Digital Signatures

MACs can provide integrity when two users share a key, however, we still aren't able to tell when a message is sent by a specific person. This is where **authentication** comes in, and is provided by digital signatures.

We want to have a message M "signed" via $S_M = \text{Sign}(S_K, M)$ with some secret key S_K. Anybody will then be able to take this S_M and verify that S_M is signed by the owner of the secret key corresponding to P_K. They know P_K but don't know S_K, yet can still verify that it was signed by the owner of P_K!

To construct this, we need a one-way function F_s which is everyone can compute one-way ($F_S(X)$) but only the person knowing S can find the inverse of: $F_S^{-1}(X)$. With this, we are equipped to define our signature scheme using RSA:

$$\text{Sign}(S_k, M) = M^{S_k} \mod n$$

To verify a signature S, we can raise $\text{Sign}(S_k, M) = M^{S_k} \mod N$ to the power of P_k. Recall that in RSA, the secret key is d and the public key is e. We know that $x^{ed} \equiv x \mod N$, so it follows that

$$(M^{S_k})^{P_k} \equiv M \mod N.$$

This could technically work for a signature, but there is one key modification we need to do first: M should be replaced with its hash $H(M)$. Therefore, our final signature is:

$$\text{Sign}(S_k, M) = H(M)^{S_k} \mod n$$

The verification follows the same idea. So, why does this work? Thinking back to our idea of some one-way F_S, we can see that $F(X) = X^{P_k} \mod N$ and $F^{-1}(X) = X^{S_k} \mod N$. If you recall from Note 4, it is computationally hard to recover S given $X^S \mod N$ (the discrete logarithm problem), so attackers that don't know S_x cannot find $F^{-1}(X)$.

To sign a message M, Alice would generate an RSA keypair and send $F^{-1}(H(M))$. Anyone who wanted to verify that this was true could find $F(F^{-1}(H(M))) = H(M)$. Remember, signatures don't provide confidentiality, so everyone knows M.

Why do we need $H(M)$? This is to ensure Mallory cannot just choose produce a signature on some random message. Consider the flawed scheme of $\text{Sign}(S_k, M) = M^{S_k} \mod n$. Mallory can choose any $x \mod N$ and pretend that $x = M^{S_k}$. The corresponding message would then be $x^{P_k} \mod N$, as $x^{P_k} = M^{S_k P_k} = M$. When somebody went to verify this, they would find that $(M^{S_k})^{P_k} \equiv (x)^{P_k}$.

We cannot set x^{P_k} to whatever we want, but it is still a valid forgery, and therefore this scheme is broken. To fix it, we introduce the hash function from earlier. For the same attack to work, Mallory would need to find $H(M) = (M^{S_k})^{P_k}$, which is infeasible given it's one-way nature.

Digitial Signature Algorithm

Instead of using RSA, we can use algorithms explicitly designed for digital signatures. The most popular today is the Digital Signature Algorithm (DSA) and its elliptic curve variant ECDSA.

First, we generate public primes p, q such that $p = aq + 1$ for integer a, i.e. $p - 1$ is a multiple of q. This is done in order to have a order q subgroup of p. We also find a generator $h \mod p$ and subgroup generator $g \equiv h^{\frac{p-1}{q}} \mod p$. g crucically has order $q \mod p$.

Next, we generate a random private key $x \mod p$ and public key $y \equiv g^x \mod p$.

To sign, we generate an ephermal signing key k and find $S_1 = (g^k \mod p) \mod q$. We then find $S_2 = k^{-1}(H(M) + xS_1) \mod q$. The signature is composed of (S_1, S_2).

To verify, we define $V_1 = H(M)S_2^{-1} \mod q$ and $V_2 = S_1 S_2^{-1} \mod q$, and check that

$$(g^{V_1} y^{V_2} \mod p) \mod q \equiv S_1$$

The proof of correctness is left as an exercise for homework. Note that k must be kept secret and **cannot** be reused. There exists a straightforward attack to recover the private key x given two different signatures using the same k (again an exercise for homework).

Cryptography in Network Security

Throughout the semester, we have explored many different types of cryptographic primitives (individual cryptographic algorithms/objects like RSA). Of course, these are most useful when combined to form a real-world secure system, as is the case with the Internet today. In order to protect you against eavesdropping and man-in-the-middle attacks, websites will use various protocols, most common among them **TLS**.

Transport Layer Security

TLS, also known as Transport Layer Security, is a high-level networking protocol that establishes a secure communication channel between a client (user trying to connect to the website) and server. There are many facets of this system, using a bunch of different cryptographic concepts. First, we want to establish the four main challenges of creating such a connection:

1) **Location**
   The client needs a way of locating the server given its URL.

2) **Identity**
   The client needs a way of verifying the server they found is actually the server they wish to

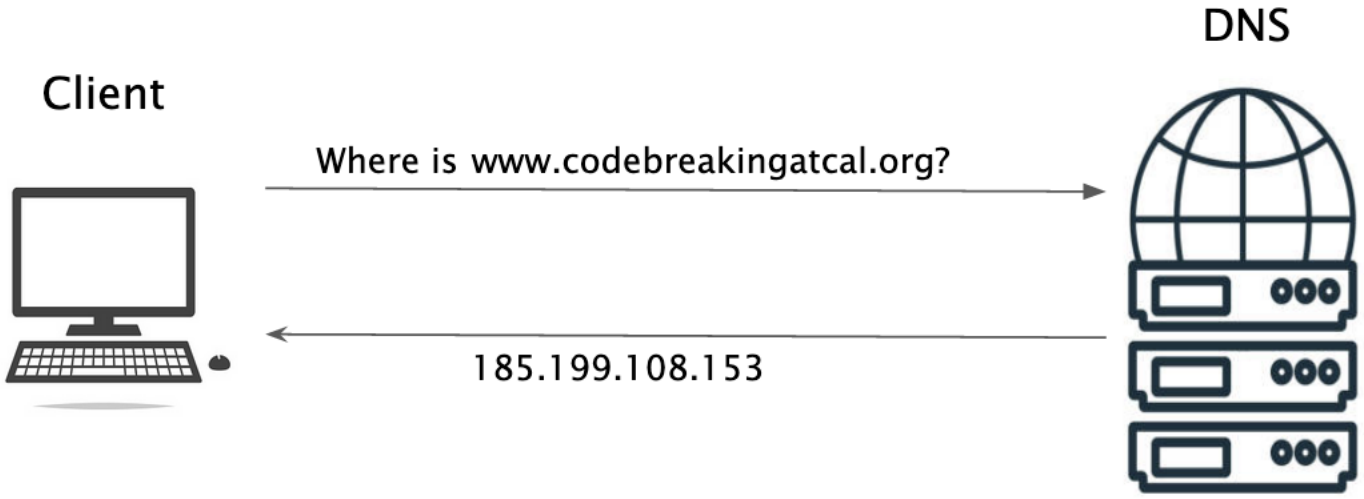
connect to.

3) **Confidentiality**
   The client and server need a way to communicate such that eavesdroppers cannot view what they are saying.

4) **Integrity**
   The client and server need a way to communicate such that their messages cannot be modified mid-transit.

Location

While mostly out of scope for this course, the problem of locating a server given its URL is handled by a **Domain Name System** (DNS) server. This server takes in a URL as an input and returns the IP address corresponding to it. Our client can then follow this IP address to the actual server.



Identity

The problem of verifying the identity of a server is chiefly solved through the use of **certificates**. A certificate is a message that says something along of the lines of "Server X uses URL Y and RSA public key Z". This is digitally signed by a **certificate authority**, which is a trusted server (usually hard-coded into a computer). Since we can trust the CA, we can trust that any message with their signature is true.

When we initially connect, the server sends over their signed certificate to the client, who can verify that it is signed by a trusted certificate authority. They can then trust that the server is not an imposter, and save their RSA public key for future use.

Confidentiality

Arguably the most complicated part of the process is in establishing a secure, encrypted channel once we have verified the identity of the server. We can assume the following has already been accomplished:

1. The client knows the server's address and RSA public key.
2. The server knows the client's RSA public key.

We need a way to exchange a shared **symmetric** key in order to establish a secure channel. To do so, we use **elliptic curve Diffie-Hellman (ECDH)** – we will define elliptic curves in the next note, but it is just a stronger version of Diffie-Hellman.. There are alternatives to ECDH for symmetric key sharing, but ECDH comes with unique advantages we will expand upon later.

For now, we will use the normal Diffie-Hellman version. The server generates the public parameters (generator, modulus) and sends a signed version to the client. The client can verify that the server sent these (verifying authenticity, integrity) and accept them as the agreed-upon parameters.

Each side generates their respective public values ($g^a \mod p$ etc), and signs the value before sending. This is so each side can verify the integrity and authenticity of the message as aforementioned. **If the messages are not signed, a man-in-the-middle attack can occur, in which an adversary makes the client and server interact with it as the supposed other party.**

Contributors:
- Ryan Cottone