Securing TCP Connections: SSL

Joe MacInnes
Department of Computer Science, College of Wooster
Wooster, Ohio 44691

***Abstract*—This paper explores the Secure Socket Layer protocol and how it is used to secure TCP connections. It provides a description of the underlying process of SSL and gives two examples of real world attacks on SSL: the heartbleed bug and the POODLE attack.**

I. Introduction

In the early days of the internet, protocols were designed to be robust, with little thought given to security. At the time, only a small number of researchers were using the internet, none of whom had malicious intent. The underlying protocols used in the internet, like TCP and UDP, date back to this time, and, as such, have no security mechanisms other than basic integrity checks (e.g checksums). As a result, security has largely been delegated to the application layer.

Secure Sockets Layer (SSL) is a direct response for this need to secure transport layer communications. It is a protocol built on top of the transport layer used to secure TCP connections, perhaps one of the most commonly used transport layer protocols. An application seeking to employ SSL uses an SSL socket, which then connects to a TCP socket (thus, SSL is somewhere between the application and transport layers). When used, SSL provides a TCP connection with three important components of security: confidentiality, integrity, and authentication.

II. The Process

In order to secure a TCP connection, SSL goes through three stages. The following descriptions of these stages assume Alice is attempting to legitimately communicate with Bob over a TCP connection, while a malicious third party, Eve lurks in the background. In addition, these descriptions are summary of the in-depth discussion on SSL provided by Kurose and Ross in their book, *Computer Networking: A Top-Down Approach* [1].

A. Handshaking and Key Derivation

Just like TCP, SSL requires a handshaking phase in order for Alice and Bob to establish a connection. In TCP, this is mostly just to allocate the necessary resources and buffers in the communicating parties, but SSL requires a few more things.

First, Alice and Bob establish a regular TCP connection (see Figure 1).

Next, Alice sends an SSL hello message to Bob. This message contains a list of cryptographic protocols that Alice supports as well as a nonce (an arbitrary, random number). Bob proceeds to choose algorithms from the list until he has symmetric key, public key, and message authentication code (MAC) algorithms. He then sends these choices back to Alice, along with a nonce of his own and his certificate.

Alice can use Bob's certificate to authenticate his identity. Certificates contain their owner's public key, but are signed by a verified signing authority. Thus, a client can get this signing authority's public key (which most browsers have built-in) and check the signature to see if it's valid.[2]

After doing this, Alice creates a fresh secret key, encrypts it with Bob's public key, and sends it back. Using this shared secret key and their respective nonces, Alice and Bob simultaneously generate two encryption keys and two MAC keys (one of each for data sent from Bob to Alice and vice-versa).

Lastly, both Alice and Bob send a MAC of all their handshake messages. Essentially, a MAC algorithm hashes some data, and the resulting hash is the MAC. Thus, if the data is tampered with along its journey from sender to receiver, then the receiver will generate a MAC from the data that doesn't match the MAC they receive from the sender. This is important for the SSL handshake phase, since Eve could tamper with the algorithm list sent from Alice

to Bob and remove stronger key algorithms - recall that this list is sent in cleartext. Figure 1 illustrates this activity.

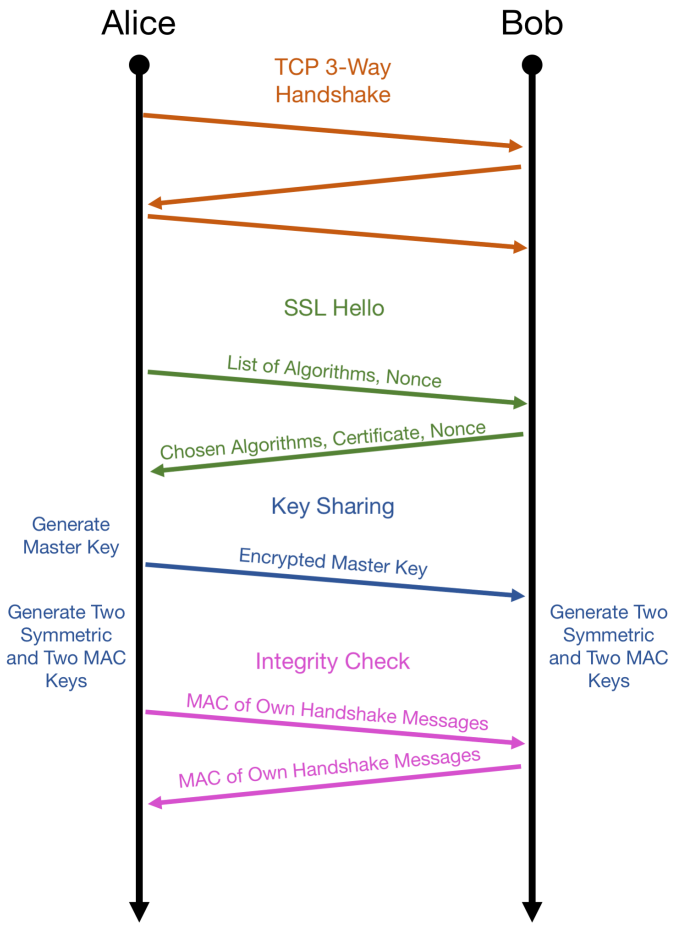

Fig. 1. The process of SSL handshaking and key derivation between Alice and Bob. Inspired by a figure in *Computer Networking: A Top-Down Approach* [1].

B. Data Transfer

With the proper keys established, Alice and Bob can begin sending data back and forth over their TCP connection. In order to do this securely, they break their respective data streams up into records. For each to send a record, they append their respective MAC key to the message data, and hash the concatenation (see Figure 3). They then encrypt this resulting hash using their respective session keys. The resulting cyphertext, along with a few options in cleartext comprise the record. This record is then sent to the other party in TCP segments as seen in Figure 2.

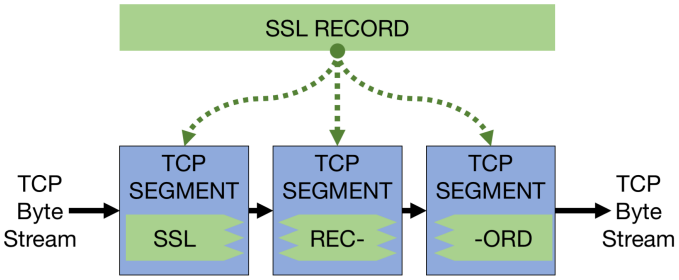

Fig. 2. An SSL record is broken apart and sent in a series of TCP segments. Inspired by a figure in *Computer Networking: A Top-Down Approach* [1].

When Alice receives a record from Bob, she first extracts the cyphertext and decrypts it using Bob's session key. Then, she takes message data from the cleartext, appends Bob's MAC key, and hashes it. If the result is the same as the received MAC, then the message hasn't been tampered with.

Under this setup, however, Eve can still destroy the integrity of the data exchanged between Alice and Bob. Since the records are sent in TCP segments, she could take the segments, reorder them, and change their sequence numbers so that they appeared to be in order. The receiver would not be aware of the reordering.

In order to prevent this attack, Alice and Bob each keep track of the sequence numbers of the data streams between one another. Every time they generate a MAC for a record, they include this sequence number counter in the data being hashed (but don't put it in the actual data sent). Thus, when the other party receives the record, they also include what they think the sequence number is when checking the MAC.

C. Connection Closure

To close the SSL connection, either Bob or Alice has to notify the other. Normally, this would be done by sending a TCP segment with the FIN flag set to 1. However, under this setup, once again Eve could meddle with the connection. All she has to do is, as a man-in-the-middle, send an early TCP FIN segment to Alice or Bob. This is known as a truncation attack, since Eve is truncating the session.

The solution is in the cleartext information included in an SSL record alongside the encrypted data and MAC. This information contains the version of SSL used, the length of the record (which lets the receiver know when it should parse a record), and finally the type. If this type is set to a termination value, then the receiver will know it should terminate the connection. While it seems odd to put these values in cleartext, these fields are incorporated when calculating the MAC (see Figure 3 for a summary of what all is hashed to produce the MAC), so they can't be tampered with without the receiver tossing out the record.

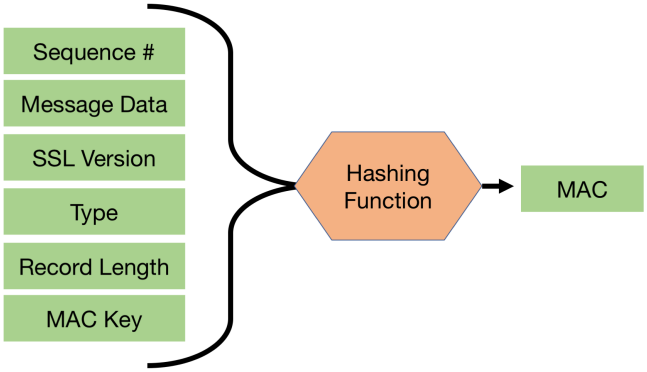

Fig. 3. All the elements that are fed into a MAC hashing function to generate the record's MAC. Inspired by a figure in *Computer Networking: A Top-Down Approach* [1].

III. SSL Vulnerabilities

Having discussed the methods SSL uses to secure a TCP connection, the following sections consider some major attacks on the protocol in recent years.

A. Heartbleed

In 2014, a bug was discovered in OpenSSL, an open source implementation of SSL, that allowed malicious users to steal information from servers. A relatively minor part of the SSL protocol not discussed above is the *heartbeat*. Communicating hosts will occasionally send tiny bits of encrypted data to one another, in order to let the other know that the connection is still open. When a host receives some heartbeat data, it stores the data in a buffer, and sends the data back to the sender.

Critically, the host sending the heartbeat data must send the data as well as how many bytes long the data is. The receiving host will allocate a buffer of that size and dump in the data, before sending it back. The OpenSSL implementation, however, never checked to see if the amount of data received matched the number of bytes the sender said it was.

Thus, the sender could send a small amount of data, but say it was a massive number of bytes. The receiver would allocate an equivalently large buffer and store the data, which would only take up a small fraction of the buffer. Because data in computer memory is not thrown away until it is overwritten, the rest of this buffer could contain valuable pieces of information like usernames and passwords. When the receiver tried to return the heartbeat data, it would copy the entire buffer and send it, giving the original sender this sensitive information (see Figure 4).

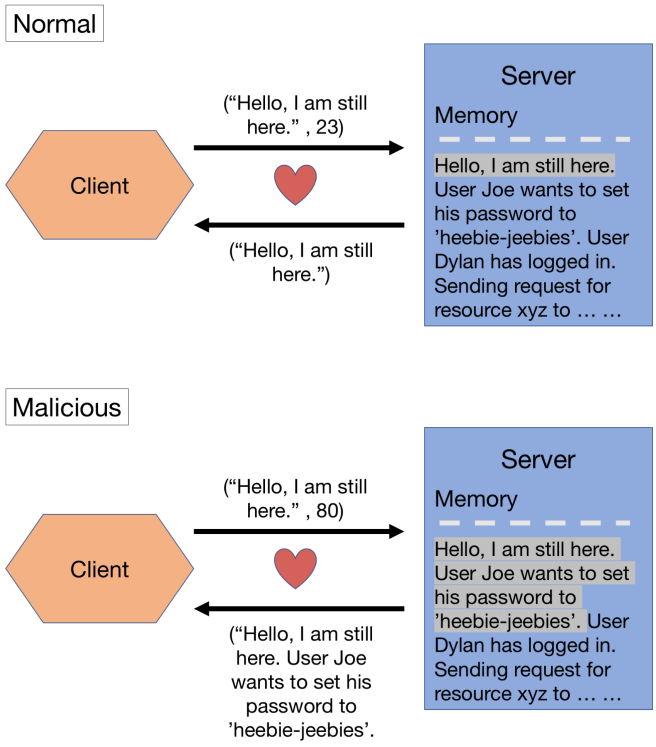

Fig. 4. An example of a normal and malicious heartbleed message. Inspired by a web comic [3].

This attack shows that, despite the solidity of the cryptographic primitives underlying SSL, implementation is equally as important. If the OpenSSL implementation had simply checked to see if the data in a heartbeat message matched the number of bytes the sender said it was, the vulnerability would not have existed. As it were, at the time of

its discovery, around 17% of all servers were using this OpenSSL implementation and were open to the attack [4].

B. POODLE

The internet consists of many hosts running various versions of SSL. As such, another minor part of SSL omitted from the description above is the way in which a client and server establish the version of SSL they will use. Basically, the server selects the best version of SSL it supports and tries to initiate a connection with the client. If this connection fails, the server will select the next best version of SSL it can use and try again.

A malicious entity engaging in a Padded Oracle On Downgraded Legacy Encryption (POODLE) attack will act as a man-in-the-middle attacker and cause these connections to fail until the server attempts to use SSL 3.0, at which point the attacker will allow the connection.

The encryption algorithms used in SSL 3.0 were found to be insecure. Given access to the cyphertext of enough messages, an attacker can successfully decrypt the information. Since the attacker is a man-in-the-middle, they do have access to the cyphertext of each message passed between the client and server, and so can retrieve encrypted information.

In response to this attack, most servers have begun to stop supporting SSL 3.0 [5].

IV. Conclusion

This paper has described SSL and shown two examples of real life exploits of the protocol that further reveal the protocol's intricacies. It should be important to note, however, that while SSL was the topic of the paper, almost all hosts now use the Transport Layer Security (TLS) protocol for securing transport layer connections. The switch was made in the early 2000s for a variety of reasons. Chief among these reasons was the fact that SSL was originally a proprietary protocol owned by Netscape. This made it very difficult to open up development of SSL. Overall, however, TLS is very similar to SSL. When referring to securing the transport layer, most texts now use the wording SSL/TLS [4].

References

[1] J. Kurose and K. Ross, *Computer Networking: A Top-down Approach*. Pearson, 2017, ISBN: 9780133594140. [Online]. Available: https : / / books . google . com / books ? id = VSAtjgEACAAJ.

[2] K. Vaniea, *Lecture notes on ssl*, University of Edinburgh, Nov. 2017.

[3] R. Munroe, *Heartbleed explanation*, XKDC Comics. [Online]. Available: https://xkcd.com/1354/.

[4] J. Fruhlinger, "What is the heartbleed bug, how does it work and how was it fixed?" *CSO IDG*, Sep. 2017. [Online]. Available: https://www.csoonline.com/article/3223203/vulnerabilities/what-is-the-heartbleed-bug-how-does-it-work-and-how-was-it-fixed.html.

[5] B. Moller, T. Duong, and K. Kotowicz, "This POODLE bites: Exploiting the SSL 3.0 fallback," *Security Advisory*, 2014.