Networking CS 161 Computer Security Notes

Lecture notes by Nicholas Weaver, David Wagner,
Peyrin Kao, Andrew Law, and Sheqi Zhang

Last update: May 2, 2021

Contact for corrections: Peyrin Kao (peyrin at berkeley.edu)

Disclaimer: These notes are still in beta and haven't been thoroughly fact-checked. In any factual dispute, all other course material takes precedence. Any feedback is welcome.

1 Networking Background

To discuss network security, first we need to know how the network is designed. This section provides a (simplified) overview of the various Internet layers and how they interact. A video version of this section is available: see Lecture 11, Summer 2020.

1.1 Local Area Networks

The primary goal of the Internet is to move data from one location to another. A good analogy for the Internet is the postal system, which we'll refer to throughout this section.

The first building block we need is something that moves data across space, such as bits on a wire, radio waves, carrier pigeons, etc. Using our first building block, we can connect a group of local machines in a **local area network (LAN)**.

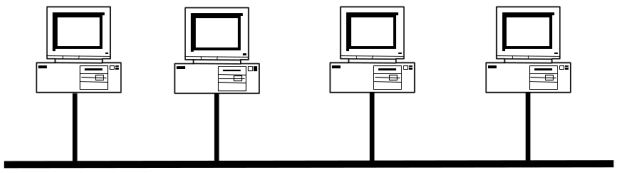

Figure 1: Computers connected in a local area network (LAN).

Note that in a LAN, all machines are connected to all other machines. This allows any machine on the LAN to send and receive messages from any other machine on the same LAN. You can think of a LAN as an apartment complex, a local group of nearby apartments that are all connected. However, it would be infeasible to connect every machine in the world to every other machine in the world, so we introduce a **router** to connect multiple LANs.

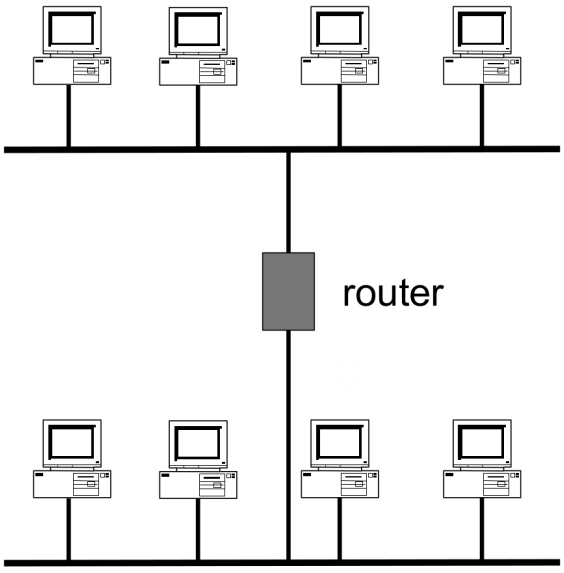

Figure 2: Two LANs connected through a router.

A router is a machine that is connected to two or more LANs. If a machine wants to send a message to a machine on a different LAN, it sends the message to the router, which forwards the message to the second LAN. You can think of a router as a post office: to send a message somewhere outside of your local apartment complex, you'd take it to the post office, and they would forward your message to the other apartment complex.

With enough routers and LANs, we can connect the entire world in a **wide area network**, which forms the basis of the Internet.

1.2 Internet layering

You may have noticed that this design uses layers of abstraction to build the Internet. The lowest layer (layer 1, also called the physical layer) moves bits across space. Then, layer 2 (the link layer) uses layer 1 as a building block to connect local machines in a LAN. Finally, layer 3 (the internetwork layer) connects many layer 2 LANs. Each layer relies on services from a lower layer and provides services to a higher layer. Higher layers contain richer information, while lower layers provide the support necessary to send the richer information at the higher layers.

This design provides a clean abstraction barrier for implementation. For example, a network can choose to use wired or wireless communication at Layer 1, and the Layer 1 implementation does not affect any protocols at the other layers.

In total, there are 7 layers of the Internet, as defined by the OSI 7-layer model. However, this model is a little outdated, so some layers are obsolete, and additional layers for security have been added since then. We will see these higher layers later.

7	Application
6.5	Secure Transport
6	*obsolete*
5	*obsolete*
4	Transport
3	(Inter)Network
2	Link
1	Physical

Figure 3: The OSI 7-layer model.

1.3 Protocols and Headers

Each layer has its own set of **protocols**, a set of agreements on how to communicate. Each protocol specifies how communication is structured (e.g. message format), how machines should behave while communicating (e.g. what actions are needed to send and receive messages), and how errors should be handled (e.g. a message timing out).

To support protocols, messages are sent with a **header**, which is placed at the beginning of the message and contains some metadata such as the sender and recipient's identities, the length of the message, identification numbers, etc. You can think of headers as the envelope of a letter: it contains the information needed to deliver the letter, and appears before the actual letter.

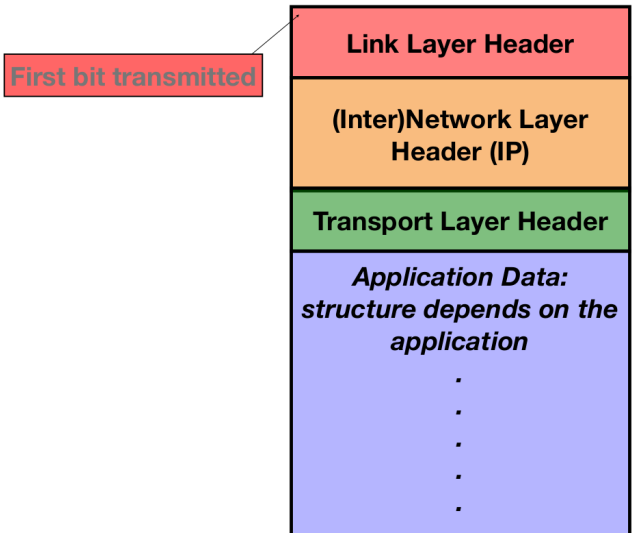

Figure 4: Multiple headers on a single packet.

Because multiple protocols across different layers are needed to send a message, we need multiple headers on each packet. Each message begins as regular human-readable text (the highest layer). As the message is being prepared to get sent, it is passed down the protocol stack to lower layers (similar to how C programs are passed to lower layers to translate C code to RISC-V to machine-readable bits). Each layer adds its own header to the top of the message provided from the layer directly above. When the message reaches the lowest layer,

it now has multiple headers, starting with the header for the lowest layer first.

Once the message reaches its destination, the recipient must unpack the message and decode it back into human-readable text. Starting at the lowest layer, the message moves up the protocol stack to higher layers. Each layer removes its header and provides the remaining content to the layer directly above. When the message reaches the highest layer, all headers have been processed, and the recipient sees the regular human-readable text from before.

1.4 Addressing: MAC, IP, Ports

Depending on the layer, a machine can be referred to by several different addresses.

Layer 2 (link layer) uses 48-bit (6-byte) **MAC addresses** to uniquely identify each machine on the LAN. This is not to be confused with MACs (message authentication codes) from the crypto section. Usually it is clear from context which type of MAC we are referring to, although sometimes cryptographic MACs are called MICs (message integrity codes) when discussing networking. MAC addresses are usually written as 6 pairs of hex numbers, such as `ca:fe:f0:0d:be:ef`. There is also a special MAC address, the broadcast address of `ff:ff:ff:ff:ff:ff`, that says "send this message to everyone on the local network." You can think of MAC addresses as apartment numbers: they are used to uniquely identify people within one apartment complex, but are useless for uniquely identifying one person in the world. (Imagine sending a letter addressed to "Apartment 5." This might work if you're delivering letters within your own apartment complex, but how many Apartment 5s exist in the entire world?)

Layer 3 (IP layer) uses 32-bit (4-byte) **IP addresses** to uniquely identify each machine globally. IP addresses are usually written as 4 integers between 0 and 255, such as `128.32.131.10`. Because the Internet has grown so quickly, the most recent version of the layer 3 protocol, IPv6, uses 128-bit IP addresses, which are written as 8 2-byte hex values separated by colons, such as `cafe:f00d:d00d:1401:2414:1248:1281:8712`. However, for this class, you only need to know about IPv4, which uses 32-bit IP addresses.

Higher layers are designed to allow each machine to have multiple processes communicating across the network. For example, your computer only has one IP address, but it may have multiple browser tabs and applications open that all want to communicate over the network. To distinguish each process, higher layers assign each process on a machine a unique 16-bit **port number**. You can think of port numbers as room numbers: they are used to uniquely identify one person in a building.

The source and destination addresses are contained in the header of a message. For example, the Layer 2 header contains MAC addresses, the Layer 3 header contains IP addresses, and higher layer headers will contain port numbers.

1.5 Packets vs. Connections

Notice that in the postal system example, the post office has no idea if you and your pen pal are having a conversation through letters. The Internet is the same: at the physical, link,

and internetwork layers, there is no concept of a connection. A router at the link layer only needs to consider each individual packet and send it to its destination (or, in the case of a long-distance message, forward it to another router somewhere closer to the destination). At the lower layers, we call individual messages **packets**. Packets are usually limited to a fixed length.

In order to actually create a two-way connection, we rely on higher layers, which maintain a connection by breaking up longer messages into individual packets and sending them through the lower layer protocols. Higher-layer connections can also implement cryptographic protocols for additional security, as we'll see in the TLS section.

Note that so far, the Internet design has not guaranteed any correctness or security. Packets can be corrupted in transit or even fail to send entirely. The IP (Internet Protocol) at layer 3 only guarantees *best-effort delivery*, and does not handle any errors. Instead, we rely on higher layers for correctness and security.

1.6 Network Adversaries

Network adversaries can be sorted into 3 general categories. They are, from weakest to strongest:

Off-path Adversaries: cannot read or modify any packets sent over the connection.

On-path Adversaries: can read, but not modify packets.

In-path Adversaries: can read, modify, and block packets. Also known as a **man-in-the-middle**.

Note that all adversaries can send packets of their own, including faking or **spoofing** the packet headers to appear like the message is coming from somebody else. This is often as simple as setting the "source" field on the packet header to somebody else's address.

2 Wired Local Networks: ARP

2.1 Cheat sheet

- Layer: Link (2)
- Purpose: Translate IP addresses to MAC addresses
- Vulnerability: On-path attackers can see requests and send spoofed malicious responses
- Defense: Switches, arpwatch

2.2 Networking background: Ethernet

Recall that on a LAN (local-area network), all machines are connected to all other machines. Ethernet is one particular LAN implementation that uses wires to connect all machines.

Ethernet started as a broadcast-only network. Each node on the network could see messages sent by all other nodes, either by being on a common wire or a network **hub**, a simple repeater that took every packet it received and rebroadcast it to all the outputs. A receiver is simply supposed to ignore all packets not sent to either the receiver's MAC or the broadcast address. But this is only enforced in software, and most Ethernet devices can enter **promiscuous mode**, where it will receive all packets. This is also called **sniffing packets**.

For versions of Ethernet that are inherently broadcast, such as a hub, an adversary in the local network can see all network traffic and can also introduce any traffic they desire by simply sending packets with a spoofed MAC address. Sanity check: what type of adversary does this make someone on the same LAN network as a victim?[1]

2.3 Protocol: ARP

ARP, the **Address Resolution Protocol**, translates Layer 3 IP addresses into Layer 2 MAC addresses.

Say Alice wants to send a message to Bob, and Alice knows that Bob's IP address is `1.1.1.1`. The ARP protocol would follow three steps:

1. Alice would broadcast to everyone else on the LAN: "What is the MAC address of `1.1.1.1`?"
2. Bob responds by sending a message only to Alice: "My IP is `1.1.1.1` and my MAC address is `ca:fe:f0:0d:be:ef`." Everyone else does nothing.
3. Alice caches the IP address to MAC address mapping for Bob.

If Bob is outside of the LAN, then the router would respond in step 2 with its MAC address.

Any received ARP replies are always cached, even if no broadcast request (step 1) was ever made.

[1] A: On-path

2.4 Attack: ARP Spoofing

Because there is no way to verify that the reply in step 2 is actually from Bob, it is easy to attack this protocol. If Mallory is able to create a spoofed reply and send it to Alice before Bob can send his legitimate reply, then she can convince Alice that a different MAC address (such as Mallory's) corresponds to Bob's IP address. Now, when Alice wants to send a local message to Bob, she will use the malicious cached IP address to MAC address mapping, which might map Bob's IP address to Mallory's MAC address. This will cause messages intended for Bob to be sent to Mallory. Sanity check: what type of adversary is Mallory after she executes an ARP spoof attack?[2]

ARP spoofing is our first example of a race condition, where the attacker's response must arrive faster than the legitimate response to fool the victim. This is a common pattern for on-path attackers, who cannot block the legitimate response and thus must race to send their response first.

2.5 Defenses: Switches

A simple defense against ARP spoofing is to use a tool like arpwatch, which tracks the IP address to MAC address pairings across the LAN and makes sure nothing suspicious happens.

Modern wired Ethernet networks defend against ARP spoofing by using **switches** rather than hubs. Switches have a MAC cache, which keeps track of the IP address to MAC address pairings. If the packet's IP address has a known MAC in the cache, the switch just sends it to the MAC. Otherwise, it broadcasts the packet to everyone. Smarter switches can filter requests so that not every request is broadcast to everyone.

Higher-quality switches include **VLAN**s (Virtual Local Area Networks), which implement isolation by breaking the network into separate virtual networks.

[2] A: Man-in-the-middle. She can receive messages from Alice, modify them, then send them to Bob.

3 Wireless Local Networks: WPA2

3.1 Cheat sheet

- Layer: Link (2)
- Purpose: Communicate securely in a wireless local network
- Vulnerability: On-path attackers can learn the encryption keys from the handshake and decrypt messages (includes brute-forcing the password if they don't know it already)
- Defense: WPA2-Enterprise

3.2 Networking background: WiFi

Another implementation of the link layer is WiFi, which wirelessly connects machines in a LAN. Because it wireless connections over cellular networks, WiFi has some differences from wired Ethernet, but these are out of scope for this class. For the purposes of this class, WiFi behaves mostly like Ethernet, with the same packet format and similar protocols like ARP for address translation.

To join a WiFi network, your computer establishes a connection to the network's **AP (Access Point)**. Generally the AP is continuously broadcasting beacon packets saying "I am here" and announcing the name of the network, also called the **SSID (Service Set Identifier)**. When you choose to connect to a WiFi network (or if your computer is configured to automatically join a WiFi network), it will broadcast a request to join the network.

If the network is configured without a password, your computer immediately joins the network, and all data is transmitted without encryption. This means that anybody else on the same network can see your traffic and inject packets, like in ARP spoofing.

3.3 Protocol

WPA2-PSK (WiFi Protected Access: Pre-Shared Key) is a protocol that enables secure communications over a WiFi network by encrypting messages with cryptography.

In WPA2-PSK, a network has one password for all users (this is the WiFi password you ask your friends for). The access point derives a **PSK (Pre-Shared Key)** by applying a password-based key derivation function (PBKDF2-SHA1) on the SSID and the password. Recall from the cryptography unit that password-based key derivation functions are designed to be slower by a large constant factor to make brute-force attacks more difficult. Sanity check: Why might we choose to include the SSID as input to the key derivation function?[3]

When a computer (client) wants to connect to a network protected with WPA2-PSK, the user must first type in the WiFi password. Then, the client uses the same key derivation

[3]By including the SSID, two different networks with the same password will still have different PSKs.

function to generate the PSK. Sanity check: Why can't we be done here and use the PSK to encrypt all further communications?[4]

To give each user a unique encryption key, after both the client and the access point independently derive the PSK, they participate in a handshake to generate shared encryption keys.

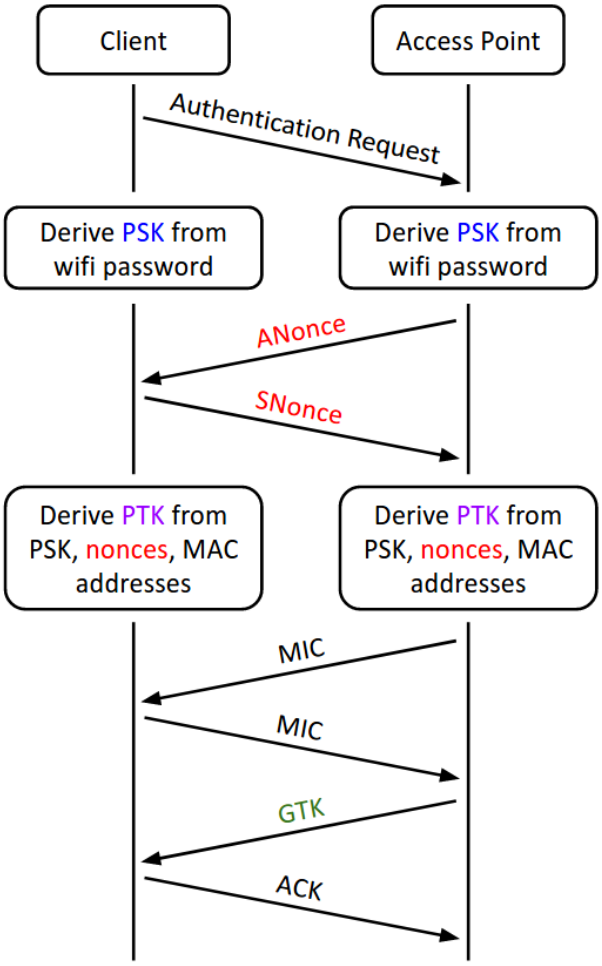

Figure 5: The WPA2 handshake.

1. The client and the access point exchange random nonces, the ANonce and the SNonce. The nonces ensure that different keys will be generated during each handshake. The nonces are sent without any encryption.

2. The client and access point independently derive the **PTK (Pairwise Transport Keys)** as a function of the two nonces, the PSK, and the MAC addresses of both the access point and the client.

3. The client and the access point exchange MICs (recall that these are MACs from the crypto unit) to check that no one tampered with the nonces, and that both sides

[4]Because everyone on the network would use the same PSK, so others on the same network can still decrypt your traffic.

correctly derived the PTK.

4. The access point encrypts the **GTK (Group Temporal Key)** and sends it to the client.
5. The client sends an ACK (acknowledgement message) to indicate that it successfully received the GTK.

Once the handshake is complete, all further communication between the client and the access point is encrypted with the PTK.

The GTK is used for messages broadcast to the entire network (i.e. sent to the broadcast MAC address, `ff:ff:ff:ff:ff:ff`). The GTK is the same for everyone on the network, so everyone can encrypt/send and decrypt/receive broadcast messages.

In practice, the handshake is optimized into a 4-way handshake, requiring only 4 messages to be exchanged between the client and the access point.

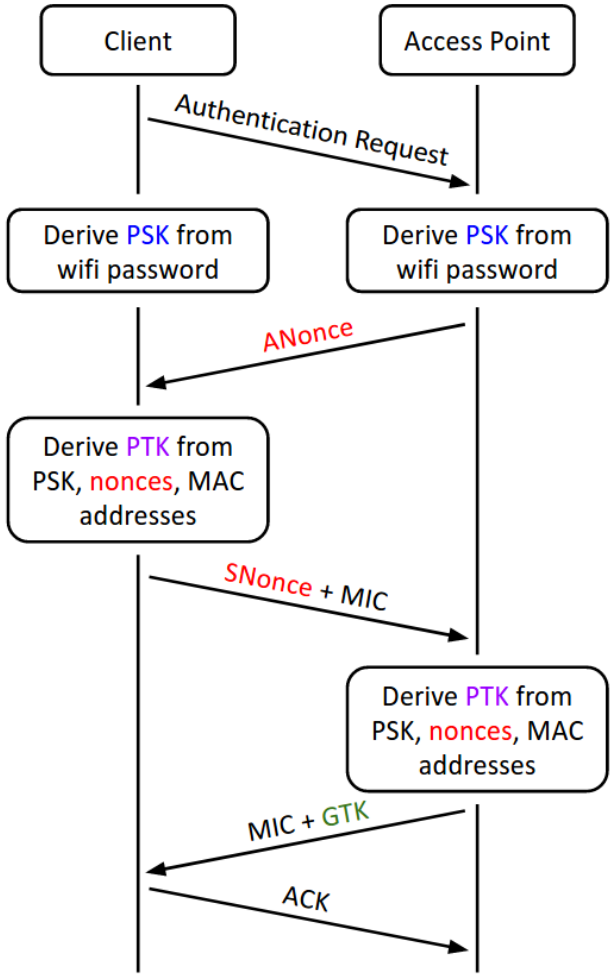

Figure 6: The optimized, 4-way WPA2 handshake.

1. The access point sends the ANonce, as before.

2. Once the client receives the ANonce, it has all the information needed to derive the PTK, so it derives the PTK first. Then it sends the SNonce and the MIC to the access point.
3. Once the access point receives the SNonce, it can derive the PTK as well. Then it sends the encrypted GTK and the MIC to the client.
4. The client sends an ACK to indicate that it successfully received the GTK, as before.

3.4 Attacks

In the WPA2 handshake, everything except the GTK is sent unencrypted. Recall that the PTK is derived with the two nonces, the PSK, and the MAC addresses of both the access point and the client. This means that an on-path attacker who eavesdrops on the entire handshake can learn the nonces and the MAC addresses. If the attacker is part of the WiFi network (i.e. they know the WiFi password and generated the PSK), then they know everything necessary to derive the PTK. This attacker can decrypt all messages and eavesdrop on communications, and encrypt and inject messages.

Even if the attacker isn't on the WiFi network (doesn't know the WiFi password and cannot generate the PSK), they can try to brute-force the WiFi password. For each guessed password, the attacker derives the PSK from that password, uses the PSK (and the other unencrypted information from the handshake) to derive the PTK, and checks if that PTK is consistent with the MICs. If the WiFi password is low-entropy, an attacker with enough compute power can brute-force the password and learn the PTK.

3.5 Defenses: WPA2-Enterprise

The main problem leading to the attacks in the previous section is that every user on the network uses the same secrecy (the WiFi password) to derive private keys. To solve this, each user needs an different, unique source of secrecy. This modified protocol is called **WPA2-Enterprise**. AirBears2 is an example of WPA2-Enterprise that you might be familiar with.

Instead of using one WiFi password for all users, WPA2-Enterprise gives authorized a unique username and password. In WPA2-Enterprise, before the handshake occurs, the client connects to a secure authentication server and proves its identity to that server by providing a username and password. (The connection to the authentication server is secured with TLS, which is covered in a later section.) If the username and password are correct, the authentication server presents both the client and the access point with a random **PMK (Pairwise Master Key)** to use instead of the PSK. The handshake proceeds as in the previous section, but it uses the PMK (unique for each user) in place of the PSK (same for all users) to derive the PTK.

WPA-2 defends against the attacks from the previous section, because the PMK is created randomly by a third-party authentication server and sent over encrypted channels to both the AP and the client. However, note that WPA2-Enterprise is still vulnerable against another authenticated user who executes an ARP or DHCP attack to become a man-in-the-middle.

4 DHCP

4.1 Cheat sheet

- Layer: 2-3 (see below)
- Purpose: Get configurations when first connecting to a network
- Vulnerability: On-path attackers can see requests and send spoofed malicious responses
- Defense: Accept as a fact of life and rely on higher layers

4.2 Protocol: DHCP

DHCP (**Dynamic Host Configuration Protocol**) is responsible for setting up configurations when a computer first joins a local network. These settings enable communication over LANs and the Internet, so it is sometimes considered a layer 2-3 protocol. The Internet layers are defined primarily for communication, so setup protocols like DHCP don't fit cleanly into the abstraction barriers in the layering model.

In order to connect to a network, you need a few things:

- An IP address, so other people can contact you
- The IP address of the DNS server, so you can translate a site name like www.google.com into an IP address (DNS is covered in more detail later)
- The IP address of the router (also called the **gateway**), so you can contact others on the Internet

The DHCP handshake follows four steps, between you (the client) and the server (who can give you the needed IP addresses)

1. **Client Discover**: The client broadcasts a request for a configuration.
2. **Server Offer**: Any server able to offer IP addresses responds with some configuration settings. (In practice, usually only one server replies here.)
3. **Client Request**: The client broadcasts which configuration it has chosen.
4. **Server Acknowledge**: The chosen server confirms that its configuration has been chosen.

The configuration information provided in step 2 (server offer) is sometimes called a **DHCP lease**. The offer may include a lease time. After the time expires, the client must ask to renew the lease to keep using that configuration, or else the DHCP server will free up those settings for other devices that request leases later.

Notice that both client messages are broadcast. Step 3 (client request) must be broadcast so that if multiple servers made offers in step 2, all the servers know which one has been

chosen. Sanity check: why must client discover be broadcast?[5]

4.3 Networking background: NAT

Because there are more computers than IPv4 addresses on the modern Internet, and not all networks support IPv6 (expanded address space) yet, DHCP supports **NAT (Network Address Translation)**, which allows multiple computers on a local network to share an IP address. When a computer requests a configuration through DHCP, the router (DHCP server) assigns that computer a placeholder IP address. This address usually comes from a reserved block of private IP addresses that are invalid on the Internet, but can be used as placeholders in the local network.

When a computer sends a packet to the Internet, the packet passes through the router first. The router stores a record mapping the internal (source) IP address to the remote (destination) IP address, for processing potential replies. Then the router replaces the placeholder IP address with a valid IP address, and sends the packet to the remote sever on the Internet. Sanity check: does this replacement happen for the source or destination IP address?[6] When the router sees an incoming packet, it checks the stored mappings, converts the destination IP address back to the correct placeholder address, and forwards the message to the original computer on the local network. With NAT, the router could potentially use a single valid IP address to send packets on behalf of every computer on the local network.

4.4 Attack

The attack on DHCP is almost identical to ARP spoofing. At the server offer step, an attacker can send a forged configuration, which the client will accept if it arrives before the legitimate configuration reply. The attacker could offer its own IP address as the gateway address, which makes the attacker a man-in-the-middle. Packets intended for the network would be sent to the attacker, who can modify them before forwarding them to the real gateway. The attacker can also become a man-in-the-middle by manipulating the DNS server address, which lets the attacker supply malicious translations between human-readable host names (www.google.com) and IP addresses (6.6.6.6).

4.5 Defenses

In reality, many networks just accept DHCP spoofing as a fact of life and rely on the higher layers to defend against attackers (the general idea: if the message sent is properly encrypted, the man-in-the-middle can't do anything anyway).

Defending against low-layer attacks like DHCP spoofing is hard, because there is no trusted party to rely on when we're first connecting to the network.

[5] A: Before DHCP, the client has no idea where the servers are.

[6] Source, since it's an outgoing packet.

5 IP Routing: BGP

5.1 Cheat sheet

- Layer: 3 (inter-network)
- Purpose: Send messages globally by connecting lots of local networks
- Vulnerability: Malicious local networks can read messages in intermediate transit and forward them to the wrong place
- Defense: Accept as a fact of life and rely on higher layers

5.2 Networking background: Subnets

Recall that IP addresses uniquely identify a single machine on the global network. (With NAT, the address could correspond to multiple machines, but this can be abstracted away when discussing IP.) When sending packets to a remote IP on a different local network, the packet must make many hops across many local networks before finally reaching its destination.

IP routes by "subnets", groups of addresses with a common prefix. A subnet is usually written as a prefix followed by the number of bits in the prefix. For example, `128.32/16` is an IPv4 subnet with all addresses beginning with the 16-bit prefix `128.32`. There are 2^{16} addresses on this prefix, because there are 32 - 16 = 16 bits not in the prefix. Sanity check: how many addresses are in the `128.32.131/24` subnet?[7] Routing generally proceeds on a subnet rather than individual IP basis.

There are some special reserved IP addresses and network blocks that do not represent machines and subnets. `127.0.0/24` and `::1` are "localhost", used to create 'network' connections to your own system. Also, `255.255.255.255` is the IPv4 broadcast address, sending to all computers within the local network.

When a client gets its configuration from DHCP, it is told its own IP address, the address of the gateway, and the size of the subnet it is on. To send a packet to another computer on the same local network, the client first verifies that the computer is on the same local network by checking that its IP address is in the same subnet (same IP prefix). Then, the client uses ARP to translate the IP address to a MAC address and directly sends the packet to that MAC address.

To send a packet to another computer on a different local network, the client sends the packet to the gateway, whose responsibility is to forward the packet towards the destination.

Past the gateway, the packet passes onto the general Internet, which is composed of many **ASs (Autonomous Systems)**, identified by unique **ASNs (Autonomous System Numbers)**. Each AS consists of one or more local networks managed by an organization, such as an Internet service provider (ISP), university, or business. Within each AS, packets can be

[7] 2^8. The prefix is 24 bits, so there are 32 - 24 = 8 bits not in the prefix.

routed by any mechanism the AS desires, usually involving a complicated set of preferences designed to minimize the AS's own cost.

When an AS receives a packet, it first checks if that packet's final destination is located within the AS. If the final destination is within the AS, it routes the packet directly to the final destination. Otherwise, it must forward the packet to another AS that is closer to the final destination.

5.3 Protocol: BGP

Routing between ASs on the Internet is determined by BGP (the Border Gateway Protocol). BGP operates by having each AS advertise which networks it is responsible for to its neighboring ASs. Then each neighbor advertises that they can process packets to that network and provides information about the AS path that the packets would follow. The process continues until the entire Internet is connected into a graph with many paths between ASs. If an AS has a choice between two advertisements, it will generally select the shortest path. Actual BGP path selection is a fair bit more complicated than described here, but is out of scope for this class (take CS 168 to learn more).

5.4 Attack: Malicious ASs

The biggest problem with BGP is that it operates on trust, assuming that all ASs are effectively honest. Thus an AS can lie and say that it is responsible for a network it isn't, resulting in all traffic being redirected to the lying AS. There are further enhancements that allow a lying AS to act as a full man-in-the-middle, routing all traffic for a destination through the rogue AS.

Recall that IP operates on "best effort". Packets are delivered whole, but can be delivered in any order and may be corrupted or not sent at all. IPv4 and lower layers usually include checksums or CRC checks designed to detect corrupted packets. Sanity check: Why do the checksums not prevent a malicious AS from modifying packets?[8]

5.5 Defenses

In practice, there's not much anyone can do to defend against a malicious AS, since each AS operates relatively independently. Instead, we rely on protocols such as TCP at higher layers to guarantee that messages are sent. TCP will resend packets that are lost or corrupted because of malicious ASs. Also, cryptographic protocols at higher layers such as TLS can defend against malicious attackers, by guaranteeing confidentiality (attacker can't read the packets) and integrity (attacker can't modify the packets without detection) on packets. Both TCP and TLS are covered in later sections.

[8] Checksums are not cryptographic. The malicious AS could modify the packet and create a new checksum for the modified packet.

6 Transport Layer: TCP, UDP

6.1 Cheat sheet

- Layer: 4 (transport)
- Purpose: Establish connections between individual processes on machines (TCP and UDP). Guarantee that packets are delivered successfully and in the correct order (TCP only).
- Vulnerability: On-path and MITM attackers can inject data or RST packets. Off-path attackers must guess the 32-bit sequence number to inject packets.
- Defense: Rely on cryptography at a higher layer (TLS). Use randomly generated sequence numbers to stop off-path attackers.

6.2 Networking background: Ports

Recall that IP, the layer 3 (inter-network) protocol, is a best-effort protocol, meaning that packets can be corrupted, reordered, or dropped entirely. Also, IP addresses uniquely identify machines, but do not support multiple processes on one machine using the network (e.g. multiple browser tabs, multiple applications).

The transport layer solves the problem of multiple processes by introducing **port numbers**. Each process on a machine that wants to communicate over the network uses a unique 16-bit port number. Recall that port numbers are unique per machine, but cannot be used for global addressing–two machines can have processes with the same port number. However, an IP address and a port number together uniquely identify one process on one machine.

On client machines, such as your laptop, port numbers can be arbitrarily assigned. As long as each application uses a different port number, incoming packets can be sorted by port number and directed to the correct application. However, server machines offering services over the network need to use constant, well-known port numbers so client machines can send requests to those port numbers. For example, web servers always receive HTTP requests at port 80, and HTTPS (secure) requests at port 443. Ports below 1024 are "reserved" ports: only a program running as root can receive packets at those ports, but anyone can send packets to those ports.

The transport layer has 2 main protocols to choose from: TCP guarantees reliable, in-order packet delivery, while UDP does not. Both protocols use port numbers to support communication between processes. The choice of protocol depends on the context of the application.

6.3 Protocol: UDP

UDP (user datagram protocol) is a best-effort transport layer protocol. With UDP, applications send and receive discrete packets, and packets are not guaranteed to arrive, just

like in IP. It is possible for datagrams to be larger than the underlying network's packet size, but this can sometimes introduce problems.

The UDP header contains 16-bit source and destination port numbers to support communication between processes. The header also contains a checksum (non-cryptographic) to detect corrupted packets.

16 bits	16 bits
Source port	Destination port
Length	Checksum

Figure 7: The UDP packet header.

6.4 Protocol: TCP

TCP (Transmission Control Protocol) is a reliable, in-order, connection-based stream protocol. In TCP, a client first establishes a connection to the server by performing a handshake. Once established, the connection is reliable and in order: TCP handles resending dropped packets until they are received on the other side and rearranging any packets received out of order. TCP also handles breaking up long messages into individual packets, which lets programmers think in terms of high-level, arbitrary-length bytestream connections and abstract away low-level, fixed-size packets.

Like UDP, the TCP header contains 16-bit source and destination port numbers to support communication between processes, and a checksum to detect corrupted packets. Additionally, a 32-bit **sequence number** and a 32-bit **acknowledgment (ACK) number** are used for keeping track of missing or out-of-order packets. Flags such as SYN, ACK, and FIN can be set in the header to indicate that the packet has some special meaning in the TCP protocol.

16 bits	16 bits
Source port	Destination port
Sequence Number	
Acknowledgment Number	
Flags	Checksum

Figure 8: The TCP packet header.

A unique TCP connection is identified by a 5-tuple of (Client IP Address, Client Port, Server IP Address, Server Port, Protocol), where protocol is always TCP. In other words, a TCP connection is a sequence of back-and-forth communications between one port on one IP address, and another port on another IP address.

TCP communication works between any two machines, but it is most commonly used between a **client** requesting a service (such as your computer) and a **server** providing the service. To provide a service, the server waits for connection requests (sometimes called listening for requests), usually on a well-known port. To request the service, the client makes a connection request to that server's IP address and well-known port.

A TCP connection consists of two bytestreams of data: one from the client to the server, and one from the server to the client. The data in each stream is indexed using sequence numbers. Since there are two streams, there are two sets of sequence numbers in each TCP connection, one for each bytestream.

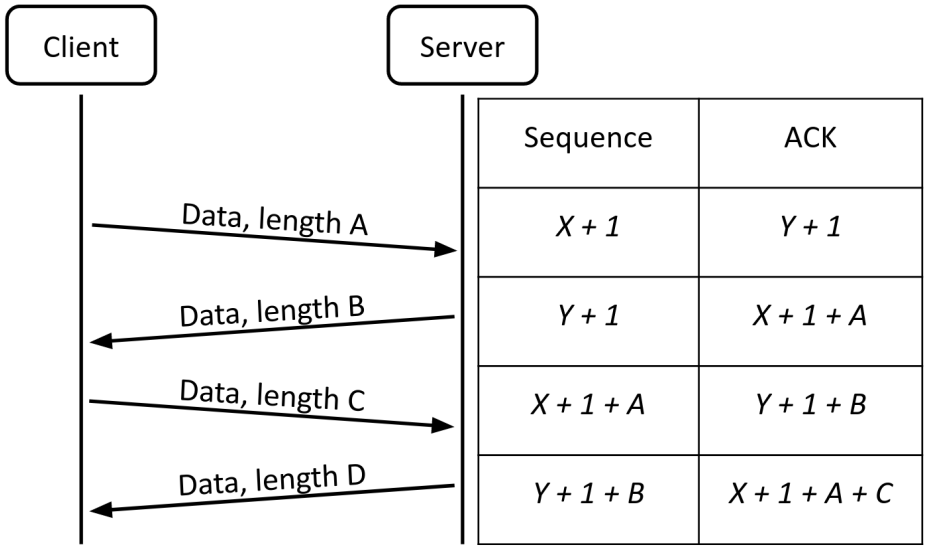

Figure 9: TCP communication.

In every TCP packet, the sequence number field in the header is set to the index of the first byte sent in that packet. In packets from the client to the server, the sequence number is an index in the client-to-server bytestream, and in packets from the server to the client, the sequence number is an index in the server-to-client bytestream. If packets are reordered, the end hosts can use the sequence numbers to reconstruct the message in the correct order.

To ensure packets are successfully delivered, when one side receives a TCP packet, it must reply with an acknowledgment saying that it received the packet. If the packet was dropped in transit, the recipient will never send an acknowledgment, and after a timeout period, the sender will re-send that packet.

If the packet is delivered, but the acknowledgment is dropped in transit, the sender will notice that it never received an acknowledgment and will re-send the packet. The recipient will see a duplicate packet (since the original packet was delivered), discard the duplicate, and re-send the acknowledgment.

Sending acknowledgment packets is wasteful in a two-way communication, so TCP combines acknowledgment packets with data packets. Each TCP packet can contain both data and an acknowledgment that a previous packet was received.

To support acknowledgments, the acknowledgment (ACK) number in the header is set to the index of the last byte received, plus 1. (This is equivalent to the index of the next byte the sender expects to receive.) In other words, in packets from the client to the server, the ACK number is the next unsent byte in the server-to-client stream, and in packets from the server to the client, the ACK number is the next unsent byte in the client-to-server stream.

Note that in each packet, the sequence number is an index in the sender's bytestream, and the ACK number is an index in the recipient's bytestream.

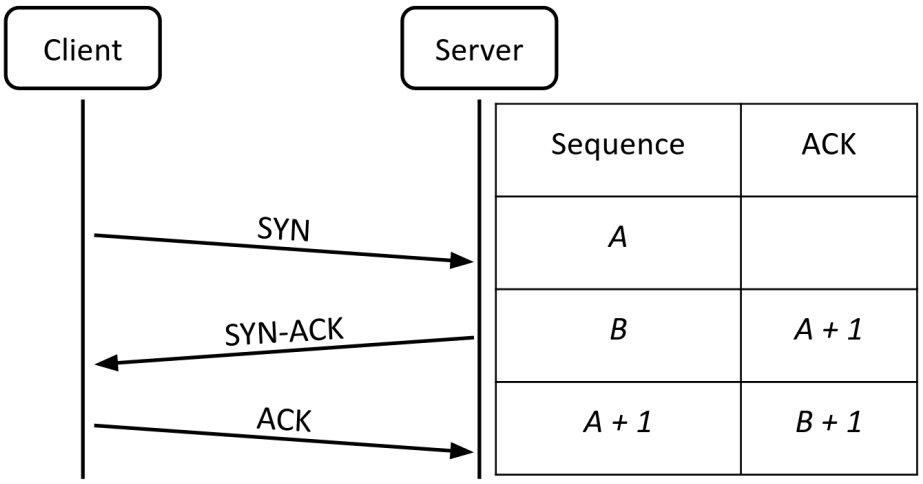

Figure 10: The TCP handshake.

Note that the sequence numbers do not start at 0 (for a security reason discussed below). Instead, to initiate a connection, the client and server participate in a three-way **TCP handshake** to exchange random initial sequence numbers.

1. The client sends a **SYN packet** (a packet with no data and the SYN flag set) to the server. The client sets the sequence number field to a random 32-bit **initial sequence number (ISN)**.
2. If the server decides to accept the request, it sends back a **SYN-ACK packet** (a packet with no data and both the SYN flag and ACK flag set). The server sets the sequence number field to its own random 32-bit initial sequence number (note that this is different from the client's ISN). The acknowledgment number is set to the client's initial sequence number + 1.
3. The client responds with an **ACK packet** (a packet with no data and the ACK flag set). The sequence number is set to the client's initial sequence number + 1, and the acknowledgement number is set to the server's initial sequence number + 1.

To end a connection, one side sends a FIN (a packet with the FIN flag set), and the other side replies with a FIN-ACK. This indicates that the side that sent the FIN will not send any more data, but can continue accepting data. This leaves the TCP connection in a "half closed" state, where one side stops sending but will receive and acknowledge further information. When the other side is done, it sends its own FIN as well, and it is acknowledged with a FIN-ACK reply.

Connections can also be unilaterally aborted. If one side sends a RST packet with a proper sequence number, this tells the other side that "I won't send any more data on this connection and I won't accept any more data on this connection." Unlike FIN packets, RST packets are not acknowledged. A RST usually indicates something went wrong, such as a program crashing or abruptly terminating a connection.

6.5 Tradeoffs between TCP and UDP

TCP is slower than UDP, because it requires a 3-way handshake at the start of each connection, and it will wait indefinitely for dropped packets to resent. However, TCP provides better correctness guarantees than UDP.

UDP is generally used when speed is a concern. For example, DNS requires extremely short response times, so it uses UDP instead of TCP at the transport layer. Video games and voice applications often use UDP because it is better to just miss a request than to stall everything waiting for a retransmission.

6.6 Attack: TCP Packet Injection

The main attack in TCP is **packet injection**. The attacker spoofs a malicious packet, filling in the header so that the packet looks like it came from someone in the TCP connection.

A related attack is **RST injection**. Instead of sending a packet with malicious data, the attacker sends a packet with the RST flag, causing the connection to abruptly terminate. This attack is useful for censorship: for example, Comcast used RST injection to abruptly terminate BitTorrent uploads.

Recall that that there are three types of network attackers. Each one has different capabilities in attacking the TCP protocol.

Off-path Adversary: The off-path adversary cannot read or modify any messages over the connection. Therefore, to attack a TCP communication, an off-path adversary must know or guess the values of the client IP, client port, server IP, and server port. Usually, the server IP address and port are well-known. Whether we know the client IP or port depends on our threat model. The off-path attacker must also guess the sequence number to inject a packet into the communication, because if the sequence number is too far off from what the recipient is expecting, it will reject the spoofed packet. Sanity check: What is the approximate probability of correctly guessing a random sequence number?[9]

On-path Adversary: The on-path adversary can read, but not modify messages. Since they can read the sequence numbers, IP addresses, and ports being used in the connection, an on-path adversary can inject messages into a TCP connection without guessing any values. As a concrete example, assume Alice has just sent a packet to Bob with sequence number X, and Bob responds with a packet of his own with sequence number Y and ACK X + 1. An on-path adversary Mallory wants to inject data into this TCP connection. While she cannot stop Alice from responding (because Mallory is not a man-in-the-middle), Mallory can race Alice's next packet with her own, using sequence number X + 1, ACK Y + 1, and Alice's IP and port. Since TCP on its own does not provide integrity, Bob will not be able to distinguish which message actually came from Alice, and which one came from Mallory.

In-path Adversary: The in-path (man-in-the-middle) adversary has all the powers of the on-path adversary and can additionally modify and block messages sent by either party. As

[9] The sequence number is 32 bits, so guessing a random sequence number succeeds with probability $1/2^{32}$.

a result, the same attack as the on-path adversary outlined above applies, and in addition, the in-path adversary doesn't have to race the party they are spoofing. A man in the middle can just block the message from ever arriving to the other party and send their own.

6.7 Defenses: TLS, random initial sequence numbers

The main problem here is that TCP by itself provides no confidentiality or integrity guarantees. To prevent injections like these, we rely on TLS, which is a higher-layer protocol that secures TCP communication with cryptography.

One important defense against off-path attackers is using random, unpredictable initial sequence numbers. This forces the off-path attacker to guess the correct sequence number with very low probability.

7 TLS

TLS (**Transport Layer Security**) is a protocol that provides an end-to-endd encrypted communication channel. (You may sometimes see **SSL**, which is the old, deprecated version of TLS.) **End-to-end encryption** guarantees that even if any one part of the communication chain is compromised (for example, if the packet passes through a malicious AS), no one except the sender and receiver is able to read or modify the data being sent.

The original OSI 7-layer model did not consider security, so TLS is usually referred to as a layer 6.5 protocol. It is built on top of layer 4 TCP (layers 5 and 6 are obsolete), and it is used to provide secure communications to layer 7 applications. Examples of applications that use TLS are HTTP, which is renamed HTTPS if TLS is used; SMTP (Simple Mail Transport Protocol), which uses the STARTTLS command to enable TLS on emails; and **VPN** (Virtual Private Network) connections, which encrypt the user's traffic.

TLS relies on TCP to guarantee that messages are delivered reliably in the proper order. From the application viewpoint, TLS is effectively just like a TCP connection with additional security guarantees.

7.1 TLS Handshake

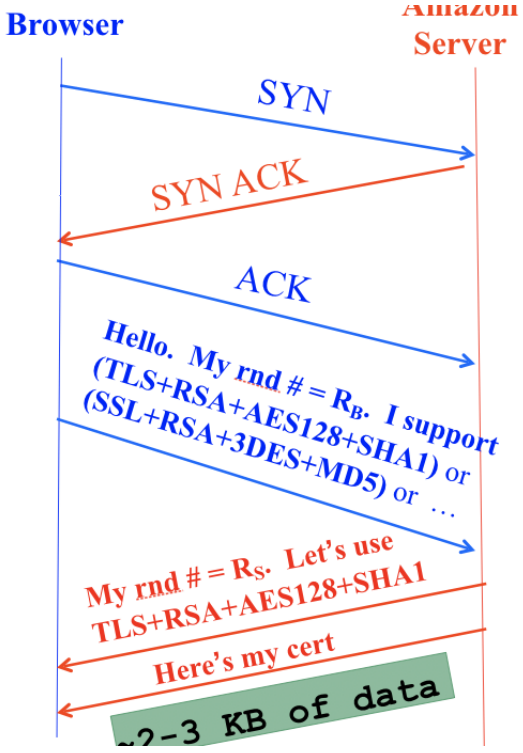

Figure 11: Part 1 of TLS handshake.

Because it's built on top of TCP, the TLS handshake starts with a TCP handshake. This lets us abstract away the notion of best-effort, fixed-size packets and think in terms of reliable messages for the rest of the TLS protocol.

The first message, `ClientHello`, presents a random number R_B and a list of encryption protocols it supports. The client can optionally also send the name of the server it actually wants to contact.

The second message, `ServerHello`, replies with its own random number R_S, the selected encryption protocol, and the server's **certificate**, which contains a copy of the server's public key signed by a **certificate authority** (**CA**).

If the client trusts the CA signing the certificate (e.g. that CA is included in the Chrome browser's pinned list of trusted CAs), then the client can use the signature to verify the server's public key is correct. If the client doesn't directly trust the CA, it may need to verify a chain of certificates in a PKI until it reaches the trusted root of the certificate chain. Either way, the client now has a trusted copy of the server's public key.

What is the public key being sent here? Every server implementing TLS must maintain a public/private key pair in order to support the PS exchange step you'll see next. We will assume that only the server knows the private key - if an attacker steals the private key, they would be able to impersonate the server, and the security guarantees no longer hold.

Sanity check: After the first two messages, can the client be certain that it is talking to the genuine server and not an impostor?[10]

The next step in TLS is to generate a random **Premaster Secret** (**PS**) known to only the client and the server. The PS should be generated so that no eavesdropper can determine the PS based on the data sent over the connection, and no one except the client and the legitimate server have enough information to derive the PS.

The first way to derive a shared PS is to encrypt it with RSA, show in the second arrow here:

[10] A: No. An attacker can obtain the genuine server's certificate by starting its own TLS connection with the genuine server, and then present a copy of that certificate in step 2.

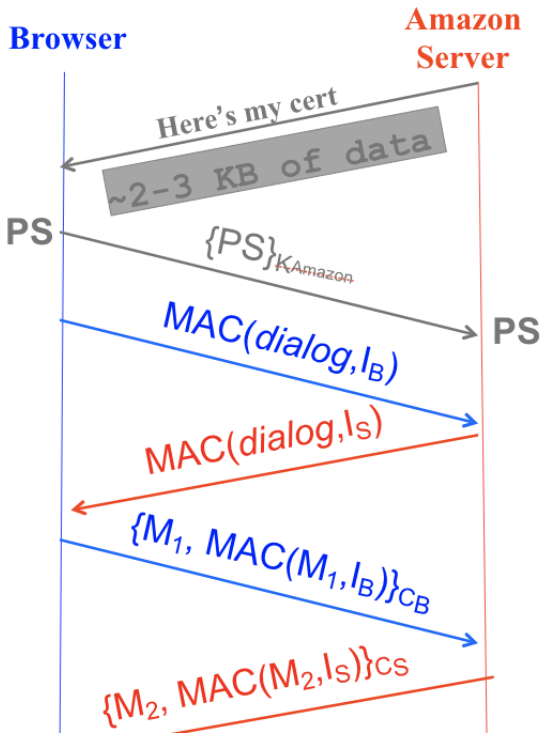

Figure 12: Part 2 of TLS handshake (RSA version).

Here, the client generates the random PS, encrypts it with the server's public key, and sends it to the server, which decrypts using its private key.

Sanity check: How can the client be sure it's using the correct public key?[11]

We can verify that this method satisfies all the properties of a PS. Because it is encrypted when sent across the channel, no eavesdropper can decrypt and figure out its value. Also, only the legitimate server will be able to decrypt the PS (using its secret key), so only the client and the legitimate server will know the value of the PS.

The second way to generate a PS is to use Diffie-Hellman key exchange, shown in the second (red) and third (blue) arrows here:

[11] A: It was signed by a certificate authority in the previous step.

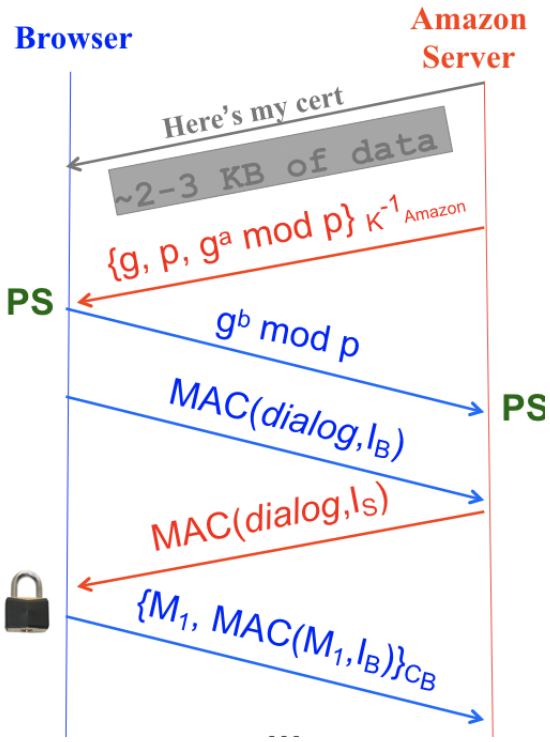

Figure 13: Part 2 of TLS handshake (Diffie-Hellman version).

The exchange looks just like classic Diffie-Hellman, except the server signs its half of the exchange with its secret key. The shared PS is the result of the key exchange, $g^{ab} \bmod p$.

Again, we can verify that this satisfies the properties of a PS. Diffie-Hellman's security properties guarantee that eavesdroppers cannot figure out PS, and no one but the client and the server know PS. We can be sure that the server is legitimate because the server's half of the key exchange is signed with its secret key.

An alternate implementation here is to use Elliptic Curve Diffie-Hellman (ECDHE). The specifics are out of scope, but it provides the same guarantees as regular DHE using elliptic curve math.

Generating the PS with DHE and ECDHE has a substantial advantage over RSA key exchange, because it provides **forward secrecy**. Suppose an attacker records lots of RSA-based TLS communications, and some time in the future manages to steal the server's private key. Now the attacker can decrypt PS values sent in old connections, which violates the security of those old TLS connections.

On the other hand, if the attacker steals the private key of a server using DHE or ECDHE-based TLS, they have no way of discovering the PS values of old connections, because the secrets required to generate the PS (a, b) cannot be discovered using the data sent over the connection ($g^a, g^b \bmod p$). Starting from TLS 1.3, RSA key exchanges are no longer allowed for this reason.

Now that both client and server have a shared PS, they will each use the PS and the random values R_B and R_S to derive a set of four shared symmetric keys: an encryption key C_B and

an integrity key I_B for the client, and an encryption key C_S and an integrity key I_S for the server.

Up until now, every message has been sent in plaintext over TLS. Sanity check: how might this be vulnerable?[12]

In order to ensure no one has tampered with the messages sent in the handshake so far, the client and server exchange and verify MACs over all messages sent so far. Notice that the client uses its own integrity key I_B to MAC the message, and the server uses its own integrity key I_S. However, both client and server know the value of I_B and I_S so that they can verify each other's MACs.

At the end of a proper TLS handshake, we have several security guarantees. (Sanity check: where in the handshake did these guarantees come from?)

1. The client is talking to the legitimate server.
2. No one has tampered with the handshake.
3. The client and server share a set of symmetric keys, unique to this connection, that no one else knows.

Once the handshake is complete, messages are encrypted and MAC'd with the encryption and integrity keys of the sender before being sent. Because these messages have full confidentiality and integrity, TLS has achieved end-to-end security between the client and the server.

7.2 Replay attacks

Recall that a **replay attack** involves an attacker recording old messages and sending them to the server. Even though the attacker doesn't know what these messages decrypt to, if the protocol doesn't properly defend against replay attacks, the server might accept these messages as valid and allow the attacker to spoof a connection.

The public values R_B and R_S at the start of the handshake defend against replay attacks. To see why, let's assume that $R_B = R_S = 0$ every time and try to execute a replay attack on RSA-based TLS. Since the attacker is sending the same encrypted PS, and R_B and R_S are not changing, the server will re-generate the same symmetric keys. Now the attacker can replay messages from the old TLS connection, which will be accepted by the server because they have the correct MACs. Using new, randomly generated values R_B and R_S every time ensures that each connection results in a different set of symmetric keys, so replay attacks trying to establish a new connection with the same keys will fail.

What about a replay attack within the same connection? In practice, messages sent over TLS usually include some counter or timestamp so that an attacker cannot record a TLS message and send it again within the same connection.

[12] A: TCP is insecure against on-path and MITM attackers, who can spoof messages.

7.3 TLS in practice

The biggest advantage and problem of TLS is the certificate authorities. "Trust does not scale", that is, you personally can't make trust decisions about everyone, but trust can be delegated, which is how TLS operates. We have delegated to a large number of companies, the **Certificate Authorities**, the responsibility of proving that a particular public key can speak for a particular site. This is what allows the system to work at all. But at the same time, unless additional measures are taken, this means that all CAs need to be trusted to speak for every site. This is why Chrome, for example, has a "pinned" CA list, so only some CAs are allowed to speak for certain websites.

Similarly, newer CAs implement **certificate transparency**, a mechanism where anyone can see all the certificates the CA has issued, implemented as a hash chain. Such CAs may issue a certificate incorrectly, but the impersonated victim can at least know this has happened. Certificates also expire and can be **revoked**, where a list of no-longer accepted certificates is published and regularly downloaded by a web browser or an online-service provides a mechanism to check if a particular certificate is revoked.

These days TLS is effectively free. The computational overhead is minor to the point of trivial: an ECDSA signature and ECDHE key exchange for the server, and such signatures and key exchanges are computationally minor: a single modern processor core can do tens of thousands of signatures or key exchanges per second. And once the key exchange is completed the bulk encryption is nearly free as most processors include routines specifically designed to accelerate AES.

This leaves the biggest cost of TLS in managing the private keys. Previously CAs charged a substantial amount to issue a certificate, but LetsEncrypt costs nothing because they have fully automated the process. You run a small program on your web server that generates keys, sends the public key to LetsEncrypt, and LetsEncrypt instructs that you put a particular file in a particular location on your server, acting to prove that you control the server. So LetsEncrypt has reduced the cost in two ways: It makes the TLS certificate monetarily free and, as important, makes it very easy to generate and use.

8 DNS

The Internet is commonly indexed in two different ways. Humans refer to websites using human-readable names such as `google.com` and `eecs.berkeley.edu`, while computers refer to websites using IP addresses such as `172.217.4.174` and `23.195.69.108`. **DNS**, or the **Domain Name System**, is the protocol that translates between the two.

8.1 Name servers

It would be great if there was single server that stored a mapping from every domain to every IP address that everyone could query, but unfortunately, there is no server big enough to store the IP address of every domain on the Internet and fast enough to handle the volume of DNS requests generated by the entire world. Instead, DNS uses a collection of many **name servers**, which are servers dedicated to replying to DNS requests.

Each name server is responsible for a specific zone of domains, so that no single server needs to store every domain on the Internet. For example, a name server responsible for the `.com` zone only needs to answer queries for domains that end in `.com`. This name server doesn't need to store any DNS information related to `wikipedia.org`. Likewise, a name server responsible for the `berkeley.edu` zone doesn't need to store any DNS information related to `stanford.edu`.

Even though it has a special purpose (responding to DNS requests), a name server is just like any other server you can contact on the Internet–each one has a human-readable domain name (e.g. `a.edu-servers.net`) and a computer-readable IP address (e.g. `192.5.6.30`). Be careful not to confuse the domain name with the zone. For example, this name server has `.net` in its domain, but it responds to DNS requests for `.edu` domains.

8.2 Name server hierarchy

You might notice two problems with this design. First, the `.com` zone may be smaller than the entire Internet, but it is still impractical for one name server to store all domains ending in `.com`. Second, if there are many name servers, how does your computer know which one to contact?

DNS solves both of these problems by introducing a new idea: when you query a name server, instead of always returning the IP address of the domain you queried, the name server can also direct you to another name server for the answer. This allows name servers with large zones such as `.edu` to redirect your query to other name servers with smaller zones such as `berkeley.edu`. Now, the name server for the `.edu` zone doesn't need to store any information about `eecs.berkeley.edu`, `math.berkeley.edu`, etc. Instead, the `.edu` name server stores information about the `berkeley.edu` name server and redirects requests for `eecs.berkeley.edu`, `math.berkeley.edu`, etc. to a `berkeley.edu` name server.

DNS arranges all the name servers in a tree hierarchy based on their zones:

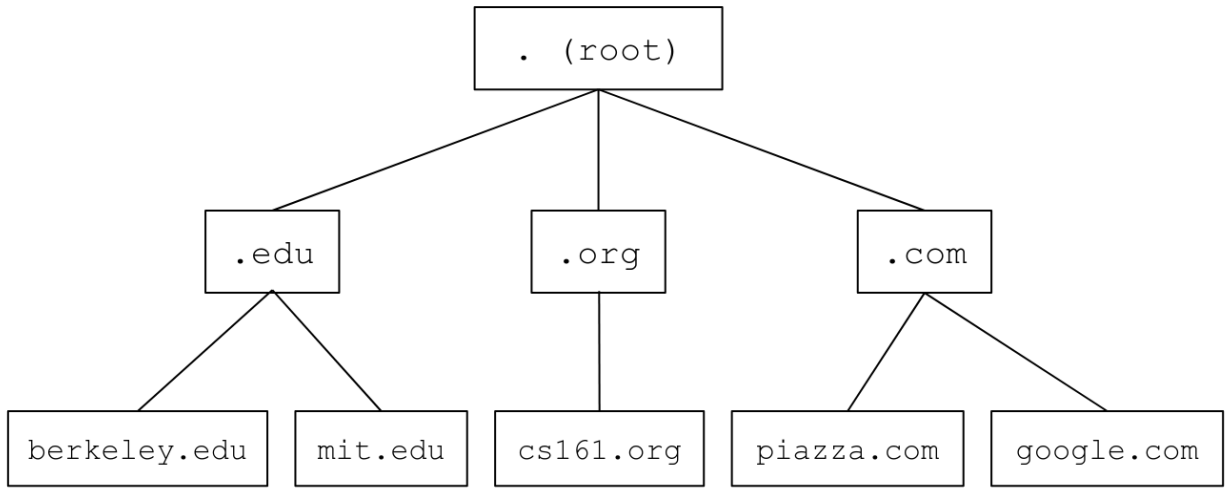

Figure 14: DNS name server hierarchy.

The **root server** at the top level of the tree has all domains in its zone (this zone is usually written as `.`). Name servers at lower levels of the tree have smaller, more specific zones. Each name server is only responsible for storing information about their children, except for the name servers at the bottom of the tree, which are responsible for storing the actual mappings from domain names to IP addresses.

DNS queries always start at the root. The root will direct your query to one of its children name servers. Then you make a query to the child name server, and that name server redirects you to one of its children. The process repeats until you make a query to a name server at the bottom of the tree, which will return the IP address corresponding to your domain.

To redirect you to a child name server, the parent name server must provide the child's zone, human-readable domain name, and IP address, so that you can contact that child name server for more information.

As an example, a DNS query for `eecs.berkeley.edu` might have the following steps. (A comic version of this query is available at https://howdns.works/.)

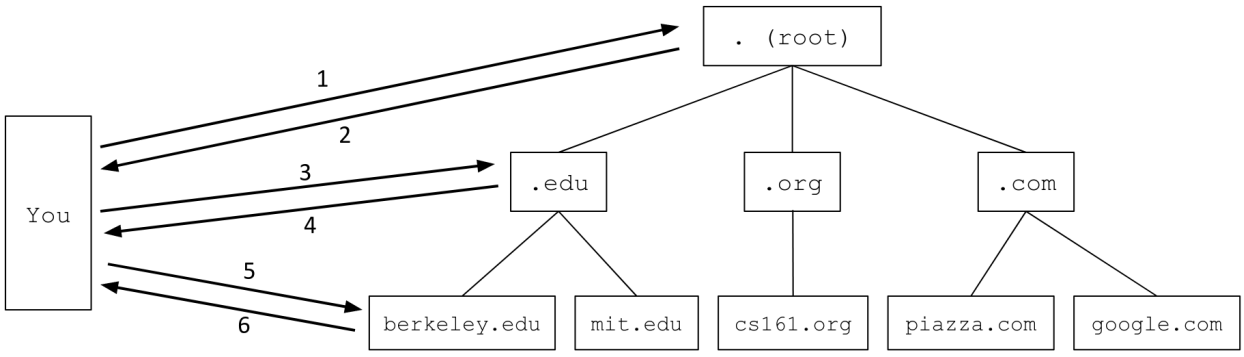

Figure 15: Steps of a DNS query.

1. You to the root name server: Please tell me the IP address of `eecs.berkeley.edu`.
2. Root server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `.edu` zone. It has human-readable domain name `a.edu-servers.net` and IP address `192.5.6.30`.
3. You to the `.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
4. The `.edu` name server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `berkeley.edu` zone. It has human-readable domain name `adns1.berkeley.edu` and IP address `128.32.136.3`.
5. You to the `berkeley.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
6. The `berkeley.edu` name server to you: OK, the IP address of `eecs.berkeley.edu` is `23.185.0.1`.

A note on who is actually sending the DNS queries in this example: Your computer can manually perform DNS lookups, but in practice, your local computer usually delegates the task of DNS lookups to a **DNS Recursive Resolver** provided by your Internet service provider (ISP), which sends the queries, processes the responses, and maintains an internal cache of records. When performing a lookup, the **DNS Stub Resolver** on your computer sends a query to the recursive resolver, lets it do all the work, and receives the response. When thinking about DNS requests, you can usually focus on the messages being sent between the recursive resolver and the name server.

Congratulations, you now understand how DNS translates domains to IP addresses! The rest of this section describes the specific implementation details of DNS.

8.3 DNS Message Format

Since every website lookup must start with a DNS query, DNS is designed to be very lightweight and fast - it uses UDP (best-effort packets, no TCP handshakes) and has a fairly simple message format.

16 bits	16 bits
Identification	Flags
# Questions	# Answer RRs
# Authority RRs	# Additional RRs
Questions (variable # of RRs)	
Answers (variable # of RRs)	
Authority (variable # of RRs)	
Additional info (variable # of RRs)	

Figure 16: The DNS packet header.

The first field is a 16 bit **identification field** that is randomly selected per query and used to match requests to responses. When a DNS query is sent, the ID field is filled with random

bits. Since UDP is stateless, the DNS response must send back the same bits in the ID field so that the original query sender knows which DNS query the response corresponds to.

Sanity check: Which type(s) of adversary can read this ID field? Which type(s) of adversary cannot read the ID field and must guess it when attacking DNS?[13]

The next 16 bits are reserved for flags, which specify whether the message is a query or a response, as well as whether the query was successful (e.g. the `NOERROR` flag is set in the reply if the query succeeded, the `NXDOMAIN` flag is set in the reply if the query asked about a non-existent name).

The next field specifies the number of questions asked (in practice, this is always 1). The three fields after that are used in response messages and specify the number of **resource records** (RRs) contained in the message. We'll describe each of these categories of RRs in depth later.

The rest of the message contains the actual content of the DNS query/response. This content is always structured as a set of RRs, where each RR is a key-value pair with an associated type.

For completeness, a DNS record key is formally defined as a 3-tuple `<Name, Class, Type>`, where `Name` is the actual key data, `Class` is always `IN` for Internet (except for special queries used to get information about DNS itself), and `Type` specifies the record type. A DNS record value contains `<TTL, Value>`, where `TTL` is the time-to-live (how long, in seconds, the record can be cached), and `Value` is the actual value data.

There are two main types of records in DNS. **A type records** map domains to IP addresses. The key is a domain, and the value is an IP address. **NS type records** map zones to domains. The key is a zone, and the value is a domain.

Important takeaways from this section: Each DNS packet has a 16-bit random ID field, some metadata, and a set of resource records. Each record falls into one of four categories (question, answer, authority, additional), and each record contains a type, a key, and a value. There are A type records and NS type records.

8.4 DNS Lookup

Now, let's walk through a real DNS query for the IP address of `eecs.berkeley.edu`. You can try this at home with the `dig` utility–remember to set the `+norecurse` flag so you can unravel the recursion yourself.

Every DNS query begins with the root server. For redundancy, there are actually 13 root servers located around the world. We can look up the IP addresses of the root servers, which are public and well-known. In a real recursive resolver, these addresses are usually hardcoded.

The first root server has domain `a.root-servers.net` and IP address `198.41.0.4`. We can use `dig` to send a DNS request to this address, asking for the IP address of `eecs.berkeley.edu`.

[13] A: MITM and on-path can read the ID field. Off-path must guess the ID field.

```
$ dig +norecurse eecs.berkeley.edu @198.41.0.4

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26114
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 13, ADDITIONAL: 27

;; QUESTION SECTION:
;eecs.berkeley.edu.          IN   A

;; AUTHORITY SECTION:
edu.                172800   IN   NS   a.edu-servers.net.
edu.                172800   IN   NS   b.edu-servers.net.
edu.                172800   IN   NS   c.edu-servers.net.
...

;; ADDITIONAL SECTION:
a.edu-servers.net.  172800   IN   A    192.5.6.30
b.edu-servers.net.  172800   IN   A    192.33.14.30
c.edu-servers.net.  172800   IN   A    192.26.92.30
...
```

In the first section of the answer, we can see the header information, including the ID field (`26114`), the return flags (`NOERROR`), and the number of records returned in each section.

The **question section** contains 1 record (you can verify by seeing `QUERY: 1` in the header). It has key `eecs.berkeley.edu`, type `A`, and a blank value. This represents the domain we queried for (the value is blank because we don't know the corresponding IP address).

The **answer section** is blank (`ANSWER: 0` in the header), because the root server didn't provide a direct answer to our query.

The **authority section** contains 13 records. The first one has key `.edu`, type `NS`, and value `a.edu-servers.net`. This is the root server giving us the zone and the domain name of the next name server we should contact. Each record in this section corresponds to a potential name server we could ask next.

The **additional section** contains 27 records. The first one has key `a.edu-servers.net`, type `A`, and value `192.5.6.30`. This is the root server giving us the IP address of the next name server by mapping a domain from the authority section to an IP address.

Together, the authority section and additional section combined give us the zone, domain name, and IP address of the next name server. This information is spread across two sections to maintain the key-value structure of the DNS message.

For completeness: `172800` is the TTL (time-to-live) for each record, set at 172,800 seconds = 48 hours here. The `IN` is the Internet class and can basically be ignored. Sometimes you

will see records of type `AAAA`, which correspond to IPv6 addresses (the usual `A` type records correspond to IPv4 addresses).

Sanity check: What name server do we query next? How do we know where that name server is located? What do we query that name server for?[14]

```
$ dig +norecurse eecs.berkeley.edu @192.5.6.30

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36257
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 3, ADDITIONAL: 5

;; QUESTION SECTION:
;eecs.berkeley.edu.           IN   A

;; AUTHORITY SECTION:
berkeley.edu.        172800   IN   NS   adns1.berkeley.edu.
berkeley.edu.        172800   IN   NS   adns2.berkeley.edu.
berkeley.edu.        172800   IN   NS   adns3.berkeley.edu.

;; ADDITIONAL SECTION:
adns1.berkeley.edu.  172800   IN   A    128.32.136.3
adns2.berkeley.edu.  172800   IN   A    128.32.136.14
adns3.berkeley.edu.  172800   IN   A    192.107.102.142
...
```

The next query also has an empty answer section, with `NS` records in the authority section and `A` records in the additional section which give us the domains and IP addresses of name servers responsible for the `berkeley.edu` zone.

```
$ dig +norecurse eecs.berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52788
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;eecs.berkeley.edu.          IN   A

;; ANSWER SECTION:
eecs.berkeley.edu.  86400   IN   A   23.185.0.1
```

Finally, the last query gives us the IP address corresponding to `eecs.berkeley.edu` in the

[14] Query `a.edu-servers.net`, whose location we know because of the records in the additional section. Query for the IP address of `eecs.berkeley.edu` just like before.

form of a single `A` type record in the answer section.

In practice, because the recursive resolver caches as many answers as possible, most queries can skip the first few steps and used cached records instead of asking root servers and high-level name servers like `.edu` every time. Caching helps speed up DNS, because fewer packets need to be sent across the network to translate a domain name to an IP address. Caching also helps reduce request load on the highest-level name servers.

9 DNS Security

9.1 Bailiwick

DNS is insecure against a malicious name server. For example, if a `berkeley.edu` name server was taken over by an attacker, it could send answer records that point to malicious IP addresses.

However, a more dangerous exploit is using the additional section to poison the cache with even more malicious IP addresses. For example, this malicious DNS response would cause the resolver to associate `google.com` with an attacker-owned IP address `6.6.6.6`.

```
$ dig +norecurse eecs.berkeley.edu @192.5.6.30

...
;; ADDITIONAL SECTION:
adns1.berkeley.edu.  172800   IN   A    128.32.136.3
www.google.com       999999   IN   A    6.6.6.6
...
```

To prevent any malicious name server from doing too much damage, resolvers implement **bailiwick checking**. With bailiwick checking, a name server is only allowed to provide records in its zone. This means that the `berkeley.edu` name server can only provide records for domains under `berkeley.edu` (not `stanford.edu`), the `.edu` name server can only provide records for domains under `.edu` (not `google.com`), and the root name servers can provide records for anything.

9.2 On-path attackers and off-path attackers

Against an on-path attacker, DNS is completely insecure - everything is sent over plaintext, so an attacker can read the request, construct a malicious response message with malicious records and the correct ID field, and race to send the malicious reply before the legitimate response. If the time-to-live (TTL) of the malicious records is set to a very high number, then the victim will cache those malicous records for a very long time.

For both on-path and off-path attackers, if the legitimate response arrives before the fake response, it is cached. Caching limits the attacker to only a few tries per week, because future requests for that domain can reference the cache, so no DNS queries are sent. Since off-path attackers must guess the ID field with a $1/2^{16}$ probability of success, and they only get a few tries per week, DNS was believed to be secure against off-path attackers, until Dan Kaminsky discovered a flaw in the DNS protocol in 2008. This attack was so severe that Kaminsky was awarded with a Wikipedia article.

9.3 Kaminsky attack

The Kaminsky attack relies on querying for nonexistent domains. Remember that the legitimate response for a nonexistent domain is an `NXDOMAIN` status with no other records, which means that nothing is cached! This allows the attacker to repeatedly race until they win, without having to wait for cached records to expire.

An attacker can now include malicious additional records in the fake response for the nonexistent `fake161.berkeley.edu`:

```
$ dig fake161.berkeley.edu

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 29439
;; flags: qr aa; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;fake161.berkeley.edu.         IN   A

;; ADDITIONAL SECTION:
berkeley.edu.      999999     IN   A    6.6.6.6
```

If the fake response arrives first, the resolver will cache the malicious additional record. Notice that this doesn't violate bailiwick checking, since the name server responsible for answering `fake161.berkeley.edu` can provide a record for `berkeley.edu`.

Now that the attacker can try as many times as they want, all that's left is to force a victim to make thousands of DNS queries for nonexistent domains. This can be achieved by tricking the victim into visiting a website that tries to load lots of nonexistent domains:

```
<img src="http://fake001.berkeley.edu/image.jpg"/>
<img src="http://fake002.berkeley.edu/image.jpg"/>
<img src="http://fake003.berkeley.edu/image.jpg"/>
...
```

This HTML snippet will cause the victim's browser to try and fetch images from `http://fake001.berkeley.edu/image.jpg`, `http://fake002.berkeley.edu/image.jpg`, etc. To fetch these images, the browser will first make a DNS request for the domains `fake001.berkeley.edu`, `fake002.berkeley.edu`, etc. For each request, if the legitimate response arrives before the malicious response, or if the off-path attacker incorrectly guesses the ID field, nothing is cached, so the attacker can immediately try again when the victim makes the next DNS request to the next non-existent domain.

The Kaminsky attack allows on-path attackers to race until their fake response arrives first and off-path attackers to race until they successfully guess the ID field. There is no way to completely eliminate the Kaminsky attack in regular DNS, although modern DNS protocols add **UDP source port randomization** to make it much harder.

Recall that UDP is a transport-layer protocol like TCP, so a UDP packet requires a source port and destination port. The destination port must be well-known and constant (in practice, it is always 53), so everyone can send UDP packets to the correct port on the name server. However, DNS doesn't specify what source port the resolver uses to send queries, so source port randomization uses a random 16-bit source port for each query. The name server must send the response packet back to the correct source port of the resolver, so it must include the source port number in the destination port field of the response. Now, an attacker must guess the 16-bit ID field and the 16-bit source port in order to successfully forge a response packet. This decreases an off-path attacker's probability of success to $1/2^{32}$, which is much harder, but certainly not impossible.

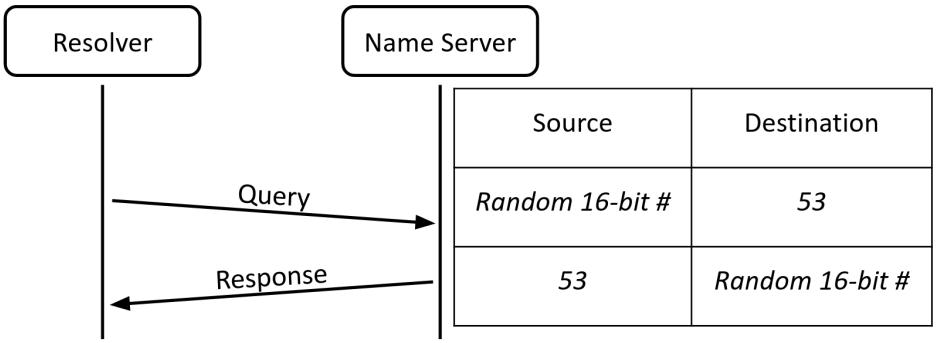

Figure 17: Source port randomization.

Sanity check: How much extra security does source port randomization provide against on-path attackers?[15]

[15] A: None, on-path attackers can see the source port value.

10 DNSSEC

DNSSEC is an extension to regular DNS that provides integrity and authentication on all DNS messages sent. Sanity check: Why do we not care about the confidentiality of DNSSEC?[16]

10.1 Signing records

We want every DNS record to have integrity and authenticity, and we want everyone to be able to verify the integrity and authenticity of records. Digital signatures are a good fit in this situation, because only someone with the private key can create signatures, and everyone can use the public key to verify signatures.

To ensure integrity and authenticity, let's have every name server generate a public/private key pair and sign every record it sends with its private key. When the name server receives a DNS request, it sends the records, along with a signature on the records and the public key, to the resolver. The resolver uses the public key to verify the signature on the records.

Because of the signatures, a network attacker (MITM, on-path, off-path) cannot tamper with the data or inject malicious data without being detected (integrity). Also, the resolver can cache the signatures and the public key, and check at any time that the records actually came from the name server (authenticity).

You might see a flaw in this design: what if a name server is malicious? Then the malicious name server could return valid signatures on malicious records. How do we modify our design to prevent this?

10.2 Delegating trust

The main issue in our design so far is we lack a *trust anchor*. We want DNSSEC to defend against malicious name servers, so we cannot implicitly trust the name servers. However, if we don't trust anybody, then DNSSEC will never work (we'll never trust any records we get), so we must first choose a trust anchor, an entity that we implicitly trust. In DNSSEC, the root servers are the trust anchor: every computer automatically assumes that the root server is honest and uncompromised. In real life, this is a safe assumption, because the organizations overseeing the Internet hold painstakingly formal ceremonies to ensure that the root server is uncompromised. (If you're interested, you can read more about the root signing ceremony here.)

Given a trust anchor, we can now *delegate trust* from the trust anchor to somebody else. If the root endorses Alice, then you can be sure that Alice is trusted as well, since you implicitly trust the root. Also, if Alice endorses Bob, then you can be sure that Bob is trusted, since you trust Alice. This trust delegation starting from the root is how DNSSEC delegates trust from the root to all legitimate name servers, while protecting against malicious name servers.

[16] A: DNS responses don't contain sensitive data. Anyone could query the name servers for the same information.

Consider two parties, root and Alice, who each have a public key and a private key. You trust root, because it is the trust anchor. The root can delegate trust to Alice by *signing Alice's public key*. The root's signature on Alice's public key effectively says that Alice's public key is trustworthy, and the root trusts any message signed by Alice using her corresponding private key.

Now, when Alice signs a message, we can use Alice's public key to verify that the message was properly signed by Alice. Also, we know that Alice's public key is trusted, because the root has signed it, and we implicitly trust the root.

If Alice was malicious, then the root would not delegate trust to her by signing her public key, because we are trusting that the root is honest and uncompromised.

We can apply this delegation idea to the entire DNS tree. Each name server will sign the public key of all its trusted children name servers. For example, root signs `.edu`'s public key. We trust root, and root signed `.edu`'s public key, so now we trust `.edu`. Next, `.edu` signs `berkeley.edu`'s public key. We trust `.edu`, and `.edu` signed `berkeley.edu`'s public key, so now we trust `berkeley.edu`.

10.3 DNSSEC Intuition

With these ideas in mind, let's revisit the DNS query for `eecs.berkeley.edu` from earlier and convert it to a secure DNSSEC query. *The DNSSEC additions are italicized.*

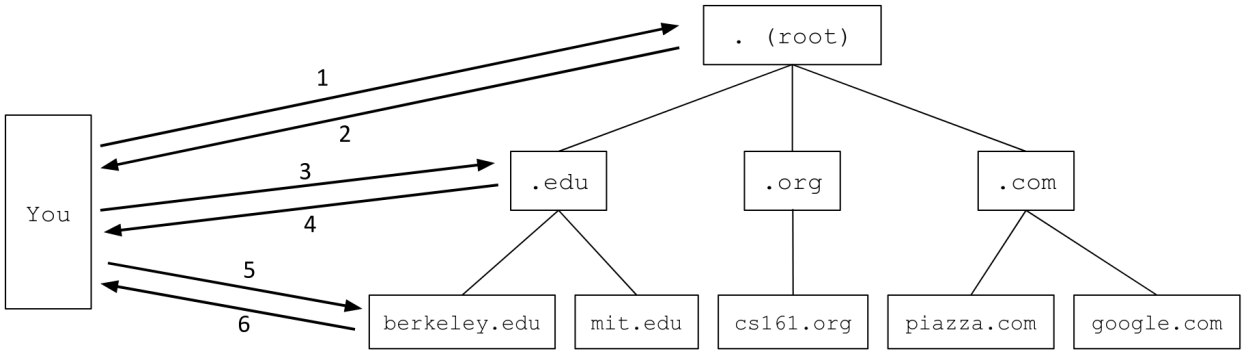

Figure 18: Steps of a DNS query.

1. You to the root name server: Please tell me the IP address of `eecs.berkeley.edu`.
2. Root server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `.edu` zone. It has human-readable domain name `a.edu-servers.net` and IP address `192.5.6.30`. *Here is a signature on the next name server's public key. If you trust me, then now you trust them too. Finally, here is my public key.*
3. You to the `.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
4. The `.edu` name server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `berkeley.edu`

zone. It has human-readable domain name `adns1.berkeley.edu` and IP address `128.32.136.3`. *Here is a signature on the next name server's public key. If you trust me, then now you trust them too. Finally, here is my public key.*

5. You to the `berkeley.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
6. The `berkeley.edu` name server to you: OK, the IP address of `eecs.berkeley.edu` is `23.185.0.1`. *Finally, here is my public key and a signature on the answer.*

Note that we implicitly trust all signed messages from the root, because the root is our trust anchor. In practice, all DNS resolvers have the root's public key hardcoded, and any messages verified with that hardcoded key are implicitly trusted.

Congratulations, you now have all the intuition for how DNSSEC works! The rest of this section shows how we implement this design in DNS.

10.4 New DNSSEC record types

To store cryptographic information in DNS messages, we need to introduce a few new record types.

The `DNSKEY` **type record** encodes a public key.

The `RRSIG` **type record** is a signature on a set of multiple other records in the message, all of the same type. For example, if the authority section returns 13 `NS` type records, you can sign all 13 records at once with one `RRSIG` type record. However, to sign the 26 `A` type records in the additional section, you would need another `RRSIG` type record. In addition to the actual cryptographic signature, the `RRSIG` type record contains the type of the records being signed, the signature creation and expiration date, and the identity of the signer (information about which public key/`DNSKEY` record should be used to verify this signature).

The `DS` **(Delegated Signer) type record** is a hash of the signer's name and a child's public key. The `DS` record, combined with a `RRSIG` record that signs the `DS` record, effectively allows each name server to sign the public key of its trusted children.

All DNSSEC cryptographic records additionally include some (uninteresting) metadata, such as which algorithm was used for signing/verifying/hashing.

You might have noticed that the number of additional records is always 1 more than the actual number of additional records that appear in the response. For example, consider the final query in our regular DNS query walkthrough:

```
$ dig +norecurse eecs.berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52788
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;eecs.berkeley.edu.          IN    A
```

```
;; ANSWER SECTION:
eecs.berkeley.edu.  86400   IN   A   23.185.0.1
```

The response reports 1 additional record but shows no additional records at all. This extra record corresponds to the `OPT` pseudosection (seen just above the question section). This pseudosection allows extra space for DNSSEC-specific flags (e.g. the `DO` flag requests DNSSEC information), but in order to be backwards-compatible with regular DNS, the section is encoded as an additional record when sent in the request and the reply.

10.5 Key Signing Keys and Zone Signing Keys

There is one final complication in DNSSEC–what if a name server wants to change its key pair? A key change is necessary if, for example, an attacker steals the private key of a trusted name server, because now the attacker can impersonate a trusted name server.

In our current DNSSEC design, a name server that wants to change keys must notify its parent name server so that the parent can change the DS record (which endorses the child's public key). As it turns out, this process is difficult to perform securely and can easily go wrong.

To minimize the use of this difficult key change protocol, each DNSSEC name server generates two public/private key pairs. The **key signing key (KSK)** is only used to sign the zone signing key, and the **zone signing key (ZSK)** is used to sign everything else.

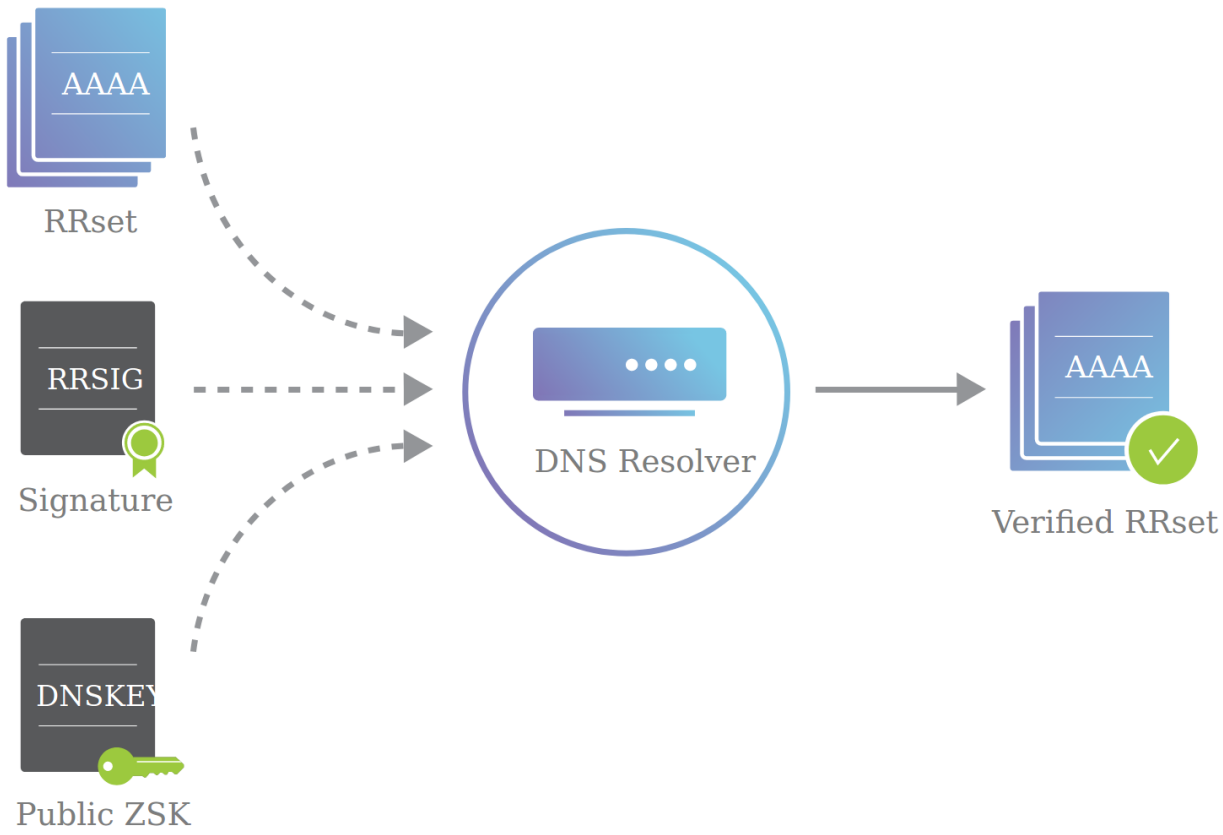

Figure 19: Validating a set of `AAAA` type records with an `RRSIG` on them and the public ZSK used to generate the signature. Source: How DNSSEC Works (Cloudflare)

In our previous design with one key pair, the name server sends (1) a set of records, (2) a signature on those records, and (3) the public key (endorsed by the parent). The DNS resolver uses the public key to verify the signature, and accepts the set of records.

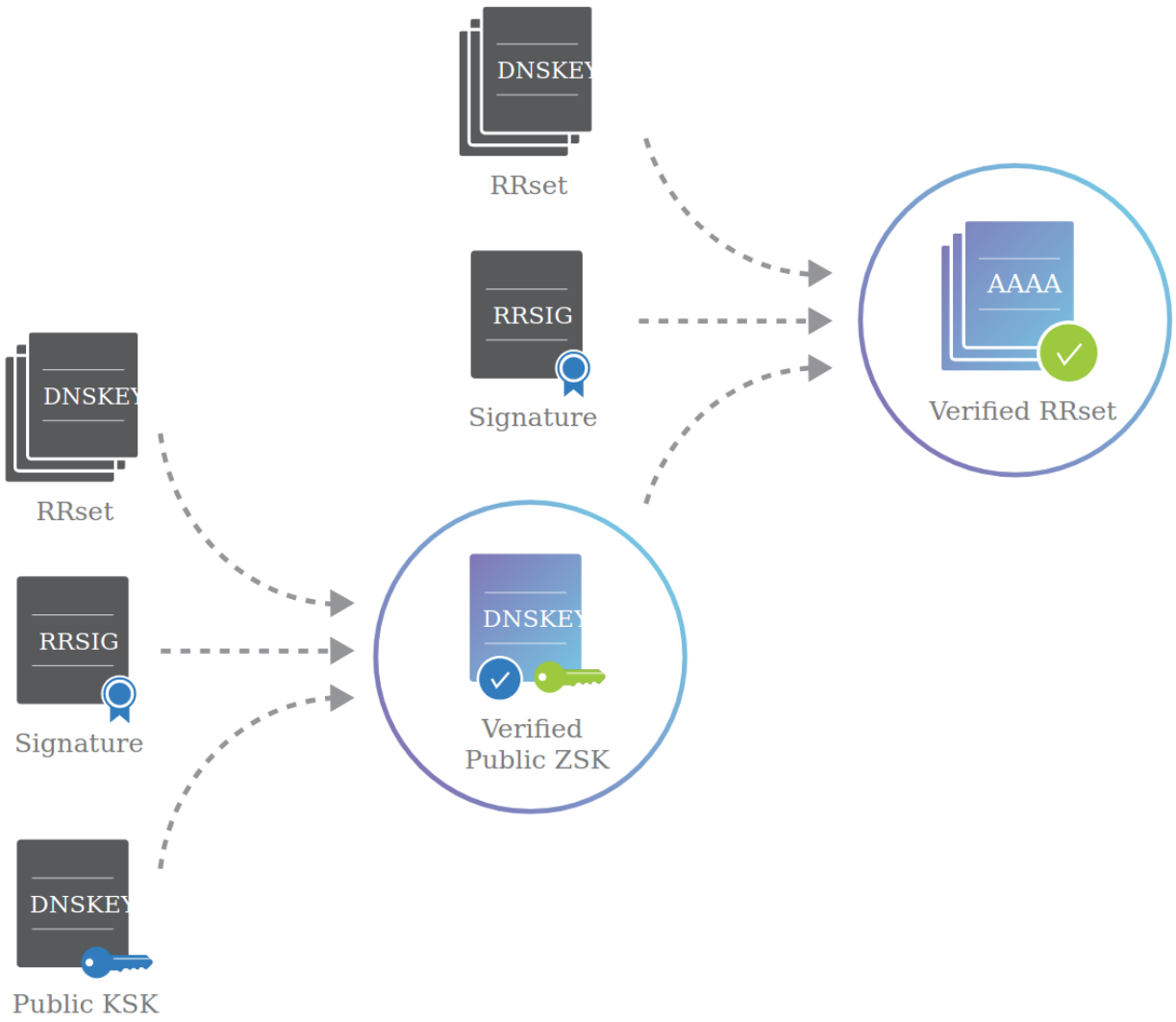

Figure 20: Validating a ZSK with an `RRSIG` on it and the public KSK used to generate the signature. Then validating the set of `AAAA` type records with an `RRSIG` on them and the public ZSK used to generate the signature. Source: How DNSSEC Works (Cloudflare)

In our new design with two key pairs, the name server sends (1) the public ZSK, (2) a signature on the public ZSK, and (3) the public KSK (endorsed by the parent). The DNS resolver uses the public KSK to verify the signature, and accepts the public ZSK. Note that this is the exact same structure that was used to sign records before, but in this case, the record is the public ZSK, signed using the KSK.

Another way to think about this step is to recall that a parent endorses a child by signing its public key. You can think of the KSK as the "parent" and the ZSK as the "child," both within one name server. The parent (KSK) endorses the child (ZSK) by signing the public ZSK.

The result of this first step is that we now have a trusted public ZSK. The second step is the same as before: the name server sends a set of records, a signature on those records (using the private ZSK), and the public ZSK (endorsed by the KSK in the previous step).

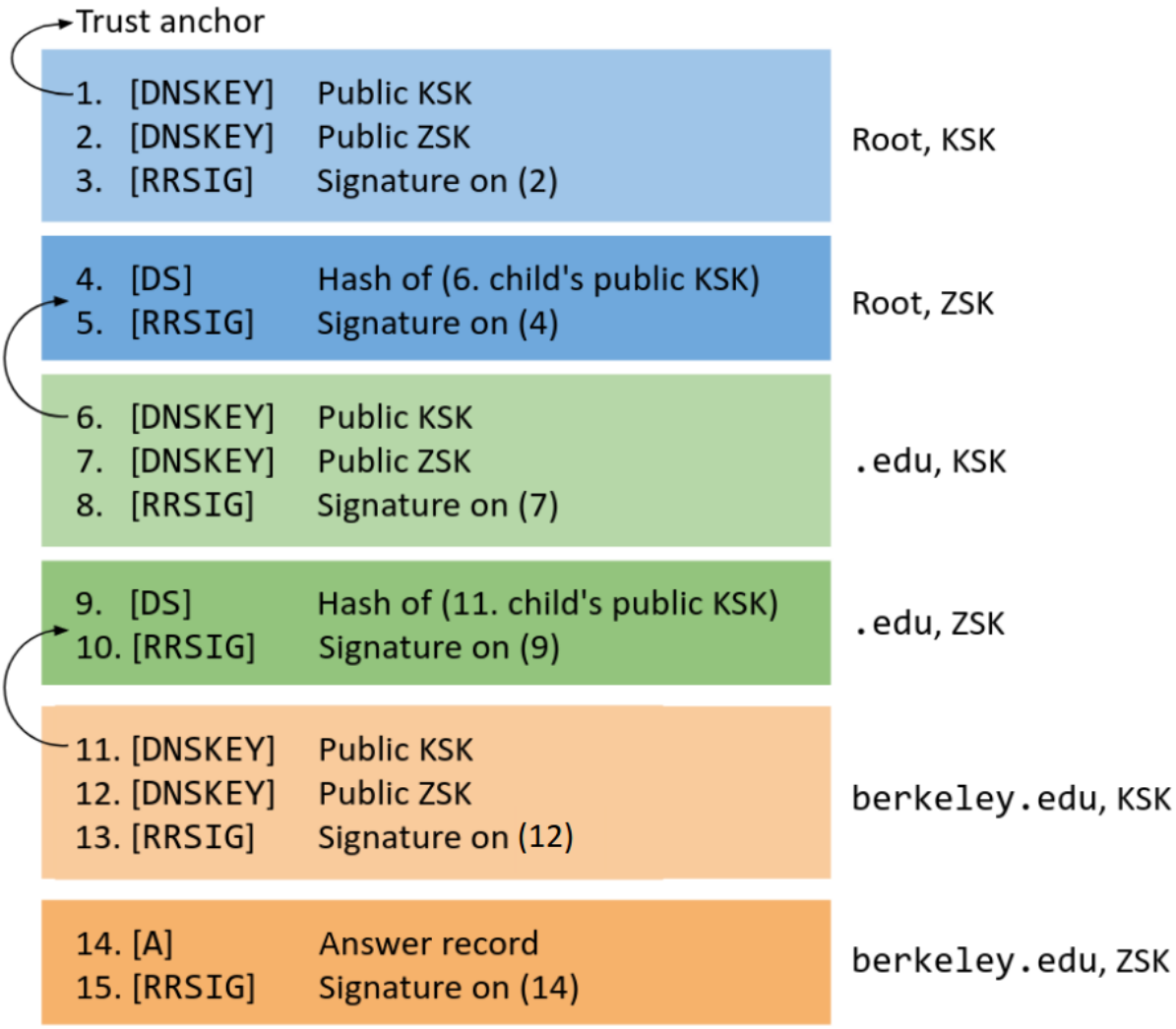

Figure 21: The DNSSEC chain of trust.

Here is a diagram of the entire two-key DNSSEC. Each color (blue, green, orange) represents a name server. The lighter shade represents records signed with the KSK. The darker shade represents records signed with the ZSK.

Verification would proceed as follows.

- Light blue: Because of our trust anchor, we trust the KSK of the root (1). The root's KSK signs its ZSK, so now we trust the root's ZSK (2-3).
- Dark blue: We trust the root's ZSK. The root's ZSK signs `.edu`'s KSK (4-5), so now we trust `.edu`'s KSK.
- Light green: We trust the `.edu`'s KSK (6). `.edu`'s KSK signs `.edu`'s ZSK, so now we trust `.edu`'s ZSK (7-8).
- Dark green: We trust `.edu`'s ZSK. `.edu`'s ZSK signs `berkeley.edu`'s KSK (9-10), so now we trust `berkeley.edu`'s KSK.

- Light orange: We trust the `berkeley.edu`'s KSK (11). `berkeley.edu`'s KSK signs `berkeley.edu`'s ZSK, so now we trust `berkeley.edu`'s ZSK (12-13).
- Dark orange: We trust `berkeley.edu`'s ZSK. `berkeley.edu`'s ZSK signs the final answer record (14-15), so now we trust the final answer.

10.6 DNSSEC query walkthrough

Now we're ready to see a full DNSSEC query in action. As before, you can try this at home with the `dig` utility–remember to set the `+norecurse` flag so you can unravel the recursion yourself, and remember to set the `+dnssec` flag to enable DNSSEC.

First, we query the root server for its public keys. Recall that the root's IP address, `198.41.0.4`, is publicly-known and hardcoded.

```
$ dig +norecurse +dnssec DNSKEY . @198.41.0.4

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 7149
;; flags: qr aa; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1472
;; QUESTION SECTION:
;.              IN    DNSKEY

;; ANSWER SECTION:
.     172800    IN    DNSKEY    256 {ZSK of root}
.     172800    IN    DNSKEY    257 {KSK of root}
.     172800    IN    RRSIG     DNSKEY {signature on DNSKEY records}
...
```

In this response, the root has returned its public ZSK, public KSK, and a `RRSIG` type record over the two `DNSKEY` type records. We can use the public KSK to verify the signature on the public ZSK.

Because we implicitly trust the root's KSK (trust anchor), and the root's KSK signs its ZSK, we now trust the root's ZSK.

Next, we query the root server for the IP address of `eecs.berkeley.edu`.

```
$ dig +norecurse +dnssec eecs.berkeley.edu @198.41.0.4

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 5232
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 15, ADDITIONAL: 27
```

```
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;eecs.berkeley.edu.                      IN   A

;; AUTHORITY SECTION:
edu.                172800   IN   NS      a.edu-servers.net.
edu.                172800   IN   NS      b.edu-servers.net.
edu.                172800   IN   NS      c.edu-servers.net.
...
edu.                86400    IN   DS      {hash of .edu's KSK}
edu.                86400    IN   RRSIG   DS {signature on DS record}

;; ADDITIONAL SECTION:
a.edu-servers.net.  172800   IN   A       192.5.6.30
b.edu-servers.net.  172800   IN   A       192.33.14.30
c.edu-servers.net.  172800   IN   A       192.26.92.30
...
```

DNSSEC doesn't remove any records compared to regular DNS–the question, answer (blank here), authority, and additional sections all contain the same records from regular DNS. However, DNSSEC adds a `DS` record and a `RRSIG` signature record on the `DS` record. Together, these two records sign the KSK of the `.edu` name server with the root's ZSK. Since we trust the root's ZSK (from the previous step), now we trust the `.edu` name server's KSK.

Next, we query the `.edu` name server for its public keys.

```
$ dig +norecurse +dnssec DNSKEY edu. @192.5.6.30

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9776
;; flags: qr aa; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;edu.          IN   DNSKEY

;; ANSWER SECTION:
edu.   86400   IN   DNSKEY  256 {ZSK of .edu}
edu.   86400   IN   DNSKEY  257 {KSK of .edu}
edu.   86400   IN   RRSIG   DNSKEY {signature on DNSKEY records}
...
```

In this response, the `.edu` name server has returned its public ZSK, public KSK, and a

RRSIG type record over the two DNSKEY type records. We can use the public KSK to verify the signature on the public ZSK.

Because we trust the .edu name server's KSK (from the previous step), and the .edu KSK signs its ZSK, we now trust the .edu name server's ZSK.

Next, we query the .edu name server for the IP address of eecs.berkeley.edu.

```
$ dig +norecurse +dnssec eecs.berkeley.edu @192.5.6.30

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60799
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 5, ADDITIONAL: 5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;eecs.berkeley.edu.                     IN  A

;; AUTHORITY SECTION:
berkeley.edu.        172800  IN  NS     adns1.berkeley.edu.
berkeley.edu.        172800  IN  NS     adns2.berkeley.edu.
berkeley.edu.        172800  IN  NS     adns3.berkeley.edu.
berkeley.edu.        86400   IN  DS     {hash of berkeley.edu's KSK}
berkeley.edu.        86400   IN  RRSIG  DS {signature on DS record}

;; ADDITIONAL SECTION:
adns1.berkeley.edu.  172800  IN  A      128.32.136.3
adns2.berkeley.edu.  172800  IN  A      128.32.136.14
adns3.berkeley.edu.  172800  IN  A      192.107.102.142
...
```

In this response, the .edu name server returns NS and A type records that tell us what name server to query next, just like in regular DNS.

In addition, the response has a DS type record and an RRSIG signature on the DS record. Sanity check: which key is used to sign the DS record?[17] Together, these two records sign the KSK of the berkeley.edu name server. Because we trust the .edu name server's ZSK (from the previous step), and the .edu ZSK signs the berkeley.edu KSK, we now trust the berkeley.edu name server's KSK.

Next, we query the berkeley.edu name server for its public keys.

```
$ dig +norecurse +dnssec DNSKEY berkeley.edu @128.32.136.3

;; Got answer:
```

[17] A: The ZSK of the .edu name server.

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4169
;; flags: qr aa; QUERY: 1, ANSWER: 5, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1220
;; QUESTION SECTION:
;berkeley.edu.          IN  DNSKEY

;; ANSWER SECTION:
berkeley.edu.  172800  IN  DNSKEY  256 {ZSK of berkeley.edu}
berkeley.edu.  172800  IN  DNSKEY  257 {KSK of berkeley.edu}
berkeley.edu.  172800  IN  RRSIG   DNSKEY {signature on DNSKEY records}
...
```

In this response, the `berkeley.edu` name server has returned its public ZSK, public KSK, and a `RRSIG` type record over the two `DNSKEY` type records. We can use the public KSK to verify the signature on the public ZSK.

Because we trust the `berkeley.edu` name server's KSK (from the previous step), and the `berkeley.edu` KSK signs its ZSK, we now trust the `berkeley.edu` name server's ZSK.

Finally, we query the `berkeley.edu` name server for the IP address of `eecs.berkeley.edu`.

```
$ dig +norecurse +dnssec eecs.berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 21205
;; flags: qr aa; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1220
;; QUESTION SECTION:
;eecs.berkeley.edu.          IN  A

;; ANSWER SECTION:
eecs.berkeley.edu.  86400   IN  A      23.185.0.1
eecs.berkeley.edu.  86400   IN  RRSIG  A {signature on A record}
```

This response has the final answer `A` type record and a signature on the final answer. Because we trust the `berkeley.edu` name server's ZSK (from the previous part), we also trust the final answer.

10.7 Nonexistent domains

Remember that DNS is designed to be fast and lightweight. However, public-key cryptography is slow, because it requires math. As a result, name servers that support DNSSEC

sign records *offline*–records are signed ahead of time, and the signatures saved in the server along with the records. When the server receives a DNS query, it can immediately return the saved signature without computing it.

Offline signing works fine for existing domains, but what if we receive a request for a nonxistent domain? There are infinitely many nonexistent domains, so we cannot sign them all offline. However, we cannot sign requests for nonexistent domains *online* either, because this is too slow. Also, online cryptography makes name servers vulnerable to an attack. Sanity check: what's the attack?[18]

DNSSEC has a clever solution to this problem–instead of signing individual nonexistent domains, name servers pre-compute signatures on *ranges* of nonexistent domains. Suppose we have a website with three subdomains:

```
b.example.com
l.example.com
q.example.com
```

If we sort every possible subdomain alphabetically, there are three ranges of nonexistent domains: everything between `b` and `l`, `l` and `q`, and `q` and `b` (wrapping around from z to a).

Now, if someone queries for `c.example.com`, instead of signing a message proving the nonexistence of that specific domain, the name server returns a **NSEC record** saying, "No domains exist between `b.example.com` and `l.example.com`. Signed, name server."

NSEC records have a slight vulnerability - notice that every time we query for a nonexistent domain, we can discover two valid domains that we might have otherwise not known. By traversing the alphabet, an attacker can now learn the names of every subdomain of the website:

1. Query `c.example.com`. Receive NSEC saying nothing exists between `b` and `l`. Attacker now knows `b` and `l` exist.
2. Query `m.example.com`. Receive NSEC saying nothing exists between `l` and `q`. Attacker now knows `q` exists.
3. Query `r.example.com`. Receive NSEC saying nothing exists between `q` and `b`. Attacker has already seen `b`, so they know they have walked the entire alphabet successfully.

Some argue that this is not really a vulnerability, because hiding a domain name like `admin.example.com` is relying on security through obscurity. Nevertheless, an attempt to fix this was implemented as **NSEC3**, which simply uses the hashes of every domain name instead of the actual domain name.

```
372fbe338b9f3bb6f857352bc4c6a49721d6066f (l.example.com)
6898bc7daf3054daae05e8763153ee1506e809d5 (q.example.com)
f96a6ec2fb6efbe43002f4cbf124f90879424d79 (b.example.com)
```

[18] A: Denial of service (DoS). Flood the name server with requests for nonexistent domains, and it will be forced to sign all of them.

The order of the domain names has changed, but the process is the same - if someone queries for `c.example.com`, which hashes to `8dca64e4b6e1724f0d84c5c25c9354d5529ab0a2`, the NSEC3 record will say, "No domains exist that hash to values between `6898b...` and `f96a6...`. Signed, name server."

Of course, an attacker could buy a GPU and precompute hashes to learn domain names anyway...and NSEC5 was born. Fortunately, it's still out of scope for this class.

11 Intrusion Detection

In this class, we've talked about many ways to prevent attacks, but not all defenses are perfect, and attacks will often slip through our defenses. How do we detect these attacks when they happen?

Imagine that you're managing a local network of computers (for example, all the web servers and employee computers in a company's office building). The local network is connected to the Internet with a router (recall that all requests from the local network to the wider Internet will pass through this router). How can we detect attacks on this network?

11.1 Types of detectors

There are three broad types of detectors. The main difference in implementation is where on the network these detectors are installed. Each type of detector has its advantages and drawbacks.

11.1.1 Network Intrusion Detection System (NIDS)

A NIDS (network intrusion detection system) is installed between the router and the internal network. This means that all requests to and from the outside Internet must pass through the NIDS. The NIDS can see (and potentially modify) all packets sent to the outside Internet and received from the outside Internet.

The biggest advantage of a NIDS is that a single NIDS is enough to cover the entire network. There's no need to install anything on the end hosts (e.g. employee computers or web servers) because all their requests will pass through the NIDS anyway. Installing a single NIDS for the whole network is a cheap solution with low management overhead.

However, there are some drawbacks to using a NIDS. Recall that even though the Internet fundamentally works by sending packets, rich information communicated with higher-layer protocols are made up of multiple packets. For example, a single message sent through TCP may consist of many small packets that are combined to form a longer message. Also, packets may be dropped or sent out of order–it's the end hosts' responsibility to rearrange the pieces correctly with TCP.

A plain NIDS that just observes individual packets would not be too useful, because it will probably see a lot of packets with partial data out of order. The NIDS may also be seeing packets from lots of different TCP connections, since every connection from inside the network goes through the NIDS. A more useful NIDS would separate packets by their connection and correctly reorder the packets within each connection together by TCP sequence number. Once the NIDS has successfully reconstructed the connection, it can read the rich information and analyze it for attacks.

To make matters worse, the TCP connection reconstructed at the NIDS may not match the TCP connection that the end host sees. Recall that each TCP packet has a time-to-live (TTL) field, which specifies how long the packet can be in transit before it expires. (This

is often measured in the number of hops, i.e. the number of machines that the packet has been sent through.) Then there could be a scenario where the NIDS receives a TCP packet because the TTL has not yet expired, but by the time it's sent to the end host, the TTL has expired, and the end host discards the message. There could also be a scenario where the NIDS sees a packet, but it gets corrupted or dropped before it reaches the recipient. Thus the NIDS must also reason about packets that potentially don't reach the end host.

The possibility of inconsistent interpretations of messages between the NIDS and the end host can be exploited for attacks. Consider a NIDS that raises an alert for an attack if it encounters the string `/etc/passwd` in any request. An attacker could send a packet with the content `%65%74%63/%70%61%73%73%77%64`. To a basic NIDS doing string matching, this won't look like an attack, but if the end host is expecting a URL-encoded string and decodes this string, then the end host will receive the string `/etc/passwd`. The NIDS has failed to detect a potential password attack! This type of attack, where the attacker tries to obfuscate the contents of an attack, is called an *evasion attack*. The possibility of evasion attacks suggests that not only does the the NIDS have to reason about inconsistent information about connections, but the NIDS must also reason about how the end hosts may potentially interpret the information in the connection.

Another major issue with NIDS is the need to deal with encrypted traffic. Most modern web traffic is encrypted with HTTPS (TLS), which is end-to-end secure. In other words, the NIDS has no way to determine the contents of the messages being sent. To allow NIDS to analyze encrypted traffic, the network may need to be configured so that the end hosts give the NIDS their private keys to allow the NIDS to decrypt TLS connections. This might not always be a desirable solution, since it compromises the security of private keys and the security guarantees of NIDS, and it may allow network analysts to see sensitive information that only the end hosts should see.

11.1.2 Host-based Intrusion Detection System (HIDS)

A HIDS (host-based intrusion detection system) is installed directly on the end hosts. For example, antivirus software might be considered a HIDS, because it is installed on the same computer that is generating and receiving network requests.

HIDS have much fewer inconsistency issues than NIDS. Since the HIDS is located on the same machine that is receiving and interpreting the requests, it can directly check what data is received and how the data is being parsed. HTTPS connections are also no longer an issue, because the NIDS can view the decrypted traffic at the end host.

However, these advantages don't come for free. Unlike NIDS, where a single implementation can defend against the entire network, a HIDS must be installed for every machine on the network. This can be very costly, especially if different machines need differently-configured HIDS.

HIDS also don't defend against all evasion attacks. For example, a web server might expect a filename input from the user and serve the matching file to the user. If the user inputs `evanbot.txt`, the server might check the `/public/files` directory and return the

`/public/files/evanbot.txt` file to the user. An attacker could supply a malicious input like `../../etc/passwd`. In Unix, `..` says to go up one directory, so this input would allow the attacker to access the passwords file, even though it's located in a different directory on the server. This type of attack is called a *path traversal attack*. To fully defend against path traversal attacks, it is not enough for the the HIDS to understand the contents of the end request. The HIDS would also need to reason about how the underlying filesystem interprets the contents of the end request. This can lead to further parsing inconsistencies and evasion attacks.

11.1.3 Logging

A third approach to intrusion detection is logging. Most modern web servers generate logs with information such as what web requests have been made, what files have been accessed, and what applications have been run. We can analyze these logs for evidence of malicious behavior or attacks.

Logging is similar to HIDS because both systems directly use information from the end host, avoiding many potential parsing inconsistencies and problems with encrypted traffic. However, like NIDS, logging may need to consider evasion attacks such as the path traversal attack, which requires smarter filesystem parsing and can lead to inconsistencies.

The biggest drawback to logging is that it cannot be done in real-time. By the time the log has been generated, the event that's being logged has already happened. This also means that if an attack has happened, a log-based system will only detect the attack after it has happened. This can be dangerous if the attack is immediately damaging. However, logs are still useful for detecting attacks after they've happened (better late than never).

In terms of cost, logging is usually cheap, because web servers already have built-in logging mechanisms. The only overhead is occasionally running an external script on those logs to search for evidence of attacks.

11.2 False Positives and False Negatives

There are two ways a detector can go wrong. A *false negative* occurs when an attack happens but the detector incorrectly reports that there is no attack. A *false positive* occurs when there is no attack, but the detector incorrectly reports that there is an attack. As an example, consider a fire alarm system. A false negative occurs if there is a fire but the fire alarm does not go off. A false positive occurs if there is no fire, but the fire alarm goes off.

It's easy to build a detector with a 0% false negative rate. Just report that there is an attack every single time. Then there will never be a case where your detector incorrectly reports that there is no attack. Similarly, a detector that never reports an attack will have a 0% false positive rate. Clearly, both of these are pretty useless detectors. In the real world, different detectors will have different false negative rates and false positive rates, and part of designing a good detector is balancing the two error rates. In general, as one error rate decreases, the other error rate will increase. Intuitively, to get fewer false positives, you must alert less often, which means you will also incorrectly fail to alert more often (higher

false negative rate). Similarly, to get fewer false negatives, you must alert more often, which means you will also incorrectly alert more often (higher false positive rate).

Suppose you have two detectors. Detector A has a false positive rate of 0.1% and a false negative rate of 2%. Meanwhile, Detector B has a false positive rate of 2% and a false negative rate of 0.1%. Which of these detectors is better? It depends on the cost of each type of error. Consider the fire alarm system–if the fire alarm gives you a false negative, then your building has burned down, but if the fire alarm gives you a false positive, then you've wasted an hour with the fire department. In this scenario, the false positive is probably less costly than the false negative, so you would probably prefer Detector B. In another scenario, a false positive might be more costly than a false negative, so you might prefer Detector A instead.

The quality of your detector also depends on the rate of attacks. Consider Detector A again. If we receive 1,000 requests a day and 5 of them are attacks, then the expected number of false positives is $0.1\% \times 995 \approx 1$ request per day. (995 requests are not attacks, and out of the non-attacks, 0.1% of them will incorrectly be reported as an attack.) However, now suppose we receive 10,000,000 (10 million) requests a day and 5 of them are attacks. Now the expected number of false positives is $0.1\% \times 9{,}999{,}995 \approx 10{,}000$ requests per day. Note that nothing has changed about the detector. The only thing that changed was the number of requests received per day (and thus the rate of attacks). However, in the second scenario, our detector is much less useful, because we have to handle 10,000 false positives every day. This example shows that accurate detection is very challenging when the rate of attacks is extremely low, because even a very good detector will flag so many false positives that it becomes impractical to manually review every single false positive. For more information on this phenomenon, read about the base rate fallacy.

11.3 Strategies of detectors

So far, we've talked about how detectors are installed and how to measure their effectiveness, but we haven't talked about how the detector actually analyzes network traffic to detect an attack. There are four main strategies for detecting an attack, each with their benefits and drawbacks.

11.3.1 Signature-based detection

The idea: Look for activity that matches the structure of a known attack.

Signature-based detection can be thought of as *blacklisting*–we maintain a list of patterns that are not allowed, and we detect if we see something in the list of disallowed patterns.

Example: We know that inputting some garbage bytes, followed by a memory address, followed by shellcode is the structure of a buffer overflow attack, so the detector can search for strings that match this pattern and classify them as attacks.

Pros:

- Detecting known signatures is easy.

- It's very good at detecting known attacks. Over time, the security community has built up huge shared libraries of attacks with known signatures.

Cons:

- It won't catch new attacks without known signatures.
- It might not catch variants of known attacks if the variant is different enough that the signature no longer matches. If the signature detector is too simple, it's easy to modify the attack slightly to circumvent the detector.

11.3.2 Anomaly-based detection

The idea: Develop a model of what normal activity looks like. Flag any activity that deviates from normal activity.

Anomaly-based detection can be thought of *whitelisting*–we maintain a list of normal patterns that are allowed, and we detect if we see something that is *not* in the list of allowed patterns.

Example: A C program expects user input. Most user input consists of letters, numbers, and symbols–things you would expect a user to type on a keyboard. We determine that normal activity is any input that can be typed on a keyboard, and flag any input that cannot be typed on a keyboard. If an attacker tries to input a buffer overflow attack with memory addresses and shellcode (raw bytes that often can't be typed on a keyboard), we detect that this doesn't match normal behavior and flag it as an attack.

Pros:

- It can catch new attacks that have never been seen before.

Cons:

- Defining normal behavior is difficult. What if you train a model for normal behavior on training data that includes attacks?
- A poor model might classify lots of attacks as normal, or classify lots of normal requests as attacks.

In general, anomaly-based behavior is mostly studied in academic papers but not widely deployed as a detection strategy.

11.3.3 Specification-based detection

The idea: Manually specify what normal activity looks like. Flag any activity that deviates from normal activity.

Specification-based detection is also a form of whitelisting. The main difference between specification-based detection and anomaly-based detection is that specification-based detection manually defines normal activity (instead of trying to learn a model for normal activity).

Example: A C programmer writes a program that asks for the user's age as input. The programmer knows that ages are numerical and specifies that normal behavior is inputting

a number. If an attacker tries to input a buffer overflow attack with memory addresses and shellcode (raw bytes that are not numbers), we detect that this doesn't match normal behavior and flag it as an attack.

Pros:

- It can catch new attacks that have never been seen before.
- If the specification is well-defined, the false positive rate can be made very low.

Cons:

- It's very time-consuming to manually write specifications for every application.

11.3.4 Behavioral detection

The idea: Look for evidence of compromise.

Unlike the other three models, behavioral detection doesn't search for attack patterns in the input. Instead, behavior detection looks for malicious behavior that an attacker might try to perform. In other words, we are looking for the result of the exploit, not the contents of the exploit itself.

Example: A C programmer writes a program that never calls the `exec` function. If an attacker tries to input a buffer overflow attack with shellcode that calls the `exec` function to spawn a shell, we detect that the code has called `exec` and flag this behavior as an attack. Note that we did not analyze the attacker input. Instead, we analyzed the program behavior and noticed that it called the `exec` function, which is evidence that the program has been compromised.

Pros:

- It can catch new attacks that have never been seen before.
- If the behavior rarely or never occurs in benign (non-attack) circumstances, the false positive rate can be made very low.
- It can be cheap to implement.

Cons:

- The attack is only detected after it's started, so there's no way to prevent the attack before it happens.
- An attacker can try to avoid detection by using different behavior to execute their attack.