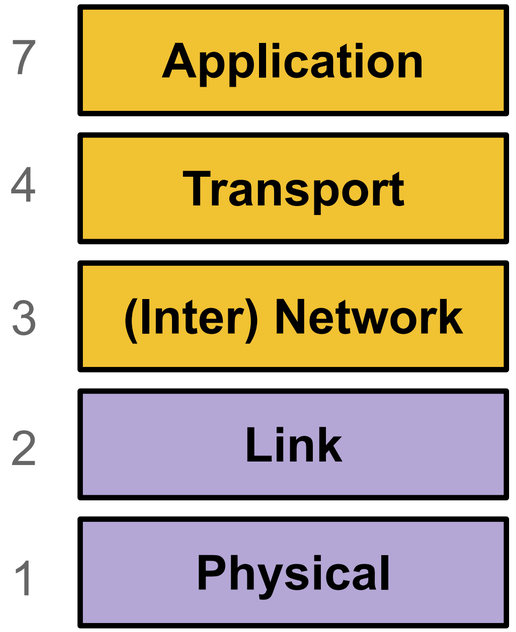
Exceptions to the Same-Origin Policy

- Exception: JavaScript runs with the origin of the page that loads it
  - Example: If `example.com` fetches JavaScript from evil.com, the JavaScript has the origin of `example.com`
  - Intuition: `example.com` has “copy-pasted” JavaScript onto its webpage
- Exception: Websites can fetch and display images from other origins
  - However, the website only knows about the image’s size and dimensions (cannot actually manipulate the image)
- Exception: Websites can agree to allow some limited sharing
  - Cross-origin resource sharing (CORS)
  - The postMessage function in JavaScript let websites communicate with each other

Last Time: Intro to Networking

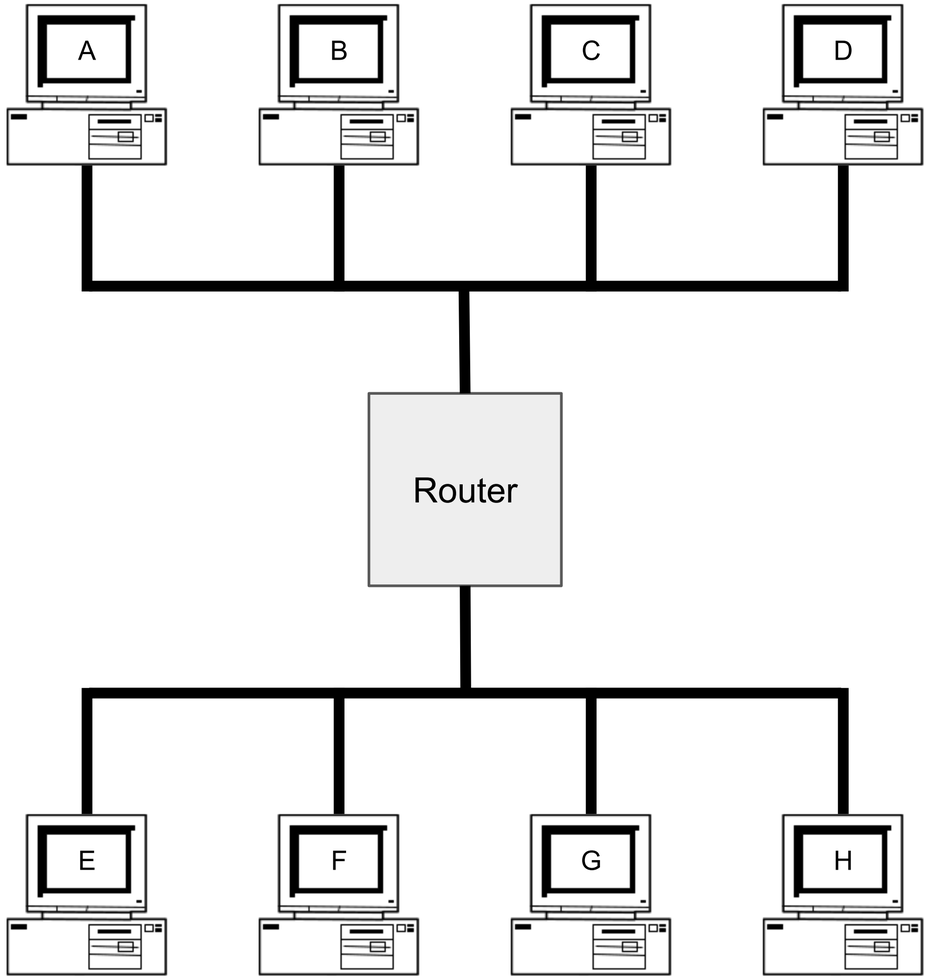
- Internet: A global network of computers
  - Protocols: Agreed-upon systems of communication

- OSI model: A layered model of protocols
  - Layer 1: Communication of bits
  - Layer 2: Local frame delivery
    - Ethernet protocol
    - MAC addresses (6-byte)
  - Layer 3: Global packet delivery
    - IP protocol
    - IP addresses (4-byte or 16-byte)
  - Layer 4: Transport of data
  - Layer 7: Applications and services



Review: Layer 2 and Layer 3

- Local area network (LAN): A set of machines connected in a local network
  - The MAC identifies devices on layer 2
- Internet protocol (IP): Many LANs connected together with routers
  - The IP identifies devices on layer 3

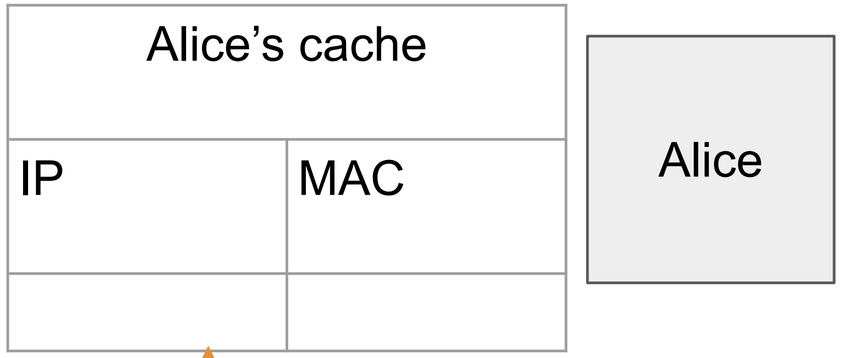


Address Resolution Protocol (ARP)

- **ARP**: Translates layer 3 IP addresses to layer 2 MAC addresses
  - Example: Alice wants to send a message to Bob on the local network, but Alice only knows Bob's IP address (`1.2.3.4`). To use layer 2 protocols, she must learn Bob's MAC address.

Attacks on ARP

Alice knows Bob's IP address (`1.2.3.4`) but wants to learn Bob's MAC address.



Alice

Bob

Charlie

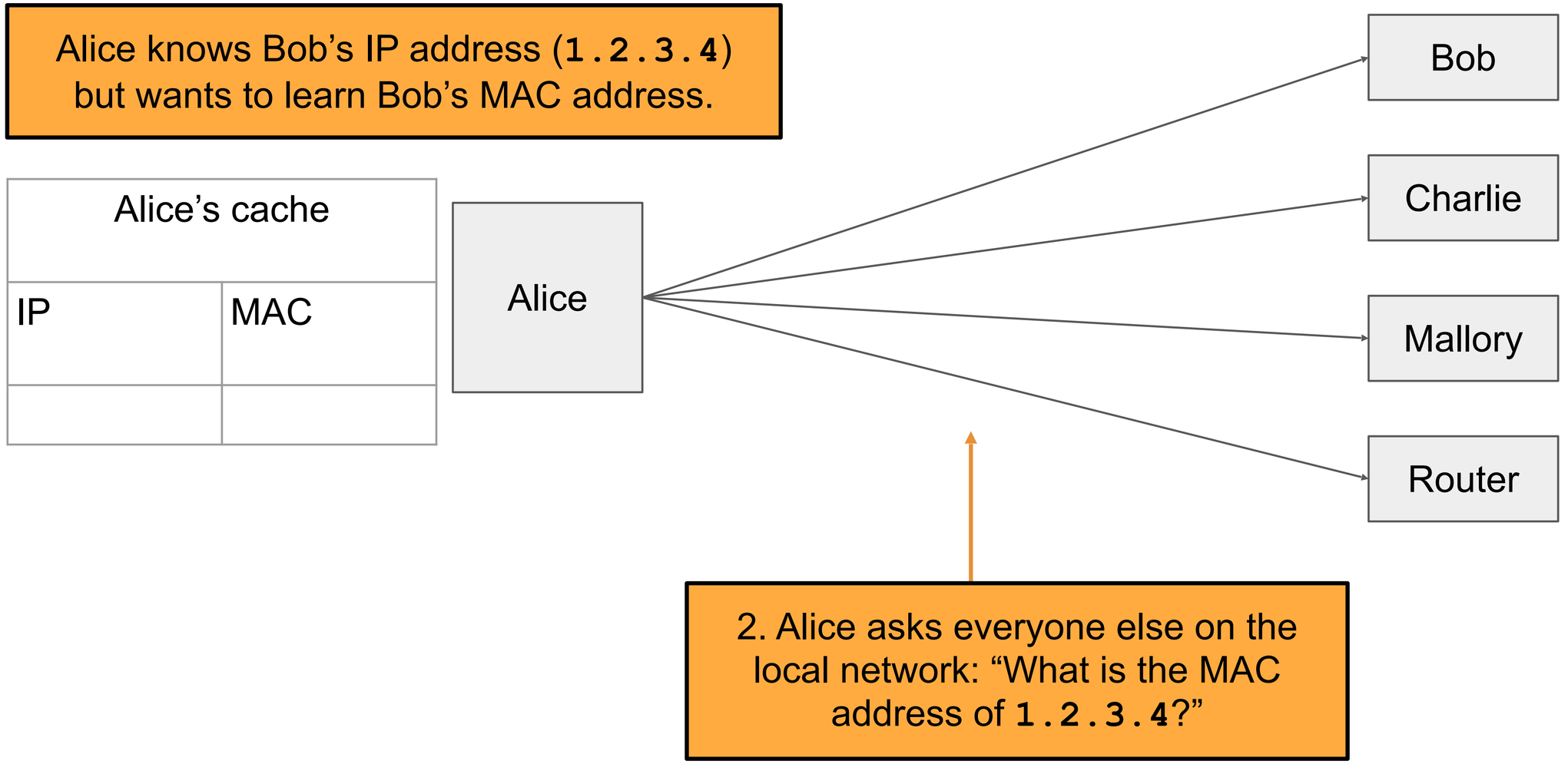
Mallory

Router

1. Alice checks her cache to see if she already knows the MAC address corresponding to `1.2.3.4`.

Since her cache is empty, she must make a request to find out.

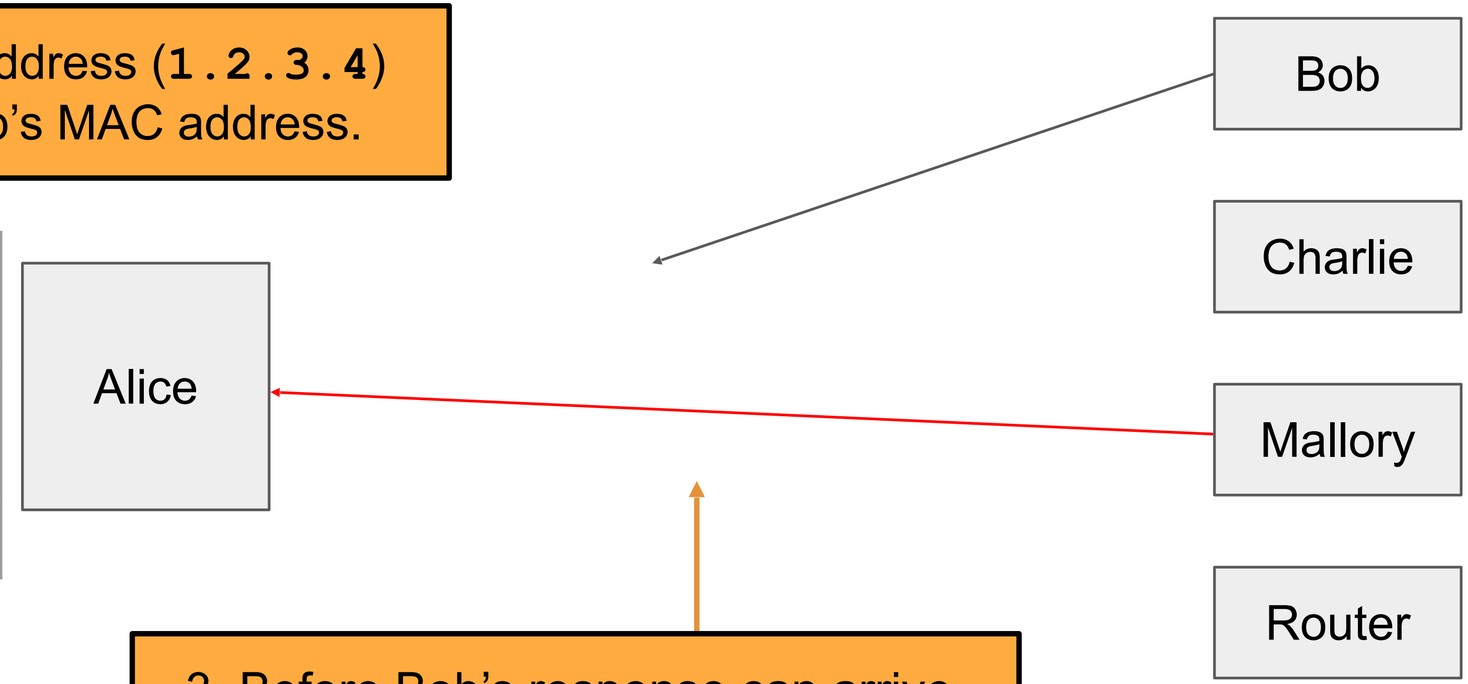
Attacks on ARP



Attacks on ARP

Alice knows Bob's IP address (`1.2.3.4`) but wants to learn Bob's MAC address.

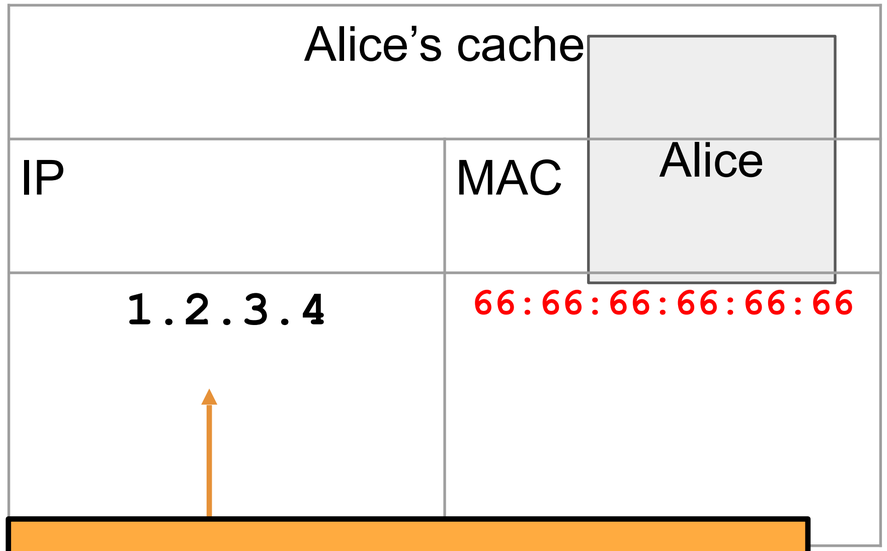
Alice's cache	
IP	MAC



3. Before Bob's response can arrive, Mallory sends a malicious response: "My IP is `1.2.3.4` and my MAC address is `66:66:66:66:66:66`."

Attacks on ARP

Alice knows Bob's IP address (`1.2.3.4`) but wants to learn Bob's MAC address.



4. Alice adds Mallory's malicious address to her cache.

Bob

Charlie

Mallory

Router

Attack: ARP Spoofing

- Alice has no way of verifying the ARP response
  - Spoofing: Any attacker on the network can claim to have the requested IP address
- Alice is only expecting one machine to respond, so she will accept the first response
  - **Race condition**: As long as the attacker responds faster, the requester will accept the attacker's response
- ARP spoofing requires Mallory to be in the same LAN as Alice
- ARP spoofing lets Mallory become a man-in-the-middle (MITM) attacker
  - When Alice sends a message to Bob, she is actually sending the message to Mallory
  - Mallory can modify the message and then send the modified message to Bob
  - Alice thinks that Bob's MAC address is `66:66:66:66:66:66` (Mallory's MAC address)

Encrypt-then-MAC or MAC-then-Encrypt?

- Encrypt-then-MAC
  - First compute Enc(K_1, M)
  - Then MAC the ciphertext: MAC(K_2, Enc(K_1, M))
- MAC-then-encrypt
  - First compute MAC(K_2, M)
  - Then encrypt the message and the MAC together: Enc(K_1, M || MAC(K_2, M))
- Which is better?
  - In theory, both are IND-CPA and EU-CPA secure if applied properly
  - MAC-then-encrypt has a downside: You don't know if tampering has occurred until after decrypting
    - Attacker can supply arbitrary tampered input, and you always have to decrypt it
    - Passing attacker-chosen input through the decryption function can cause side-channel leaks
- **Always use encrypt-then-MAC** because it's more robust to mistakes

Example: HMAC

- Issues with NMAC:
  - Recall: NMAC(K_1, K_2, M) = $H(K_1 \,||\, H\,(K_2 \,||\, M))$
  - We need two different keys
  - NMAC requires the keys to be the same length as the hash output (n bits)

- HMAC(K, M):
  - Compute K' as a version of K that is the length of the hash output
    - If K is too short, pad K with 0's to make it n bits (be careful with keys that are too short and lack randomness)
    - If K is too long, hash it so it's n bits
  - Output $H((K' \oplus opad) \,||\, H((K' \oplus ipad) \,||\, M))$