Mechanics: HTTP vs HTTPS vs H2

In this lesson, we'll look at some mechanics pertaining to HTTP.

We'll cover the following

The evolution of HTTP

HTTP has seen two considerable semantic changes: `HTTP/1.0` and `HTTP/1.1`.

“Where are HTTPS and HTTP2,” you ask?

HTTPS and HTTP2 (abbr. H2) are technical changes, as they introduced new ways to deliver messages over the internet without heavily affecting the semantics of the protocol.

HTTPS is a secure extension to HTTP. It involves establishing a common secret between a client and a server, making sure we’re communicating with the correct party, and encrypting messages that are exchanged with the common secret. We will go over more on this later.

While HTTPS was aimed at improving the security of the HTTP protocol, H2 was geared towards speeding the process up. H2 uses binary rather than plaintext messages, supports multiplexing, and uses the HPACK algorithm to compress headers. To make a long story short, H2 was a performance boost to HTTP/1.1.

Websites owners were reluctant to switch to HTTPS since it involved additional round-trips between client and server (as mentioned, a common secret needs to be established between the two parties), slowing the user experience down. With H2, which is encrypted by default, there are no more excuses as features like multiplexing and server push make it perform better than plain HTTP/1.1.

HTTPS

HTTPS (*HTTP Secure*) aims to let clients and servers talk securely through TLS (Transport Layer Security), the successor to SSL (Secure Socket Layer).

The problem that TLS solves is fairly simple, and can be illustrated with a simple metaphor: Your significant other calls you in the middle of the day, while you're in a meeting, and asks you to tell them the password of your online banking account as they need to transfer money for your child's tuition. It is critical that you communicate it *right now*, otherwise you face the prospect of your child being turned away from school the following morning.

You are now faced with two challenges:

- **authentication**: ensuring you're actually talking to your significant other and not someone pretending to be them.
- **encryption**: communicating the password without your coworkers being able to understand it and write it down.

What do you do? This is the problem that HTTPS tries to solve.

Public key certificates

In order to verify who you're talking to, HTTPS uses Public Key Certificates that state the identity behind a particular server. When you connect to an IP address via HTTPS, the server behind that address will present you its certificate for you to verify their identity. Going back to our analogy, this could simply be you asking your significant other to tell you their social security number. Once you verify that the number is correct, you gain an additional level of trust.

This does not prevent attackers from learning the victim's social security number, stealing your partner's smartphone and calling you. How do we verify the identity of the caller?

Certificate authority

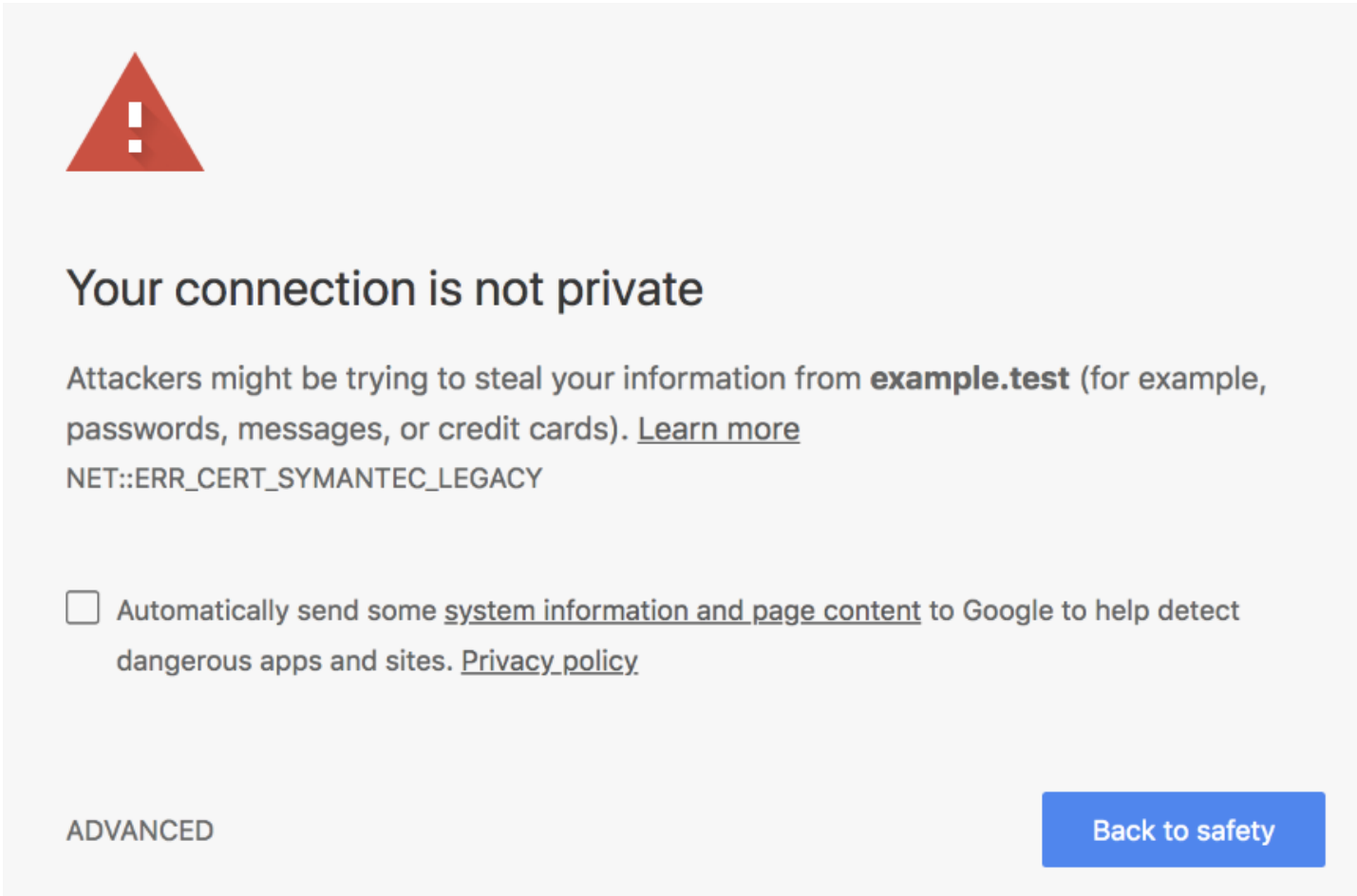
Rather than directly asking your partner to say their social security number, you make a phone call to your mom who happens to live next door and you ask her to go to your apartment to make sure your partner is saying their social security number. This adds an additional level of trust, as you do not consider your mom a

threat and rely on her to verify the identity of the caller.

In HTTPS terms your mom is called a CA, or Certificate Authority. A CA's job is to verify the identity behind a particular server and issue a certificate with its own digital signature. This means that, when I connect to a particular domain, I will be presented with a certificate generated by the CA and not the domain's owner, called a self-signed certificate.

An authority's job is to make sure they verify the identity behind a domain and issue a certificate accordingly. When you order a certificate, commonly called an *SSL certificate*, the authority might give you a phone call or ask you to change a DNS setting in order to verify you're in control of the domain in question. Once the verification process is complete, it will issue the certificate that you can then install on your webservers.

Clients will then connect to your servers and be presented with this certificate so that they can verify it is genuine. Browsers have relationship with CAs, in the sense that they keep track of a list of trusted CAs in order to verify that the certificate is in fact trustworthy. If a certificate is not signed by a trusted authority, the browser will display an informative warning to the users.



Google Chrome's warning when using an old Symantec certificate, which has been distrusted by the

browser

ℹ Chrome VS Symantec

As you might expect, the relationship between browser vendors and CAs is extremely important. If a vendor distrusts a particular authority, all their certificates are going to be flagged to the average user.

An interesting accident happened with Symantec and its subsidiaries in late 2017 when Google Chrome decided to distrust and phase out support for certificates the authority issued before a certain date. They did this by citing "*questionable website authentication certificates issued by Symantec Corporation's PKI. […] During the subsequent investigation, it was revealed that Symantec had entrusted several organizations with the ability to issue certificates without the appropriate or necessary oversight, and had been aware of security deficiencies at these organizations for some time*".

In other words, this meant Google decided Symantec was issuing certificates without the right background checks, resulting in Chrome displaying a security warning when browsing websites that used old Symantec certificates.

In the next lesson, we'll study encryption.