Rob Heaton

Software Engineer

One track lover/Down a two-way lane

How does HTTPS actually work?

27 Mar 2014

HTTPS is simply your standard HTTP protocol slathered with a generous layer of delicious SSL/TLS encryption goodness. Unless something goes horribly wrong (and it can), it prevents people like the infamous Eve from viewing or modifying the requests that make up your browsing experience; it's what keeps your passwords, communications and credit card details safe on the wire between your computer and the servers you want to send this data to. Whilst the little green padlock and the letters "https" in your address bar don't mean that there isn't still ample rope for both you and the website you are viewing to hang yourselves elsewhere, they do at least help you communicate securely whilst you do so.

1. What is HTTPS and what does it do?

HTTPS takes the well-known and understood HTTP protocol, and simply layers a SSL/TLS (hereafter referred to simply as "SSL") encryption layer on top of it. Servers and clients still speak exactly the same HTTP to each other, but over a secure SSL connection that encrypts and decrypts their requests and responses. The SSL layer has 2 main purposes:

- Verifying that you are talking directly to the server that you think you are talking to
- Ensuring that only the server can read what you send it and only you can

read what it sends back

The really, really clever part is that anyone can intercept every single one of the messages you exchange with a server, including the ones where you are agreeing on the key and encryption strategy to use, and still not be able to read any of the actual data you send.

2. How an SSL connection is established

An SSL connection between a client and server is set up by a handshake, the goals of which are:

- To satisfy the client that it is talking to the right server (and optionally visa versa)
- For the parties to have agreed on a “cipher suite”, which includes which encryption algorithm they will use to exchange data
- For the parties to have agreed on any necessary keys for this algorithm

Once the connection is established, both parties can use the agreed algorithm and keys to securely send messages to each other. We will break the handshake up into 3 main phases - Hello, Certificate Exchange and Key Exchange.

1. *Hello* - The handshake begins with the client sending a ClientHello message. This contains all the information the server needs in order to connect to the client via SSL, including the various cipher suites and maximum SSL version that it supports. The server responds with a ServerHello, which contains similar information required by the client, including a decision based on the client’s preferences about which cipher suite and version of SSL will be used.

2. *Certificate Exchange* - Now that contact has been established, the server has to prove its identity to the client. This is achieved using its SSL certificate, which is a very tiny bit like its passport. An SSL certificate

contains various pieces of data, including the name of the owner, the property (eg. domain) it is attached to, the certificate's public key, the digital signature and information about the certificate's validity dates. The client checks that it either implicitly trusts the certificate, or that it is verified and trusted by one of several Certificate Authorities (CAs) that it also implicitly trusts. Much more about this shortly. Note that the server is also allowed to require a certificate to prove the client's identity, but this typically only happens in very sensitive applications.

3. *Key Exchange* - The encryption of the actual message data exchanged by the client and server will be done using a symmetric algorithm, the exact nature of which was already agreed during the Hello phase. A symmetric algorithm uses a single key for both encryption and decryption, in contrast to asymmetric algorithms that require a public/private key pair. Both parties need to agree on this single, symmetric key, a process that is accomplished securely using asymmetric encryption and the server's public/private keys.

The client generates a random key to be used for the main, symmetric algorithm. It encrypts it using an algorithm also agreed upon during the Hello phase, and the server's public key (found on its SSL certificate). It sends this encrypted key to the server, where it is decrypted using the server's private key, and the interesting parts of the handshake are complete. The parties are sufficiently happy that they are talking to the right person, and have secretly agreed on a key to symmetrically encrypt the data that they are about to send each other. HTTP requests and responses can now be sent by forming a plaintext message and then encrypting and sending it. The other party is the only one who knows how to decrypt this message, and so Man In The Middle Attackers are unable to read or modify any requests that they may intercept.

3. Certificates

3.1 Trust

At its most basic level, an SSL certificate is simply a text file, and anyone with a text editor can create one. You can in fact trivially create a certificate claiming that you are Google Inc. and that you control the domain gmail.com. If this were the whole story then SSL would be a joke; identity verification would essentially be the client asking the server "are you Google?", the server replying "er, yeah totally, here's a piece of paper with 'I am Google' written on it" and the client saying "OK great, here's all my data." The magic that prevents this farce is in the digital signature, which allows a party to verify that another party's piece of paper really is legit.

There are 2 sensible reasons why you might trust a certificate:

- If it's on a list of certificates that you trust implicitly
- If it's able to prove that it is trusted by the controller of one of the certificates on the above list

The first criteria is easy to check. Your browser has a pre-installed list of trusted SSL certificates from Certificate Authorities (CAs) that you can view, add and remove from. These certificates are controlled by a centralised group of (in theory, and generally in practice) extremely secure, reliable and trustworthy organisations like Symantec, Comodo and GoDaddy. If a server presents a certificate from that list then you know you can trust them.

The second criteria is much harder. It's easy for a server to say "er yeah, my name is er, Microsoft, you trust Symantec and er, they totally trust me, so it's all cool." A somewhat smart client might then go and ask Symantec "I've got a Microsoft here who say that you trust them, is this true?" But even if Symantec say "yep, we know them, Microsoft are legit", you still don't know whether the server claiming to be Microsoft actually is Microsoft or something much worse. This is where digital signatures come in.

3.2 Digital signatures

As already noted, SSL certificates have an associated public/private key pair. The public key is distributed as part of the certificate, and the private key is kept incredibly safely guarded. This pair of asymmetric keys is used in the SSL handshake to exchange a further key for both parties to symmetrically encrypt and decrypt data. The client uses the server's public key to encrypt the symmetric key and send it securely to the server, and the server uses its private key to decrypt it. Anyone can encrypt using the public key, but only the server can decrypt using the private key.

The opposite is true for a digital signature. A certificate can be "signed" by another authority, whereby the authority effectively goes on record as saying "we have verified that the controller of this certificate also controls the property (domain) listed on the certificate". In this case the authority uses their private key to (broadly speaking) encrypt the contents of the certificate, and this cipher text is attached to the certificate as its digital signature. Anyone can decrypt this signature using the authority's public key, and verify that it results in the expected decrypted value. But only the authority can encrypt content using the private key, and so only the authority can actually create a valid signature in the first place.

So if a server comes along claiming to have a certificate for Microsoft.com that is signed by Symantec (or some other CA), your browser doesn't have to take its word for it. If it is legit, Symantec will have used their (ultra-secret) private key to generate the server's SSL certificate's digital signature, and so your browser use can use their (ultra-public) public key to check that this signature is valid. Symantec will have taken steps to ensure the organisation they are signing for really does own Microsoft.com, and so given that your client trusts Symantec, it can be sure that it really is talking to Microsoft Inc.

3.3 Self-signing

Note that all root CA certificates are “self-signed”, meaning that the digital signature is generated using the certificate’s own private key. There’s nothing intrinsically special about a root CA’s certificate - you can generate your own self-signed certificate and use this to sign other certificates if you want. But since your random certificate is not pre-loaded as a CA into any browsers anywhere, none of them will trust you to sign either your own or other certificates. You are effectively saying “er yeah, I’m totally Microsoft, here’s an official certificate of identity issued and signed by myself,” and all properly functioning browsers will throw up a very scary error message in response to your dodgy credentials.

This puts an enormous burden on all browser and OS publishers to trust only squeaky clean root CAs, as these are the organisations that their users end up trusting to vet websites and keep certificates safe. This is not an easy task.

3.4 What are you trusting?

It’s interesting to note that your client is technically not trying to verify whether or not it should trust the party that sent it a certificate, but whether it should trust the public key contained in the certificate. SSL certificates are completely open and public, so any attacker could grab Microsoft’s certificate, intercept a client’s request to Microsoft.com and present the legitimate certificate to it. The client would accept this and happily begin the handshake. However, when the client encrypts the key that will be used for actual data encryption, it will do so using the real Microsoft’s public key from this real certificate. Since the attacker doesn’t have Microsoft’s private key in order to decrypt it, they are now stuck. Even if the handshake is completed, they will still not be able to decrypt the key, and so will not be able to decrypt any of the data that the client sends to them. Order is maintained as long as the attacker doesn’t control a trusted certificate’s private key. If the client is somehow tricked into trusting a certificate and public key whose private key is controlled by an attacker, trouble begins.

4. Really really fun facts

4.1 Can a coffee shop monitor my HTTPS traffic over their network?

Nope. The magic of public-key cryptography means that an attacker can watch every single byte of data exchanged between your client and the server and still have no idea what you are saying to each other beyond roughly how much data you are exchanging. However, your normal HTTP traffic is still very vulnerable on an insecure wi-fi network, and a flimsy website can fall victim to any number of workarounds that somehow trick you into sending HTTPS traffic either over plain HTTP or just to the wrong place completely. For example, even if a login form submits a username/password combo over HTTPS, if the form itself is loaded insecurely over HTTP then an attacker could intercept the form's HTML on its way to your machine and modify it to send the login details to their own endpoint.

4.2 Can my company monitor my HTTPS traffic over their network?

If you are also using a machine controlled by your company, then yes. Remember that at the root of every chain of trust lies an implicitly trusted CA, and that a list of these authorities is stored in your browser. Your company could use their access to your machine to add their own self-signed certificate to this list of CAs. They could then intercept all of your HTTPS requests, presenting certificates claiming to represent the appropriate website, signed by their fake-CA and therefore unquestioningly trusted by your browser. Since you would be encrypting all of your HTTPS requests using their dodgy certificate's public key, they could use the corresponding private key to decrypt and inspect (even modify) your request, and then send it onto it's intended location. They probably don't. But they could.

Incidentally, this is also how you use a proxy to inspect and modify the otherwise inaccessible HTTPS requests made by an iPhone app.

4.3 So what happened with Lavabit and the FBI?

Lavabit was Edward Snowden’s super-secure email provider during the NSA leaks insanity of 2013. As we’ve seen, no amount of standard hackery could allow the FBI to see any data on its way between Lavabit and its customers. Without the private key for the Lavabit SSL certificate, the agency was screwed. However, a helpful US judge told the Lavabit founder, Ladar Levison, that he had to hand over this key, effectively giving the FBI free reign to snoop traffic to its heart’s content. Levison made a valiant attempt to stall by handing over the 2,560 character key in 11 hard copy pages of 4-point type, but was slammed with an order requiring him to hand over the key in a useful format or face a $5,000/day fine until he did.

Once he complied, GoDaddy, the Lavabit CA, revoked the certificate, having (correctly) deemed it compromised. This added the Lavabit certificate to a Certificate Revocation List (CRL), a list of discredited certificates that clients should no longer trust to provide a secure connection. Compromised, self-signed or otherwise untrustworthy certificates cause browsers to display a big red error message and to either discourage or outright prohibit further actions by the user. Unfortunately, browsers will continue to trust a broken certificate until they pull the newest updates to the CRL, a process which is apparently imperfect in practice.

5. Conclusion

HTTPS is not unbreakable, and the SSL protocol has to evolve constantly as new attacks against it are discovered and squashed. But it is still an impressively robust way of transmitting secret data without caring who sees your messages. There are of course many implementation details not mentioned here, such as the exact format and order of the handshake messages, abbreviated handshakes to pick up recent sessions without having to renegotiate keys and cipher suites, and the numerous different encryption options available at each stage. The key thing to remember is that whilst HTTPS keeps data safe on the wire to its destination, it in no way protects you (as a user or a developer) against XSS or database leaks or any of the other things-that-go-bump-in-the-night. Be happy that it's got your back, but stay vigilant. In the immortal words of Will Smith, "Walk in shadow, move in silence, guard against extra-terrestrial violence."

If you enjoyed this, you'll probably enjoy my post explaining the details of 2015's FREAK vulnerability in SSL.

More posts on Security

- A tale of love, betrayal, social engineering and Whatsapp
- The SSL FREAK vulnerability explained
- Fun with your friend's Facebook and Tinder sessions
- Cookieless user tracking for douchebags
- The Padding Oracle Attack - why crypto is terrifying
- How to hack a Rails app using its secret_token

‹ Lessons From A Silicon Valley Job Search

Downfall of a once readable blog post ›

Archive / RSS / Email / Twitter