Lecture 22: Network security and TLS

Anirudh Sivaraman

2018/12/03

This week we'll talk about network security. We'll start with Transport Layer Security (TLS). TLS provides a means to securely communicate across the Internet using TCP. What does this mean? What security properties might we want from a communication system? Here are the security properties TLS actually provides. Let's assume that we have a sender S and receiver R that are trying to communicate with each other.

1. *Authentication*: Can S be sure that it is really talking to receiver R and not an impersonator pretending to be R?
2. *Confidentiality*: Can S and R be sure that someone else can not access the content of their conversation?[1]
3. *Integrity*: Can S and R be sure that the message received by R was a bit-exact copy of the message sent out by S? In other words, can both S and R be sure that the message was not tampered with during transit?

Notably, TLS does not provide *privacy*: it does not hide the fact that S is communicating with R, which might be in many cases more valuable than *what* S and R are talking about.

1 Security in the physical world

Let's look at how these properties could be provided in a physical setting because the physical solutions provide a good analogy for the electronic ones used in TLS today. [2]

One way to provide authentication is to use an ID card that has been issued by an organization that you have faith in, e.g., the way a driving license is issued by the New York Department of Motor Vehicles. The DMV takes it upon itself to verify periodically that the driver is who they say they are and then issues an ID card that the driver can present to any other entity to authenticate themselves. One way to provide confidentiality is to lock the message within a box with a powerful lock that can not be broken into easily, but can be easily opened by a recipient who has the key for that lock. One way to provide integrity alone in the real world is to send the message in a clear laminated cover that is hard to tamper with.

2 Security in the electronic world and cryptography

Security in the electronic world builds on top of cryptography, a branch of theoretical computer science that mathematically formalizes security properties such as authentication, confidentiality, and integrity. We provided security in the physical world by making some physical action hard: hard to pick a lock (confidentiality), hard to tamper with a laminated cover (integrity), or hard to fake an ID (authentication). Similarly, hardness in the electronic world rests on certain computational/mathematical problems being hard to solve for any computationally bounded adversary who has access to a large, but finite, number of computers at their disposal.

One such problem is the prime factorization problem: the problem of breaking down an integer into a product of primes. It is suspected that this problem is hard. More specifically, it is suspected that the hardest integers

[1] More precisely, someone with bounded computational resources.

[2] This is just an analogy at best. Security is a very detail-heavy topic, and security details are more important than loose analogies if you're dealing with secure systems in the real world. That said, analogies are useful pedagogically.

to factorize are the semiprimes: integers that are a product of exactly 2 (possibly identical) primes. The largest semiprime that has been factored into its two constituent primes so far is a 768 bit long semiprime. This factorization took about 2000 years of compute time on a 2.2 GHz AMD Opteron and about 2 years of real time.

We won't worry about which mathematical problem is hard in this lecture and which mathematical problem forms the basis of different *cryptographic primitives* (or building blocks) that provide different security properties. Instead, we'll just provide the interface to these primitives and assume someone else has done the hard job of figuring out the right mathematical problem (i.e., prime factorization) and the right instance of that problem (i.e., which subprime). This is also the recommended approach when building secure systems because finding hard instances of hard mathematical problems is hard and error-prone even for cryptographers.

The primitives we'll be using are the following:

1. PubEncrypt(publickey, cleartext) $->$ ciphertext. This primitive takes a public key (a large string of bits) and a message such as "Hello, how are you" (called cleartext because its contents are visible for everyone to see) and turns it into a jumbled mess called the ciphertext. Ciphertext is unintelligable to everyone except the intended recipient, who has a private key (another large string of bits) corresponding to this public key.

2. PubDecrypt(privatekey, ciphertext) $->$ cleartext. This is the inverse of the PubEncrypt function that retrieves the cleartext from the ciphertext. Communicating entities have public-private key pairs that allows them to encrypt cleartext messages for the other end of the communication channel and decrypt ciphertext destined to them.

3. SymEncrypt(key, cleartext) $->$ ciphertext and SymDecrypt(key, ciphertext) $->$ cleartext are the symmetric analogues of PubEncrypt and PubDecrypt. Unlike the Pub functions that use a pair of *asymmetric* keys, the public and the private keys, to encrypt and decrypt communication, the Sym functions use a *single* key for both encryption and decryption. The use of a single key makes the Sym functions considerably faster; however, they suffer from the problem of having both parties agree on a shared key up front.

4. Sign(privatekey, message)$->$ signature allows an entity to sign any message (cleartext or ciphertext) with a private key to produce a signature that effectively endorses the message.

5. Verify(publickey, message, signature) $->$ True/False allows an entity to verify that a previous signature actually endorses the message that it claims to be endorsing.

6. Sign and Verify are primitives used in digital signatures. There are also symmetric versions of Sign and Verify that take the same symmetric key for both Sign and Verify. These are called Message Authentication Codes and are used to provide message integrity.

In addition, we'll be using the concept of a certificate, which is a combination of an entity's name, its public key, a signature of an endorser endorsing the fact that an entity by this name has this public key, and the name of the endorser as well.

3 The concepts behind TLS

Now that we have discussed several cryptographic primitives let's see how they come together in TLS. TLS operates on top of TCP: the TLS headers that are required to establish a secure communication channels are part of the TCP payload. This is similar to how HTTP operates on top of TCP. To make matters a bit confusing, the most common use of TLS is as a secure communication channel for HTTP itself called HTTPS. In this use case, the HTTP headers and payloads become the TLS payload. To understand this nesting of headers, you can load any web page that uses HTTPS (such as www.google.com) and use Wireshark to inspect the packets that were exchanged during the page load.

Before transmitting any data, the entities involved in a TLS connection exchange a sequence of special packets in a process called a TLS handshake. You can think of the TLS handshake as similar to the TCP

connection establishment process, where you need to accept a connection before you can transmit any data on it. TLS provides multiple security properties (authentication, confidentiality, and integrity) and requires multiple rounds of message exchange between the two entities to guarantee these properties. We won't get into the mechanics of these exchanges, but it is important to know that these multiple rounds do increase the latency of communication; a fair bit of research has gone into eliminating many of these rounds (or coalescing them with each other). Instead, we'll just provide some intuition for how TLS provides authentication, confidentiality, and integrity.

3.1 Authentication

First let's start with authentication. How does a sender know that it is talking to the correct receiver or the other way around? In practice, it is much more common that the sender (client) requires to know that the receiver (server) is not a fake than the other way around. This is what is required, for instance, when your laptop connects to google.com. You want to be sure that you are connecting to google.com, but Google doesn't necessarily need to know who you are or whether you are who you say you are. Client/sender-side authentication is usually established through other means such as passwords or dual-factor authentication.

OK. So how does a server like google.com authenticate itself to the client? This is where the concept of a certificate comes in. During the client-server handshake, the server sends to the client a certificate that has been signed by an endorser known as the certificate authority (CA), e.g., Symantec (previously Verisign). How do we trust the CA itself? The CA, in turn, presents the client with a certificate that is endorsed by a higher CA. This recursion needs to have a base case somewhere, so there is a set of *root CAs* that are trusted to sign their own certificates. As expected, if an organization running the root CA is compromised, all bets are off. A browser, such as Chrome, IE, or Firefox, is shipped with a list of root CA certificates that it trusts based on human trust in the physical world.

How does a server gets a certificate from an organization like Symantec? This process involves an application coupled with some manual verification to ensure that the server is who they say they are, e.g., I shouldn't be able to get a certificate saying I am amazon.com. If really required, this process can involve a notary public, a public officer who is authorized to sign off and endorse many important documents (e.g., a copy of your passport). You can think of notary publics as the human equivalent of a CA.

Because the process of obtaining and presenting certificates is tedious, it is unusual to use certificates to authenticate senders, even though certificates are a more secure mechanism than passwords. This is just one example of the constant tradeoff between security and usablity; it is important that secure systems are also easily usable because otherwise the inevitable consequence is the use of insecure alternatives.

3.2 Confidentiality

During the handshake, the client and the server can exchange their public keys to allow the client to encrypt data meant for the server alone and vice versa. Recall that the server already does this as part of presenting its certificate for authentication to the client. The client can similarly present its public key to the server. Once each entity has the public key of the other, confidentiality can be achieved by using the Pub primitives we discussed earlier.

However, public key encryption is slow. Hence the the Pub primitives are used to securely exchange a shared secret key between the client and the server. Some care has to be taken to ensure that the shared secret key isn't easily guessable by a snooping party in the middle. For instance, using a small shared secret key is a bad idea because it makes it more likely that it can be guessed. Once the shared secret key has been established securely by both the client and the server, they can use symmetric encryption using the shared secret key to guarantee confidentiality.

3.3 Integrity

We can use the Sign and Verify primitives to provide integrity. Concretely, the client signs its encrypted messages to the server using the symmetric key to endorse the message and vice versa. TLS operates in two modes: one

uses separate algorithms for integrity and confidentiality and another combines the two into a single algorithm because of errors that have been observed when implementing the two separately.

4 IPSec: Network-layer security

Another security solution in the networking world is IPSec. Unlike TLS, which operates by encrypting TCP payloads, but leaving the TCP header in cleartext, IPSec works by encrypting IP payloads, i.e., it encrypts the ports and options fields in the TCP/UDP header. Because the IP layer is connectionless—unlike TCP that establishes a connection before transmitting clear or ciphertext—the equivalent of the TCP handshake must happen separately. IPSec is used in many secure virtual private networks, where employees at different regional offices of a country are provided with the illusion that they are connected to the same physical company network with IP addresses in a contiguous range.

5 What could go wrong?

It may seem like using good cryptography built on top of hard instances of hard mathematical problems is all there is to building a secure communication channel. In practice, many things can go wrong—and have gone wrong. Here's a small list.

1. The certificate authority may not validate someone enough before issuing them a certficate. One example of this happened in 2001 when an imposter claiming to be a Microsoft employee got Verisign to issue them a certificate, which had to be countered by Microsoft issuing a bulletin saying that browsers should not trust that certificate: `https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2001/ms01-017`

2. The CA itself could be compromised, as was the case with the Dutch CA DigiNotar: `https://en.wikipedia.org/wiki/DigiNotar`. If this CA is part of the set of root CAs that the browser trusts unconditionally, the browser will end up trusting any certificate signed by this CA.

3. Private keys could be stolen by hacking into a machine using vulnerabilities in the machine's software. If a private key is stolen, it is no longer guaranteed that only one entity has access to the private key. This was one of the consequences of the Heartbleed vulnerability: `https://en.wikipedia.org/wiki/Heartbleed`.

4. The hard instance of a hard problem may not be hard anymore because computers have gotten faster over time, and the Internet and cloud computing allows us to recruit thousands of computers to work on a single task. For instance, the DES symmetric key encryption algorithm was proposed in 1975 and was considered secure for two decades. Then, in 1998, a set of 1856 custom silicon chips developed by the Electronic Frontier Foundation (EFF) was able to decrypt an encrypted DES message in less than a day by simply trying out every possible secret key. Other forms of such *brute-force attacks* have been demonstrated using the Internet as a large distributed computing platform in the late 1990s and cloud computing services such as Amazon as a large distributed computing platform since the 2010s.

6 Some philosophical thoughts on security

Unlike other aspects of systems and networking, security is a very black-and-white binary goal: given certain security properties that your system aspires to guarantee, it either provides them or not. There is very little in between. This is in contrast to quantitative metrics such as latency, throughput, loss rate, or availability where the latency could be high or low or where an availability of 99.99% is more useful than an availability of 99.999%.

As a result of this all-or-nothing nature of security, guaranteeing it is hard. A secure system is built out of many components and every one of these systems needs to be secure in order for the entire system to be secure.

Saltzer and Kaashoek provide a good discussion of these issues and many failed secure systems in their book:
https://ocw.mit.edu/resources/res-6-004-principles-of-computer-system-design-an-introduction-spring-20
online-textbook/protection_open_5_0.pdf

In the few examples we have seen so far, *the details matter* in security, probably more so than any other aspect of networking. A private key is only as secret or secure as the operating system it is housed in. The operating system is only as secure as the mechanism used to distribute the binary for the operating system or the binaries for OS updates. Any mistake anywhere can compromise security. Cryptography is only one piece of the puzzle. It is necessary but by no means sufficient: bad cryptography, such as using an easy instance of a hard problem will break your system, but good cryptography alone guarantees very little (e.g., if your private key is stolen, all bets are off).