Digital Signatures, Certificates, TLS, and Side Channel Attacks

Digital signatures

We often physically sign documents. Digital signatures are the electronic analog of that, done using public key cryptography.

The way public key cryptography usually works is that Alice generates a public and private key pair, other people encrypt messages with the public key, and Alice decrypts them with her private key. But things can be done in the reverse order, where Alice can "encrypt" things with her private key and people can "decrypt" them with the public key. If something is encrypted with the private key, and we decrypt it with the public key back into the original plaintext message, then we can be pretty sure that it must have been encrypted with the private key since the public and private keys are tied together. If the message had been encrypted with any private key other than Alice's, Alice's public key would not be able to decrypt it into the original message.

For a real-world analogy, suppose Alice has a special invisible ink pen that is only visible under a very specific frequency of light. She can sign papers using that pen, and we can verify it was Alice that signed by shining the right light on it. If a signature appears when we shine the light on it, then we know it was Alice that signed it, since only her pen would work with that light. In this analogy the pen is Alice's private key and the light is the corresponding public key. In cryptography, private and public keys are tied together in a similar way to how the pen and light are tied together.

We can use this to provide a sort of authentication. If Alice wants to prove that she has read a document, she can hash the document and encrypt the hash with her private key. This is a digital signature of the document. Others can verify this signature by decrypting with the public key and verifying that the decrypted hash matches the hash of the document. Only Alice's private key should have produced an encryption that the public key would be able to decode into the original hash. So we know that it was Alice's private key that signed the document.

RSA was for years the main digital signature algorithm, and its still widely used for certificates, but another algorithm, ECDSA (elliptic curve digital signature algorithm) is taking its place in newer applications.

A note about hashing It's important that the hash of the document be signed with the private key and not the original document. Public key cryptography is too slow to use on a large document, but it's fine for a short hash. Also, there are some insecurities with using something like RSA on an unhashed document, though they are a little more technical to get into than we want to here. Generally, the document is hashed and some padding is added before applying the private key.

Certificates

The last piece of modern cryptography concerns authentication. When we see a public key belonging to Alice, how do we know it belongs to Alice and not to someone impersonating her? This is the weak point of modern cryptography. Everything up to this point involves some pretty solid mathematics, but this part relies on humans, who are sometimes weak and lazy.

After Alice generates a public/private key pair, she goes to an entity called a *certificate authority* (CA). The CA is charged with verifying that Alice is who she claims to be, possibly by checking documentation she provides. They then certify that Alice's public key is valid by digitally signing it with their own private key. The CA's signature verifies that the public/private key pair really does belong to Alice. Alice's public key, the CA's signature, and a little other information, such as an expiration date, are grouped into something called a *certificate*.

Anyone can verify the certificate for themselves by checking the CA's signature. But this pushes the problem up another level: we now have to trust the CA. Who are they? How do we know that the private key they used to sign Alice's key really belongs to them? The answer is that the CA itself has to be verified via the same certificate process by a higher level CA. That then pushes the problem up to the higher-level CA. They also need to be verified. This can't keep going forever, and we end up getting a tree of CAs in a hierarchy. This tree ends at what

are called root certificate authorities. Root certificates are self-signed, which means, in the end, we just have to trust these certificates. They are typically built into browsers and operating systems, so we end up having to trust companies like Apple, Google, and Microsoft. This whole process is usually encompassed by the term *public key infrastructure* or PKI.

Having to trust big companies has led to problems. An interesting instance of this was the Lenovo Superfish scandal. Lenovo wanted a way to inject ads into HTTPs traffic. Because of certificates, they couldn't do that, so they had a special root certificate installed on some of the computers they made in 2014-5 which allowed them to man-in-the-middle HTTPs traffic to inject adds. However, the private key, which was the same on all the computers, was protected by a weak password. Once attackers had that password, they could man-in-the-middle connections as well.

Besides having to trust root certificates, another weak point of the system is the document verification process. Not all CAs are careful about this, and even if they are, a smart attacker could trick them. Some CAs have been hacked in the past and their private keys were used to create bogus certificates for sites like Google.

Viewing certificates Most browsers allow you to see a site's certificate. Often this works by clicking the icon next to the URL bar. Your browser does the job of verifying the certificates for the sites you visit. Occasionally, you'll get a certificate warning if something is off. A lot of times, it just comes from a site that misconfigured something with their certificate, but it could also come from someone trying to hijack a connection. Most users, unfortunately, will blindly click through the warning message.

Self-signed certificates You may occasionally see self-signed certificates where someone signs for themselves instead of going through a CA. Sometimes, people do this because they need a certificate in order to make things work, but they don't want to or need to go through the trouble of getting verified by a CA. If you see a self-signed certificate at a friend's private site where they're testing out some new product, likely that certificate is fine. But if you are visiting a major website and see a self-signed certificate, that could indicate someone is trying to man-in-the-middle your connection.

TLS

Transport Layer Security (TLS) is a widely used protocol that uses much of the cryptography we've covered to secure data. TLS is used to secure HTTP traffic. When HTTP is run over TLS, it is known as HTTPs (the "s" is for "secure"). TLS is also used to secure email protocols like SMTP, and some IoT devices use it to communicate securely. People are also starting to use TLS (via HTTPs) to secure DNS traffic.

Web traffic uses HTTP, which is an unencrypted protocol. User names, passwords, credit numbers, and everything else are all visible over HTTP to anyone sniffing network traffic network traffic. In the mid 1990s, a technology called SSL (Secure Sockets Layer) was developed to add security to HTTP. SSL went through a few versions and was eventually renamed TLS version 1.0. People still use terms SSL and TLS interchangeably. The current version of TLS is 1.3, though both versions 1.2 and 1.3 are in common use. Older versions have a variety of security problems and should be avoided. Version 1.2 has some problems, but those problems are not easily exploited. Version 1.3 simplifies 1.2 and fixes many of its problems.

TLS starts with a handshake. Let's look at the version 1.3 of the handshake. The client sends a hello message that contains, among other things, the *ciphersuites* it supports. This includes what encryption ciphers, signature algorithms, and message authentication types it can handle. In version 1.3, the key exchange is always elliptic curve Diffie-Hellman, and the client hello message will contain the client's public value (what we called A in earlier notes). In reply, the server sends its own hello message. This contains its own public value for Diffie-Hellman (the B value in the earlier notes). The server looks at the list of ciphersuites sent by the client, chooses the strongest one that it also supports, and sends its choice in its hello message. The server also presents its certificate at this time, which the client then verifies.

The command-line utility `curl` can be used to show the handshake when used with the `-v` option. Here part of the output from doing `curl -v https://www.google.com`.

```
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
```

To see more detailed information on the handshake, Wireshark can be used to capture a TLS session, and `openssl s_client` can be used at command line on a system that has OpenSSL installed. One thing you can see with these is the ciphersuite list for your browser. Below are a few of the couple dozen ciphersuites that my web browser supports. The first two are the strongest ones, and the last one is the weakest one.

```
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_3DES_EDE_CBC_SHA
```

TLS problems One type of attack on TLS is called a *downgrade attack*. This is where an attacker man-in-the-middles a TLS handshake and convinces both sides to use the weakest ciphersuite they both support. The attacker then uses some known attacks to break the encryption. For instance, 3DES in CBC mode is subject to something called a padding oracle attack. TLS version 1.3 takes some measures to prevent this attack, but it is a potential problem on older versions.

There have been several other attacks on SSL/TLS over the years. The worst was called Heartbleed. It involved a buffer overflow exploit in code from the openSSL library that allowed attackers to read a server's memory to extract key information. It's estimated that around 17% of the HTTPs servers were vulnerable to the attack by the time it was found in 2014.

Side-channel attacks

The encryption algorithms we have covered (DES, AES, RSA) are all mathematically pretty sound. The weakness is the programmers who implement the algorithms in code. If a programmer is not careful, they can introduce ways for an attacker to figure out a secret key without breaking the algorithm itself. Attacks of this sort, that rely on information obtained from observing the algorithm running, are called *side-channel attacks*. Some of them are especially devious.

To give a sense for how these work, consider a simple algorithm that tests if an input string matches a secret string. Here is pseudocode for what that string comparison actually does.

```
if length(input) != length(secret):
    return false
else:
    Loop over the characters of the input:
        if current character of input != current character of secret:
            return false
return true
```

If the two strings have a different length, then it will immediately return false. If they are the same length, then it goes through comparing character-by-character. So for strings of different length, the string comparison will be quicker than if they are of the same length. An attacker could try inputs of varying length until they find one for which the comparison takes noticeably longer. That will tell them the length. Then they start guessing starting characters A, B, C, or whatever until they get a comparison that takes longer. That indicates that they have the first character right (because then the program has to take time to check the second character, whereas

it returns false right away if the first characters don't match). They continue this way, guessing the second, third, etc. letters, until they have the whole string. The side channel here is the time it takes the string comparison algorithm to perform its job. To fix the problem, some code has to be added to make sure that all comparisons take the same amount of time.

This type of timing side channel can be used to break strong algorithms, like AES and RSA. For instance, in RSA, when we compute $C^d \bmod n$, the algorithm that does the raising to a power does its work based on the binary structure of the secret key d, doing more work for 1s and less work for 0s. Timing analysis can be used to figure out the key from there.

Also, since the CPU is working harder on those 1s than on the 0s, its power output varies. People can attach some sensors to measure the power output of the CPU and use that to figure out the key. An especially wild variation of this is acoustic cryptanalysis, which uses the high-pitched sounds the CPU puts out to figure out when it is working harder than other times. These attacks are not theoretical; they have actually been implemented.

Another interesting attack is AES cache timing. Here the term cache refers to an area of fast memory on the CPU. Memory accesses to things in the cache take less time than accesses to things stored in RAM. If one process does an AES encryption, after it's done, the cache will contain some values related to the encryption. Programs can't read the cache directly, but they can use the fact that cache accesses take less time than regular ones. This is called, probing, where the program will try out certain values and note how long it takes to access them. With enough of these probes, it can make a reasonable guess as to some things that are in the cache and figure out the key from there. One place this can happen is on a virtual private server (VPS). Often a single cloud computer hosts multiple different websites. A process running on behalf of one of those sites can use cache probing to figure out AES keys based on information left in the cache by other sites.

This is just a small sampling of side-channel attacks. There are many others based on things such as keystroke timing and electromagnetic radiation leaks from monitors.