CS 161 Computer Security

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

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 Intro to Internet

1.1 Internet layering

Consider sending a letter using snail mail. You'd seal your letter in an envelope with your friend's address on it. The post office delivers the letter to the address, where your friend can open the letter and read it.

The protocols for sending messages over the Internet follow a very similar structure. We generally talk about the network in terms of layers, based on the OSI 7-layer model. Higher layers, like the letter you wrote, contain richer information, and they are wrapped inside lower layers, like the envelope, which contain information about where and how the message should be sent. It might also help to view the layers as an onion, where each layer is peeled back to reveal the next higher layer.

The OSI model dates back to the late 1970s and is somewhat outdated, so we really only concern ourselves with 5 layers. The model is supposed to allow higher layers to abstract away any details at lower layers, but as we will see, many of the security protocols don't fit into one single layer.

7. Application. This is the human-readable content you want to send, such as the HTML of a webpage or the text of an email. The actual structure of this content depends on what exactly your application is.

4. Transport. This layer creates an end-to-end connection between your server and the destination server you want to communicate with. The two main options here are to use TCP, which guarantees that messages are sent in order, or UDP, which doesn't.

3. Network. This layer finds routes through the Internet in order to actually send messages. The IP address protocol is used here to give a global address to every location on the network.

2. Link. This layer breaks down the routes in the network layer into individual hops between local subnetworks. It takes many hops at the link layer in order to route a message.

1. Physical. This is the lowest layer, where individual bits are encoded with physical protocols such as voltage levels to send them over a link.

1.2 Dumb network

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 network layers, there is no concept of a connection. All the routers at the lower layers need to do is look at a packet and deliver it to the proper destination. In order to actually create a connection, we rely on the transport and application layers, which establish a connection by sending packets between you and the destination using protocols at those layers.

1.3 Network Adversaries

For a given connection, we are concerned with three separate types of adversaries. They are, from weakest to strongest:

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

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

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

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

2 Lower Layers: ARP, DHCP

2.1 Networking background: LANs, Ethernet

Computers in a small area (an office or a university campus, for example) connected through the link layer form a **local area network** (LAN).

The most common link layer is **Ethernet**, which assigns a 6-byte **MAC address** (Media Access Controller address) to each computer on the LAN. This is not to be confused with MACs (message authentication codes) from the crypto section, which we will rename as MICs (message integrity codes) for the networking unit. 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 frame to everyone on the local network".

Ethernet started as a broadcast-only network. Each node on the network could hear 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 frames 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 frames. 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]

2.2 ARP

2.2.1 Protocol

From unpacking the layer 3 (network) header, we have the global IP address of the destination. However, at the link layer, everything is addressed with local MAC addresses. Thus we need a way to translate global IP addresses into local MAC addresses. The protocol that does this is **ARP**, the **Address Resolution Protocol**.

Say Alice wants to send a message to Bob, and knows 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`."
3. Alice caches the IP address to MAC address mapping for Bob.

[1]A: On-path

If Bob is outside of the LAN, then the gateway would make response in step 2 with its MAC address.

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

2.2.2 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 Mallory's MAC address belongs to Bob. Now, Alice will send any messages intended for Bob 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.

2.2.3 Defenses: Switches

A simple defense against ARP spoofing is to use a tool like arpwatch, which tracks the IP address to MAC address pairings 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. VLANs also have the ability to configure a mirror port, which sends a copy of all packets transmitted to a specific port for network monitoring.

2.3 DHCP

2.3.1 Protocol

DHCP (**Dynamic Host Configuration Protocol**) handles the setup when a computer first joins a network. 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 www.google.com into an IP address (we will cover DNS in detail later)
- the IP address of the gateway, so you can contact the Internet

[2]A: Man-in-the-middle

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.

Notice that both client messages are broadcast. Client request must be broadcast so that all the servers know which one has been chosen. Sanity check: why must client discover be broadcast?[3]

Sanity check: How might an adversary attack this protocol? See next section for answer.

2.3.2 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 is sent quickly enough. The attacker now controls the client's gateway, which makes the attacker a man-in-the-middle, just like in ARP spoofing. The attacker can also become a man-in-the-middle by manipulating the DNS setting or by offering its own IP as the gateway address, which causes all packets sent to the victim to be rerouted to the attacker.

2.3.3 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 we lack a trusted foundation to build upon when we're first connecting to the network.

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

3 Layer 3: IP

IP (Internet Protocol) is basically the universal layer 3, designed to connect "networks of networks" by sending network packets. There are two primary versions, IPv4 and IPv6. For most of the class we only consider IPv4 but the protocols are generally similar. The biggest difference between v4 and v6 is the size of addresses and therefore the # of unique destinations available. For IPv4 the address is a 32b number, usually written as 4 integers between 0 and 255, such as 128.32.131.10. IPv6 instead supports 128b addresses, and they are generally written as 8, 4-byte hex values, such as cafe:f00d:d00d:1401:2414:1248:1281:8712. A single long run of 0 bytes in an IPv6 address can be replaced by two colons, so ::1 is really 0000:0000:0000:0000:0000:0000:0000:0001

IP also cares and routes by "subnets", groups of addresses with a common prefix indicated by # as the # of bits. So 128.32/16 is an IPv4 subnet which specifies 216 addresses, and 128.32.131/24 specifies 28 addresses. Routing generally proceeds on a subnet rather than individual IP basis.

When a client gets its configuration it is told its 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 local network, identified as having the same prefix, the client needs to directly discover the layer 2 address of the destination, a process we discuss later. Otherwise, it sends the packet onto the gateway address whose responsibility is to forward it on further towards the destination.

Past the gateway the packet passes onto the general Internet which is composed of cooperating ASs (Autonomous Systems), identified by unique ASNs (Autonomous System Numbers). Within an AS the packet can be routed by any mechanism the AS desires, usually involving a complicated set of preferences designed to minimize the AS's own cost. If the receiving AS owns that network it routes the packet to the final destination, otherwise between ASs routing is determined by BGP (the Border Gateway Protocol), passing the packet closer to the final destination.

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 what is the AS path that the packets would follow. The process continues and, if an AS has a choice between two advertisements it will generally select the shortest path (although actual BGP path selection is a fair bit more complicated).

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.

Overall, IP operates on "best effort". Packets are delivered whole, but can be delivered in any order and may be corrupted (although the lower layers and IPv4 both include checksums or CRC checks designed to detect corrupted packets).

There are some special IP addresses and network blocks. 127.0.0/24 and ::1 are "localhost", used to create ‘network’ connections to your own system. 10/8, 172.16/12, and 192.168/16 are the private IPv4 addresses. They are not routed on the Internet and can instead be used for internal purposes for NAT (Network Address Translation). Finally, 255.255.255.255 is the IPv4 broadcast address, sending to all computers within the local network.

4 Layer 4: TCP, UDP

4.1 TCP

TCP (Transmission Control Protocol) is the workhouse protocol at layer 4. It is a reliable, in-order, connection based stream protocol. That is, a client first creates a persistent connection to the server. Once established, the connection is reliable and in order: messages are received on the other side in the same order they were sent. Finally it provides a stream abstraction; TCP programs don't think in terms of packets or datagrams but just "send data" and "receive data".

TCP connections themselves are identified by a 5-tuple of (Client IP, Client Port, Server IP, Server Port, proto=TCP). A server listens for requests, usually on a set of "known ports". Examples include port 22 (ssh), port 80 (http), port 443 (https). Ports below 1024 are "reserved" ports and only a program running as root can listen on those ports, but anyone can send to those ports. The client tends to use "ephemeral" ports, just the next available port or even just a random port.

To initiate a connection a client sends a TCP SYN to the server. In sending the connection, the client chooses a random 32 byte initial sequence number. If the server decides to accept the request, it sends back a SYN+ACK packet (a packet with both the SYN flag and ACK flag set), with the acknowledgement set to the client's sequence + 1 and its own initial sequence number. The client then finally responds with an ACK, completing the "3-way handshake" and establishing the connection.

Once a connection is established, each side sends data and the other side acknowledges the data. If a packet is "dropped" (lost), the message won't be acknowledged and so the sender will simply retry sending the data. Of course, it could be the acknowledgement that got dropped in which case the receiver will ignore the duplicated data but resend the ACK. This is also an important part of TCP's "Congestion control"; under standard TCP, when drops occur, it assumes there is congestion and sends data at a slower rate.

To end a connection, one side sends a FIN, which is acknowledged and tells the other side "I won't send any more data, but I will accept more 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.

There is also an ability to abort a connection. If a 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". RSTs are not acknowledged as they usually mean "something went wrong", such as a program crashing or abruptly terminating a connection.

4.1.1 Attack: TCP Spoofing

Let's think about how a malicious adversary could attack the TCP protocol. First, recall that we have three threat models to consider:

Off-path Adversary: The off-path adversary cannot read or modify any messages over the connection. Therefore, to attack TCP communication between Alice and Bob, an off-path adversary must know or guess the values of the 4-tuple (Client IP, Client Port, Server IP, Server Port), as well as the sequence numbers currently being used by Alice and Bob in order to inject data into the communication.

On-path Adversary: The on-path adversary can read, but not modify messages. Since this adversary is able to observe the sequence numbers, IPs, and ports being used in the connection, an on-path adversary can *easily* inject messages into a TCP connection. 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 adversary has all the powers of the on-path adversary and can additionally modify and block messages sent by either party. As 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.

Notice that TCP by itself provides no confidentiality nor integrity guarantees. To prevent attacks like these, we look to TLS, which uses the cryptography you have learned in the last unit to provide a secure channel of communication.

4.2 UDP

UDP (user datagram protocol) is the unreliable counterpart to TCP. It is an unreliable datagram protocol, so applications send and receive discrete messages.

Like TCP, connections are identified by 5-tuple, but unlike TCP, UDP *offers no guarantees about reliability*. If a datagram is dropped, there is no attempt in UDP to recover and resend. Similarly, datagrams can be reordered. It is possible for datagrams to be larger than the underlying network's packet size, but this can sometimes introduce problems.

UDP is generally used when latency is a concern, such as for very fast protocols like DNS or for video games and voice applications where it is better to just miss a request than to stall everything waiting for a retransmission.

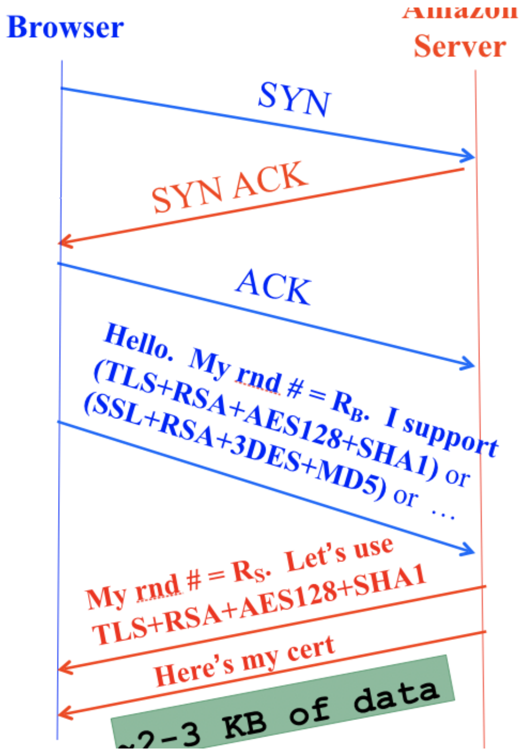
5 TLS

TLS (**Transport Layer Security**) is a protocol that provides an **end-to-end** 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, no one except the sender and receiver is able to read or modify the data being sent.

The Internet layer corresponding to TLS is not exactly clear. According to the OSI viewpoint, it would be an application at Layer 7, but in reality, it's more like Layer 6.5, since many applications use TLS to create an end-to-end encrypted channel, and then build the actual application on top of TLS. Examples of applications that use TLS are HTTP, which becomes HTTPS; 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 is built on top of TCP so that it can also guarantee messages are delivered reliably in the proper order. From the application viewpoint, TLS is effectively just like a TCP connection with additional security guarantees.

5.1 TLS Handshake



Browser
Amazon Server
SYN
SYN ACK
ACK
Hello. My rnd # = R_B. I support (TLS+RSA+AES128+SHA1) or (SSL+RSA+3DES+MD5) or …
My rnd # = R_S. Let's use TLS+RSA+AES128+SHA1
Here's my cert
~2-3 KB of data

Because it's built on top of TCP, the TLS handshake begins where the TCP handshake leaves off. 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**).

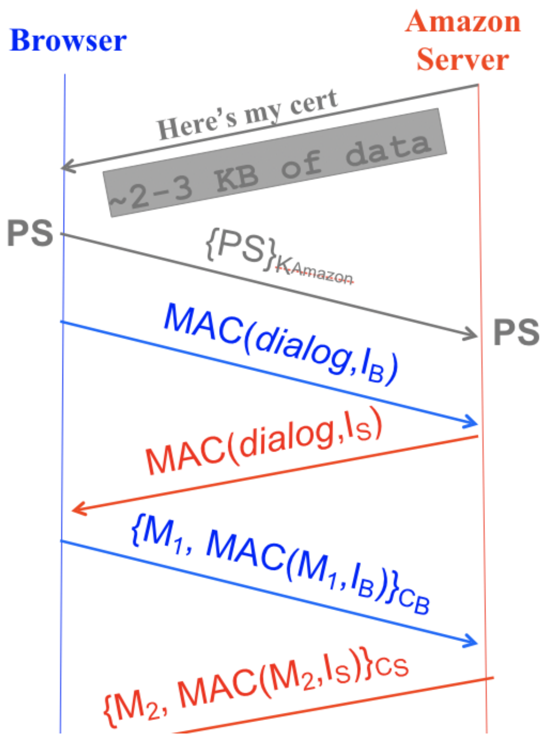
If the client trusts the CA signing the certificate (e.g. that CA is included in the Chrome browser's pinned list of trusted CAs), then the client can use the signature to verify the server's public key is correct. If the client doesn't directly trust the CA, it may need to verify a chain of certificates 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?[4]

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 generate a PS is to use an RSA key exchange, show in the second arrow here:



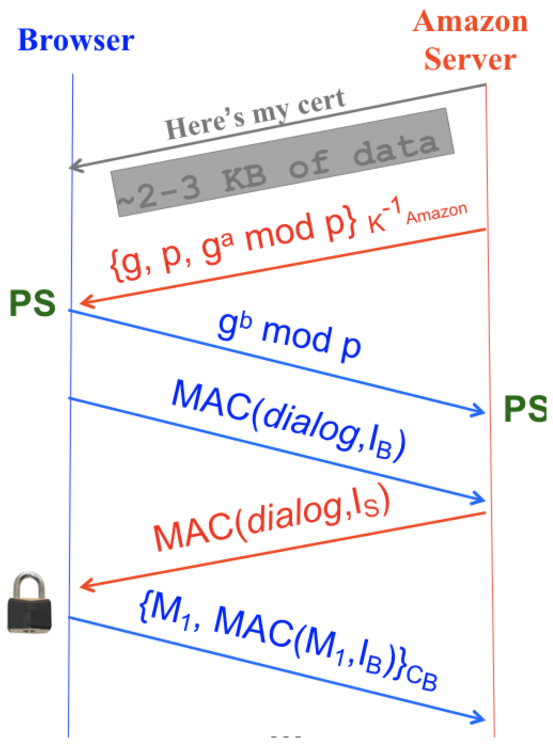
[4]A: No. An attacker can obtain the genuine server's certificate by starting its own TLS connection with the genuine server, and then present a copy of that certificate in step 2.

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?[5]

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:



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-

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

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?[6]

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.

5.2 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

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

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.

5.3 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.

6 DNS

The Internet is commonly indexed in two different ways. Humans refer to websites using human-readable names such as `http://google.com` and `http://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.

6.1 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 and has a fairly simple message format.

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) cannot read the ID field and must guess it when attacking DNS?[7]

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 (`NOERROR` for success, `NXDOMAIN` if the query asks about a non-existent name).

The next field specifies the number of questions asked (always 1 in practice). 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 contains `<Name, Class, Type>`, where `Name` is the actual key, `Class` is `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.

6.2 DNS Servers

In practice, your local computer usually delegates the task of DNS lookups to a **DNS Recursive Resolver**, 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

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

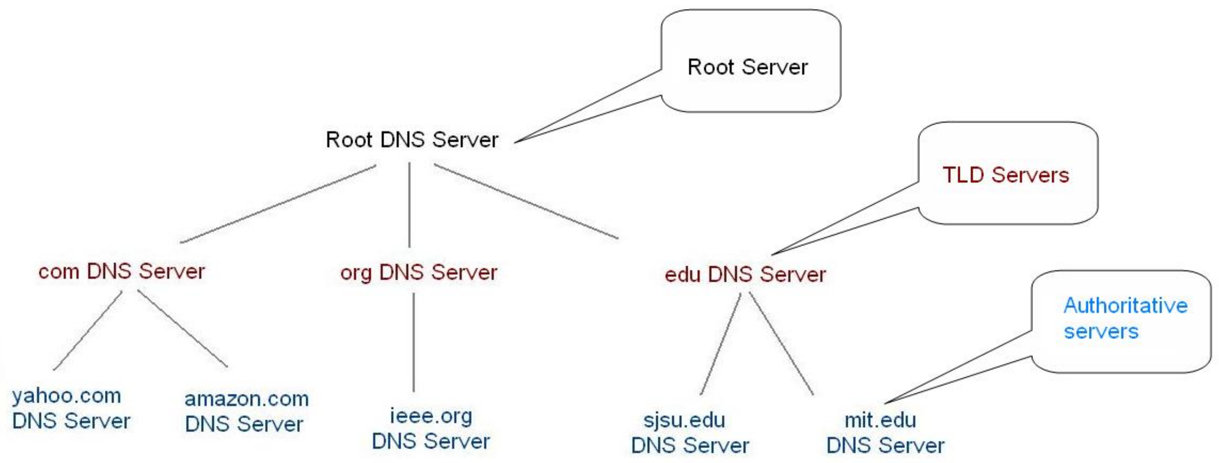
response. The recursive resolver is usually provided by your ISP and/or configured into your network connection by DHCP.

The **DNS Authority Servers** or **name servers** are servers on the Internet responsible for answering DNS queries. There is a special set of authority servers, the **root servers**, that are publicly known - you can see them for yourself here.

When we step through the process of a DNS lookup, we will be looking at messages sent between the recursive resolver and various authority servers.

6.3 DNS Lookup

If the Internet were small enough, we could let each of the root servers answer every DNS query directly, but with the current size of the Internet, this is clearly infeasible. Instead, DNS takes inspiration from search trees and answers queries recursively. You can think of the DNS name servers as being connected in a tree structure:



Every DNS query starts by asking one of the root servers: "Where is `eecs.berkeley.edu`?" Instead of answering directly, the root server will reply by redirecting you to the appropriate name server: "I don't know, but you can ask the `.edu` name server."

Since you don't have an answer yet, your next step is to ask the `.edu` name server: "Where is `eecs.berkeley.edu`?" The reply will take you one level further down the tree: "I don't know, but you can ask the `berkeley.edu` name server."

You still don't have an answer, so you ask the `berkeley.edu` name server: "Where is `eecs.berkeley.edu`?" Because you have reached the bottom of the tree, the `berkeley.edu` name server will respond: "`eecs.berkeley.edu` is located at `23.195.69.108`," completing the recursive DNS query.

There is one slight problem with this lookup process. DNS name servers, like websites, have names and IP addresses associated with them. When you receive a reply like "I don't know, but you can ask the `.edu` name server," you'd have to make another DNS query to figure out the IP address corresponding to the `.edu` name server. To avoid this circular problem, if a name server intends to redirect you to another name server, the reply must also tell you where that name server is located, e.g. "I don't know, but you can ask the `.edu` name server, located at `192.5.6.30`."

Now let's see a real DNS query in action. You can try this at home with the `dig` utility - remember to set the `+norecurse` flag so you can unravel the recursion yourself.

Again, every DNS query begins with the root server. We can look up the IP addresses of the root servers, although in a real recursive resolver these addresses are usually hardcoded.

```
$ 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 status (`NOERROR`), and the number of records returned.

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 is exactly what we queried for.

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 telling us "I don't know, but you can ask `a.edu-servers.net`, which is an `.edu` name server." 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 part of the response that tells us where to find the next name server to ask.

We saw two record types in this response: `A` type records map names to IP addresses, and `NS` type records map a DNS zone to a name server.

For completeness: the `172800` is the TTL (time-to-live) for each record, set at 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?[8]

```
$ 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 pointing us to `berkeley.edu` name servers, and `A` records in the additional section telling us where those name servers can be found.

```
$ 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
```

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

```
;; 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 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.

7 DNS Security

7.1 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.

```
$ 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 use **bailiwick checking**, which only allows a name server to provide records under its domain. This means that the `berkeley.edu` name server can only provide records for `berkeley.edu` (not `stanford.edu`), the `.edu` name server can only provide records for `.edu` domains (not `google.com`), and the root name servers can provide records for anything.

7.2 On-path attackers

Against an on-path attacker, DNS is completely insecure - everything is sent over plaintext, so an attacker simply needs to fill in the correct ID field and add malicious records and race to send the fake reply before the legitimate response. If the TTL is set to be very high, the victim will now associate those websites with attacker-controlled IP addresses 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, which limits the attacker to only a few tries per week. Since off-path attackers must guess the ID field with a $1/2^{16}$ probability of success, 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.

7.3 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 `cs161.berkeley.edu`:

```
$ dig cs161.berkeley.edu

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

;; QUESTION SECTION:
;cs161.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 `cs161.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://cs001.berkeley.edu/image.jpg"/>
<img src="http://cs002.berkeley.edu/image.jpg"/>
<img src="http://cs003.berkeley.edu/image.jpg"/>
...
```

The Kaminsky attack allows on-path attackers to race until their fake response arrives first and off-path attackers to additionally brute-force 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. (DNS doesn't specify what port the resolver uses to send queries, so source port randomization uses a random 16-bit source port for each query.) This decreases an off-path attacker's probability of success to $1/2^{32}$, which is harder but certainly not impossible.

Sanity check: How much extra security does source port randomization provide against on-path attackers?[9]

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

8 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?[10]

DNSSEC is designed as a **public key infrastructure** (PKI), which creates a chain of trust as you work your way down the DNS tree.

A chain of trust must start somewhere, so we assume that the root servers are trusted. Now, when a name server points us to one of its children, it must also *endorse* anything signed by that child. A simplified DNSSEC conversation:

1. Resolver to root server:

   "Where is `eecs.berkeley.edu`?"

2. Root server to resolver:

   "I don't know, try asking the `.edu` name server at `192.5.6.30`. I hereby endorse any message signed by the `.edu` name server. Signed, root."

3. Resolver to `.edu` name server:

   "Where is `eecs.berkeley.edu`?"

4. `.edu` name server to resolver:

   "I don't know, try asking the `berkeley.edu` name server at `128.32.136.3`. I hereby endorse any message signed by the `berkeley.edu` name server. Signed, `.edu` name server."

5. Resolver to `berkeley.edu` name server:

   "Where is `eecs.berkeley.edu`?"

6. `berkeley.edu` name server to resolver:

   "`eecs.berkeley.edu` is located at `23.195.69.108`. Signed, `berkeley.edu` name server."

How can we verify that the IP address provided in step 6 is correct? I know that the message in step 6 is signed by `berkeley.edu`, and in step 4, `.edu` endorsed anything signed by `berkeley.edu`. Thus `.edu` is endorsing the message in step 6.

How can I trust `.edu`'s endorsement in step 4? I know that the message in step 4 is signed by `.edu`, and in step 2, root endorsed anything signed by `.edu`. Thus root is endorsing `.edu`'s endorsement of the message in step 6. And since I trust root, I now trust the IP address in step 6.

What happens if an attacker tries to forge step 6? The attacker won't have `berkeley.edu`'s private signing key, so they have no way of creating a message that will be trusted by the

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

resolver. In fact, because every step is signed, as long as the attacker hasn't stolen any secret keys, they have no way of forging any message in the DNSSEC conversation.

To formalize the concepts of signatures and endorsements, we use digital signatures from the public-key cryptography unit. Signing works how you'd expect - for example, in step 2, "Signed, root" is actually a digital signature on the entire message using root's secret signing key.

How do we endorse messages signed by someone else? Remember that we use a public key to verify signatures, so if Alice wants to endorse Bob, Alice will sign *Bob's public verification key* using *Alice's secret signing key.* We can use Bob's public key to verify that the message was properly signed by Bob, and because Alice signed Bob's public key, we know that Alice is endorsing Bob.

8.1 DNSSEC query walkthrough

Now we're ready to see a full DNSSEC query in action.

```
$ 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        NS

;; 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      {cryptogoop}
edu.                 86400    IN   RRSIG   DS {cryptogoop}

;; 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
...
```

You might have noticed in the previous section, there was always one extra additional record that didn't show up in the additional section. This record corresponds to the `OPT` pseudo-section. This section 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.

The question, answer (blank), authority, and additional sections all contain the same records from regular DNS. The only difference is the extra two records in the authority section. The first of these is of type `DS` (**Delegated Signer**) and encodes the public key of the next name server we will talk to, in this case the `.edu` name server. The second of these is of type `RRSIG` and contains the signature of the public key in the `DS` record (notice the type `DS` in the value section of the record).

Sanity check: Whose secret signing key is used to generate the signature in the `RRSIG` record?[11]

The next query to the `.edu` name server is also identical to the original DNS query, with the addition of a `DS` record containing the `berkeley.edu` name server's public key, and a `RRSIG` record signing the `DS` record using the `.edu` name server's secret signing key.

The final query to the `berkeley.edu` name server will give us the `A` type answer record as before, along with an `RRSIG` type record signing the `A` type answer record using the `berkeley.edu` name server's secret signing key.

8.2 Nonexistent domains

DNSSEC works fine for existing domains, but encounters a problem if we want to sign nonexistent records. Remember that DNS is designed to be fast, so name servers can't afford to sign a message proving the domain is nonexistent on-demand. Also, online cryptography makes name servers vulnerable to an attack. Sanity check: what's the attack?[12]

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

[11]The DNS response is from root, so the `RRSIG` is signed with root's secret signing key.

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

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)
```

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.