Overview of SSL/TLS

Luke Anderson

luke@lukeanderson.com.au

12th May 2017

University Of Sydney

Overview

1. Introduction

1.1 Raw HTTP

1.2 Introducing SSL/TLS

2. Certificates

3. Attacks

Introduction

Raw HTTP is Insecure

In stock standard HTTP, everything goes across the line in plaintext.

This leaves connections vulnerable to:

- message tampering
- eavesdropping on connections
- man-in-the-middle attacks

On top of that, there are numerous flaws in the underlying TCP/IP protocols.

Introducing SSL/TLS

Transport Layer Security (TLS) with its predecessor Secure Socket Layer (SSL) are cryptographic protocols for secure communication.

They use:

- Asymmetric cryptography for authentication
- Symmetric encryption for confidentiality of messages
- MACs for integrity

HTTP sits on top of that to produce HTTPS (HTTP Secure).
The only information that leaks is the IP address and TCP port you're connecting to, as well as the size of the messages you're sending and receiving.

While previously only considered for "secure operations", now it's suggested to be used everywhere by default.

TLS Protocol

Client → Server: (SSL version, available ciphers, other info)
Server → Client: (SSL version, select cipher, certificate)

Client authenticates the server (messages sent should be signed by cert).

Client (possibly in conjunction to server, depending on cipher) creates the pre-master secret for the session, encrypts using server's cert, sends to the server.

(Optional) Server may authenticate the client using client's certificate.
All messages from this point are encrypted.

Certificates

Where do the Certificates come from?

How do we decide whether to trust the certificate the server sends?

- A **Certification Authority** (CA) is a trusted third party that issues digital certificates.
- Certificates for CAs are shipped with operating systems and browsers, and other software.
- Each time a server sends a certificate to a browser, the browser will check to see if one of the CAs it trusts has signed the certificate.

Who are these CAs?

- Small number of multinational companies, with a significant barrier to entry.
- Recent player: `https://letsencrypt.org`.

Issues with Certification Authorities

Do we actually trust them?

- Many have poor security practises, and are willing to co-operate with governments.
- The small number of root CAs allow other CAs to sign on their behalf.
- In 2011, fraudulent certificates obtained from Comodo were used for MITM attacks in Iran — they can man-in-the-middle *any* website they want.

They sell based on ridiculous “features”:

- Cost of 128-bit and 256-bit certificates are commonly different, even though next to no extra work goes into the second.

Saying “this website secured by SSL” gives a false sense of confidence.

- Very easy for scammers to get themselves an SSL certificate for a domain they own.

How do you acquire a certificate?

Domain Validation: The CA determines you own the domain by one of:

- Having you respond to an email sent to `admin@`, `postmaster@`, etc.
- Having you publish a certain DNS TXT record.
- ... and more.

Issue with domain validation: do any of the above methods actually prove that you *own* the domain?

Extended validation certificates

- CA verifies you pass a set of identity verification criteria.
- Costs more than a normal certificate.
- Puts a green bar at the top of your browser.

Extended validation certificates still don't stop a faked normal certificate from intercepting traffic.

How does it work?

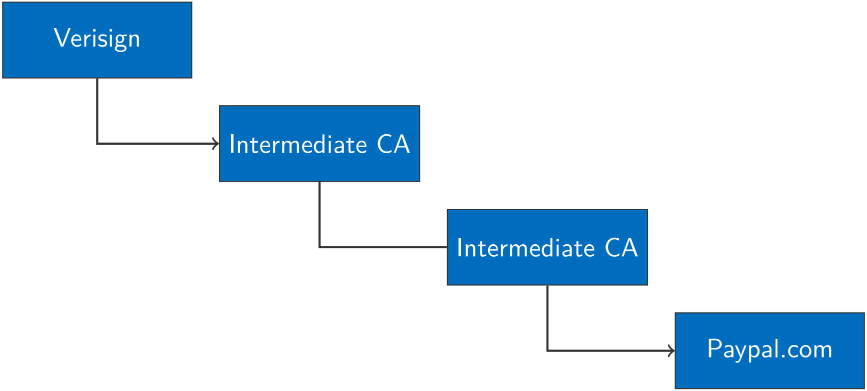
1. Generate a public/private keypair for your server.
2. Generate a Certificate Signing Request (CSR), containing domain name, public key, and other relevant details, and send this to a CA.
3. CA confirms any necessary details (domain validation or extended validation).
4. The CA signs the certificate and sends it back.
5. Install the certificate on your server.

Version	3
Issuer	GlobalSign
Validity	2015-12-11 to 2016-12-11
Common Name	`*.wikipedia.org`
Public Key	Elliptic: `04:cb:···:0b:fd`
Signature algorithm	PKCS#1 SHA-256 RSA
Signature	`b2:c6:···:39:fd`

An abridged copy of Wikipedia's current certificate. The last two fields are added by the CA.

Certificate Chaining

Root CAs can give permission to other CAs to sign SSL certificates.



Certificate Revocation

There is a method to revoke SSL certificates, called **Online Certificate Status Protocol** (OCSP).

1. Whenever a browser sees a new SSL cert, it makes a request to the OCSP URL embedded in the CA's signing certificate.
2. The OCSP server sends a signed response indicating whether the certificate is still valid.
3. The signature on the response only covers *some* of the response data, and does not include the response status. Hence a MITM can send back any response status.

The outcome of this is that attackers can intercept an OCSP request and send a *tryLater* response status. All browsers think this is fine (they can't tell otherwise) and avoid raising an issue.

Even if SSL/TLS were perfect...

There are still many attacks on SSL/TLS, even assuming the transport itself is perfect.

Many take advantage of users and the inability of browsers to recommend the correct course of action when things “aren’t right”.

Recommended viewing: the DEFCON presentation “More tricks for defeating SSL in practice” by Moxie Marlinspike.

Attacks

Beast and RC4

BEAST (Browser Exploit against SSL/TLS)
A practical exploit taking advantage of a protocol flaw with CBC in TLS.

Implication
If an attacker can read and inject packets quickly, then they can eavesdrop on any SSL connection using *-CBC (i.e. AES-CBC).

This is very useful in local networks. (e.g. WiFi in shops, LAN in labs).

TLS1.1 (2006) fixed, but was not widely adopted.
Now TLS1.2 soon to be 1.3

From Experts

○ Heavily suggested to use **RC4**.

○ RC4 is a stream cipher, doesn't require CBC.

Beast and RC4

- RC4 is **quicker** and **cheaper** than AES-CBC.
- Heavily adopted by Google.

Issues

- Minor:
  - Government Standards require AES-CBC not RC4.
- **Major:**
  - RC4 has a **bias** (preference towards 0 or 1 depending on input).
  - This happens in the first **256 bytes** of ciphertext.
  - Requires many (2^{32}) RC4 ciphertexts to exploit if it is random.
- **Worse:**
  - What lies at the first few bytes of SSL/TLS?
  - **THE COOKIE!**
    - If you steal the cookie, then game over!

The CRIME Attack (2012)

The CRIME[1] (Compression Ratio Info-leak Made Easy) attack is based on taking advantage of compression.

1. Rationale: make the protocol use less traffic by compressing data before encrypting it. (SPDY, TLS-compression)
2. If encrypted data is totally opaque, surely encrypted compressed data is too, right?
3. *Wrong!* The size of the encrypted payload is visible, and compression makes size give away quite a lot about the payload.

The above observation can be turned into an attack on protocols that compress and then encrypt.

- It is common to have cookies storing an authentication token.
- Cookies are sent with every request (in the HTTP headers).
- What if the attacker controls part of that request?

[1] It's not a real vulnerability if it doesn't have a lame backronym

The CRIME Attack

Suppose an attacker can get a victim to send a message of the form $E_k(\text{Compress}(m \parallel a))$ to a server, where a is controlled by an attacker, and m contains authentication data (like cookies, for example), but k and m are unknown to the attacker.

$$\begin{aligned}
\text{Let } a &= \text{"}\texttt{sessionKey=a}\text{"} \implies \text{Size}(E_k(\text{Compress}(m \parallel a))) = 100 \\
a &= \text{"}\texttt{sessionKey=b}\text{"} \implies \text{Size}(E_k(\text{Compress}(m \parallel a))) = 100 \\
\dots \\
a &= \text{"}\texttt{sessionKey=f}\text{"} \implies \text{Size}(E_k(\text{Compress}(m \parallel a))) = 95
\end{aligned}$$

Now the attacker knows it is likely that the string “`sessionKey=f`” occurs somewhere in the message m. Now the attacker repeats for “`sessionKey=fa`”, “`sessionKey=fb`”, and so on.

The POODLE Attack (2014)

The POODLE (Padding Oracle On Downgraded Legacy Encryption) attack is a *padding oracle* attack against SSLv3.

1. In CBC mode, padding bytes must be appended before encryption.
2. The last byte of the padding block is known to the attacker (a function of message length).
3. SSLv3 does not specify the contents of the other padding bytes, so the server cannot check these.
4. SSLv3 uses MAC-then-Encrypt, rather than Encrypt-then-MAC.

Details of the attack are explained here.

The outcome is that due to poor design in padding specification and MAC order, a man-in-the-middle can get the server to decrypt an arbitrary block of data, and the server will accept the message if and only if the last byte is correct. This gives an attacker 256 guesses (on average) before recovering a byte from the message.

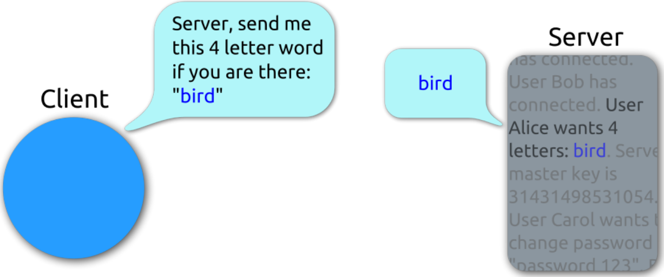
Heartbleed (2014)

Unlike CRIME, BEAST, POODLE, etc, Heartbleed was not a vulnerability in the protocol, but in a specific implementation of the protocol, the OpenSSL library. Unfortunately, this library was *very* widely used.

- The Heartbeat Extension (RFC 6520) of TLS allows a client to send a short message to the server, confirming the server's presence.
- This is meant to increase the reliability of TLS over long-lived connections.
- The heartbeat message is of the form (message length, message).
- OpenSSL assumed that the given message length was the true message length.

Heartbeat is a classic bug occurring by a lack of proper bounds checking.

Heartbeat – Normal usage



Client

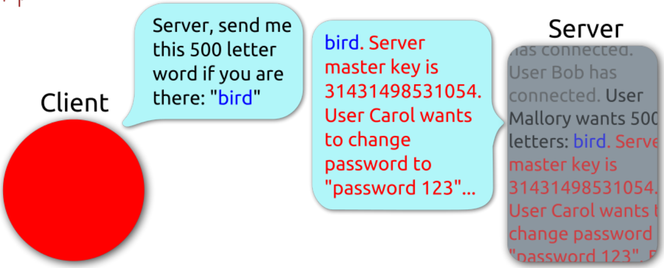
Server, send me this 4 letter word if you are there: "bird"

bird

Server

has connected. User Bob has connected. User Alice wants 4 letters: bird. Serve master key is 31431498531054. User Carol wants t change password "password 123". F

Heartbeat – Malicious usage



Client

Server, send me this 500 letter word if you are there: "bird"

bird. Server master key is 31431498531054. User Carol wants to change password to "password 123"...

Server

has connected. User Bob has connected. User Mallory wants 500 letters: bird. Serve master key is 31431498531054. User Carol wants t change password "password 123". F