An Inquiry of Modern Issues in Transport Layer Security

Michael J. Keen

Ball State University

Fall 2020

Introduction

Transport Layer Security (TLS) is a protocol that is responsible for securing end-to-end communications between clients and servers over computer networks, the most common of the latter being the Internet. TLS was preceded by another protocol known as Secure Sockets Layer (SSL), the first version of which was introduced in 1994[1]. However, subsequent security vulnerabilities led to the development of TLS as a replacement in 1999; as of 2020, all versions of SSL have been deprecated and are therefore no longer in widespread use[1]. However, TLS was originally designated as a version of SSL, so the terms SSL and TLS are often used reciprocally[6]. TLS is usually implemented over the Transmission Control Protocol (TCP) and Internet Protocol (IP) layer, therefore providing reliable communications between a server and client while also ensuring the confidentiality and data integrity of the information sent between them[6]. Therefore, TLS has played a crucial role in the development of the modern Internet, particularly because the exchange of potentially sensitive information (e.g., passwords, financial information) can take place without the fear of said information being stolen by a man-in-the-middle (MITM) attack. However, TLS is not impenetrable; vulnerabilities and problems, which can be present in the protocols, the algorithms, or even simply the implementation, have long hindered the ability of TLS to keep private information completely secure. The discussion of several of these modern obstacles will be the primary focus of this paper.

Overview

TLS is composed of six distinct protocols, which are shown in Figure 1 on top of the TCP/IP layer; overall, these protocols occupy two separate layers of their own. The record protocol's primary function is to provide base security services for the protocols situated above it. As shown, the protocols above the record protocol are the handshake, the change cipher spec,

HTTP, and the heartbeat. However, due to their relevance for topics discussed later, this paper will only cover the handshake, heartbeat, and alert protocols in detail. The problems discussed will mainly revolve around vulnerabilities that can be exploited by attackers, as well as the relative security of current TLS protocols.

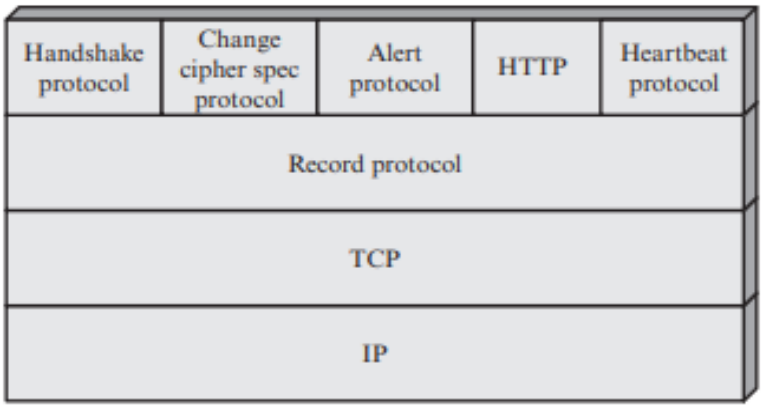

Figure 1: Five application-layer protocols of TLS, situated above the Record Protocol and the TCP/IP protocols

Statement of the Problem

This paper will talk about four different problems concerning the current state of TLS. The Concepts and Terminology section will provide clarification about the protocols and mechanisms discussed here, the latter of which will be essential to understand the full scope of the issues at hand. The problems discussed will be as follows:

1. **Attacks on handshake protocol**[1]: These attacks that could lead to discovery of the secret key during the handshake protocol.
2. **Heartbleed**[2,5]: The flaw in the OpenSSL library's implementation of the TLS heartbeat protocol.

3. **Attacks against authentication**[4]: These attacks target TLS renegotiation, wireless networks, as well as RSA to jeopardize the ability of TLS to authenticate clients and servers.
4. **TLS renegotiation security**[3]: The concept of renegotiation in TLS comes with inherent security risks.

Concepts and Terminology

Of all the protocols that make up TLS, the handshake protocol is the most complicated and the most critical. It is responsible for establishing connections between a client and server and for generating a secret symmetric key that will be used to encrypt information sent between them. This process must be completed before any information can be transferred to prevent the interception of sensitive data by an attacker. The handshake protocol consists of four separate phases, which are shown in detail in Figure 2.

The first phase is responsible for initiating the connection. A TLS handshake begins when a client attempts to connect to a specific but arbitrary server by sending what is known as a client_hello message. This message contains information about the highest version of TLS that the client supports, a random structure with an embedded timestamp to guard against replay attacks, and a session ID parameter that signals whether the client desires to establish a new session or update an existing one; a value of zero denotes a new session, and any other numeric value denotes an updated session[6]. It also contains a list of the cipher suites (with the first one being the most preferred) and compression methods that the client supports. Once a server receives the client_hello message, it responds to the client with another message known as the server_hello message. This message is structured much of the same way as the client_hello message; the server sends the selected version of TLS (the highest they both support), another

random sequence independent of the client's, and the new session ID (if the session is not being updated)[1]. It also sends the selection of a cipher suite and a compression method from the lists sent by the client. Information in the cipher suite parameter includes the algorithm used for key exchange (such as RSA or Diffie-Hellman), the encryption algorithm (such as DES or RC4), the message authentication code (MAC) algorithm, as well as whether the encryption algorithm is a stream or block cipher. The reception of the server_hello message by the client thus concludes the first phase[1].

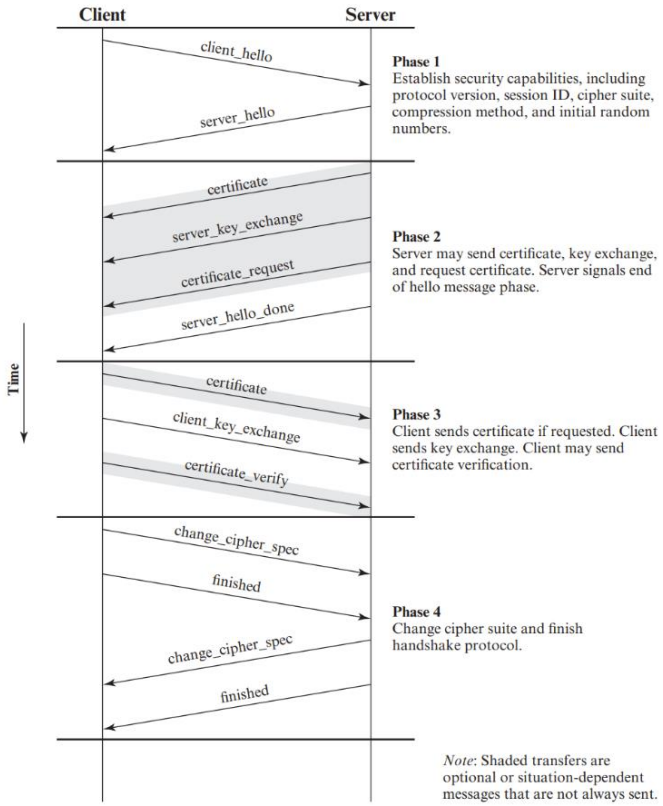

Figure 2: The handshake protocol and the messages sent in each phase

At the start of the second phase, the server has several messages that may be sent, though as shown above, they are not sent in all situations. One of these discretionary messages is the server's certificate, which can be used by the client to verify the server's identity. This is also usually necessary for the key exchange to occur. After the server sends its certificate, it can also send a server_key_exchange message. This is required when the client and server use RSA or

certain types of Diffie-Hellman as the algorithm for the key exchange[1]. Afterward, the server can also request the client to verify itself by sending a certificate_request message. This message contains the public key's algorithm as well as the purpose of said public key (e.g., signature, authentication, etc.). The request also contains a list of certificate authorities that are recognized by the server[6]. If the client's certificate authority is not in this list, then the server will be unable to independently verify the client. The final message of the second phase is always required by the handshake protocol. The server sends a server_hello_done message, which notifies the client that the server has finished the hello exchange and is ready to begin exchanging keys. The server will not proceed until the client responds.

During the third phase, the server and client collaborate to exchange a shared secret key that will be used to encrypt the information sent to and from each other. Additionally, if the server had requested that the client verify itself with its certificate at the end of the previous phase, then the client will send its certificate. If the client has no certificate, then it will reply to the server with a message saying so[1]. Once the client receives the server_hello_done message and finds the server's certificate to be acceptable, the client proceeds with key generation. If the chosen algorithm is RSA, then the client generates an RSA key pair that is called the pre-master secret, as this key will be used to generate, but will not be used as, the final secret key[1]. If the algorithm is a type of Diffie-Hellman, then the client will instead use their respective Diffie-Hellman key. The client then takes this information and sends it in a message known as the client_key_exchange[1]. After this message, the client can also further authenticate itself to the server if by sending a certificate_verify message. This only occurs if the server requested the client's certificate and the client's certificate uses an algorithm that supports digital signatures, such as RSA. This marks the end of the third phase.

The fourth phase of the TLS handshake is relatively short and concludes the protocol. The client starts this phase by using the Change Cipher Spec Protocol to send a change_cipher_spec message. This message informs the server that data immediately following will be encrypted using the secret key that was exchanged[6]. The client then sends a "finished" message, indicating it is ready to exchange data. Upon receiving these messages, the server will send both its own change_cipher_spec message and "finished" message as well[1]. Upon reception of these messages by the client, the handshake is complete; any information exchanged afterwards will be protected by the shared secret key[6].

In comparison to the handshake protocol, the heartbeat protocol is relatively simple. It is responsible for generating a signal known as a "heartbeat"; this signal is sent infrequently for the purpose of ensuring that the receiver is still connected even if there is no active traffic between the sender and the receiver. It can also be used to create traffic during a state of inactivity, lest the client-server connection between appear suspicious to a firewall[1]. Another version of TLS, known as Datagram Transport Layer Security (DTLS), also uses the payload in the heartbeat protocol for the purpose of guarding against packet loss, as this version of TLS uses UDP packets instead of TCP packets, making the connection more unreliable[6]. The heartbeat protocol operates by using a message type known as a heartbeat_request message. Since the heartbeat protocol is bidirectional, this message can be sent by either the client or server. The heartbeat_request message includes parameters that specify a payload, the payload size, and padding. The payload and the padding are a random assortment of bytes, and the payload size is always between 16 bytes and 64 kilobytes. Upon reception of the heartbeat_request message by the recipient, the latter immediately sends another message back, which is known as a heartbeat_response. This message includes an exact copy of the payload that was included in the

heartbeat_request[1]. This indicates to the sender that the recipient is functioning properly, and that the connection should continue.

The alert protocol is used to communicate alerts to either the client or the server, which can occur during the connection negotiation as well as during the connection maintenance. Messages sent by the alert protocol have a predictable structure; they always have a total of two parameters, with each parameter occupying a single byte[1]. The first parameter conveys the gravity of the alert and only accepts two values: “warning” or “fatal”. In the case of “warning”, the connection can usually proceed, but a value of “fatal” indicates that the connection will be aborted[1]. The second byte provides further details about the alert, as it holds a value that corresponds to a specific message, some of which are shown in Figures 3 and 4. Certain messages always have a fixed corresponding alert level. For example, handshake_failure and bad_record_mac, are always marked as “fatal,” while some, such as no_renegotation, are always marked as “warning.”

- **unexpected_message:** An inappropriate message was received.
- **bad_record_mac:** An incorrect MAC was received.
- **decompression_failure:** The decompression function received improper input (e.g., unable to decompress or decompress to greater than maximum allowable length).
- **handshake_failure:** Sender was unable to negotiate an acceptable set of security parameters given the options available.
- **illegal_parameter:** A field in a handshake message was out of range or inconsistent with other fields.

Figure 3: Fatal messages sent by the alert protocol

- **close_notify:** Notifies the recipient that the sender will not send any more messages on this connection. Each party is required to send a close_notify alert before closing the write side of a connection.
- **bad_certificate:** A received certificate was corrupt (e.g., contained a signature that did not verify).
- **unsupported_certificate:** The type of the received certificate is not supported.
- **certificate_revoked:** A certificate has been revoked by its signer.
- **certificate_expired:** A certificate has expired.
- **certificate_unknown:** Some other unspecified issue arose in processing the certificate, rendering it unacceptable.
- **decrypt_error:** A handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a finished message.
- **user_canceled:** This handshake is being canceled for some reason unrelated to a protocol failure.
- **no_renegotiation:** Sent by a client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these messages would normally result in renegotiation, but this alert indicates that the sender is not able to renegotiate. This message is always a warning.

Figure 4: Other messages sent by the alert protocol

Project Goals

The goals of this project are to provide an in-depth explanation of the four problems mentioned in the Statement of the Problem section, along with an accompanying description of the solution. It will also detail any unresolved problems that might still exist within the current context. The project will achieve these goals by educating the reader on the alert, handshake, and heartbeat protocols before the discussion of the problems at hand.

Literature Review and Current Status

According to William Stallings, vulnerabilities for the TLS handshake protocol have been known at least since the year 1998[1]. One method for exploiting a particular vulnerability involved manipulation of the RSA implementation within TLS to compromise the shared secret key, allowing attackers to decrypt all communications between the client and server. Upon discovery of this vulnerability, patches that were designed to make this attack infeasible were

issued. However, this led attackers to not only develop other methods intended to bypass the patches, but also to ensure that each new attack executed much faster than the original [1].

In April 2014, it was found that a library responsible for the implementation of TLS protocols suffered from a high severity vulnerability that came to be known as Heartbleed. Characterized by some as "the worst vulnerability since e-commerce began on the Internet" [5], Heartbleed was found in the implementation of the heartbeat protocol provided by the OpenSSL library. It was a severe memory problem that, when exploited, could allow attackers to access sensitive data on the server, including names, passwords, and the server's private key. According to Shashank Kyatam, the vulnerability stemmed from a section of code that checked for the reception of heartbeat_request messages[2]. As stated in the Concepts and Terminology section, each heartbeat_request message contains a parameter for a payload as well as the size of that payload. Normally, the heartbeat protocol would check to ensure that the size of the payload corresponds to the value provided in the payload size parameter. However, this check was not present in the implementation by OpenSSL. This made it possible that the heartbeat protocol could accept a heartbeat_request message in which the payload size was smaller than stated in the parameter[2]. As a result, when the server would attempt to retrieve the content of the payload from memory for the heartbeat_response message, the payload would not be large enough to occupy the size denoted by the parameter. This would cause the server to deposit the contents of anything that happened to come next in memory inside of the heartbeat_response payload, which would be server data likely not related to the heartbeat protocol. Though an attacker might not initially receive any valuable information, repeated heartbeat_request messages would increase the likelihood that they would eventually retrieve sensitive data[2]. Although Heartbleed has since been patched, it was determined to have been present in OpenSSL for at least two years prior.[5]

The amount of time the vulnerability was exposed lends credence to the possibility that it could have been discovered and/or exploited by an attacker before it was made public.

The main objective of TLS is to provide cryptographic functionality to software engineers who may not have any knowledge about protecting their web applications and servers from attacks. However, Karthikeyan Bhargavan, et. al. state that when these engineers rely too ignorantly on TLS for general authentication and protection, it can leave the server dangerously vulnerable[4]. This is can be attributed to client impersonation attacks, which can target "TLS renegotiations, wireless networks, challenge-response protocols, and channel-bound cookies.[4]" These attacks are also capable of manipulating the key exchange process, whether using RSA or Diffie-Hellman, as well as weaknesses that are present in TLS operating over HTTP. Left untreated, the authors assert that these vulnerabilities could be exploited to attack prominent web applications and, via client impersonation, possibly compromise sensitive personal data belonging users of said web applications[4].

TLS renegotiation occurs when a client-server pair wishes to continue interacting after a session key has expired, when one of them wishes to adjust the cipher suite, or when the server requests a certificate from the client after the conclusion of the handshake protocol. Though this feature of TLS can be useful, it is not without its downsides. Florian Giesen, Florian Kohlar, and Douglas Stebila discuss how an attacker could use TLS renegotiation to "splice together its own session with that of a victim, resulting in a man-in-the-middle attack on TLS-reliant applications such as HTTP[3]." It is because of vulnerabilities such as this one that some servers do not offer the option of renegotiation, and thus send out the no_renegotiation message that is mentioned in the Concepts and Terminology section. However, the authors proceed to state that they have a solution that will enhance the security TLS renegotiation and help to guard against future attacks.

Solutions

Though these issues may have serious implications for the security of TLS, many of them have solutions that are addressed by the authors. In the case of attacks on the handshake protocol, Stallings states that continued revision of the implementation of RSA within TLS is the best strategy for the prevention of future attacks. He notes that the introduction of TLS v1.3 in 2018 completely discards RSA as a key exchange algorithm and relies solely on Diffie-Hellman or Elliptic Curve Cryptography[1]. This all but ensures the elimination of security issues with RSA, including the implementation vulnerabilities, once v1.3 becomes the standard.

In the case of the Heartbleed vulnerability, the patch for servers and clients using the OpenSSL library was issued shortly after discovery[2]. Even so, this patch only offers protection if the servers and clients apply the update. Failure to do so will allow the vulnerability to remain and the contents of the server to continue to be exploited[2]. Additionally, given the possibility that the vulnerability was discovered by an attacker before it was made public, users of websites whose servers use OpenSSL should consider their credentials, as well as other personal information, compromised. Therefore, these users should ensure that have changed all sensitive data that they are able to after the vulnerability was patched, and if not, consider it as soon as possible.

To prevent vulnerabilities with TLS regarding client impersonation, it is recommended that modifications are made to the process for key exchange. Specifically, it is proposed that the process creates “a new TLS channel binding, called tls-session-hash, that captures all the negotiated parameters for a session[4].” This would allow for better handling of the secret key and thus keep it safe from possibly being intercepted by attackers. It would do this through additional

verification of the client, which would also verify the client's identity and keep personal information safe.

TLS renegotiation can present security risks, but there are steps that can be taken to ensure protection against potential attacks, with many of the remedies being relatively simple[3]. This can include creating a hash of the messages from the record protocol, allowing both the client and server to verify that are receiving each other's messages. Countermeasures such as this can be effective even when dealing with powerful attackers, such as those sponsored by foreign states[3].

Open Problems

Though the Solutions sections addresses specific aspects of security within TLS, there are still open issues within the overall protocol that do not yet have a definitive resolution. Specifically, Bhargavan, Lavaud, Fournet, Pironti, and Strub note that while their solution may protect against attacks that use client impersonation, they recommend that the TLS protocol undergo a revision that will strengthen authentication methods, such as by addressing the issues within "channel bindings in SASL and compound authentication in PEAP[2]." They mentioned that they are collaborating with members of the TLS authors to revise these authentication issues, and that, at the time of writing, discussions were still ongoing regarding the best approach.

As stated in the Solutions section, Stallings notes that though patches have deployed for specific instances of vulnerabilities within the TLS implementation of RSA, there are likely to be more vulnerabilities discovered in the future[1]. And as attackers uncover these novel vulnerabilities that might possibly prove to be more devastating, the implementation will simply have to be revised further. Stallings notes that a "'perfect' protocol and a 'perfect'

implementation strategy are never achieved. A constant back and forth between threats and countermeasures determines the evolution of Internet-based protocols[1].”

Though Giesen, Kohlar, and Stebila are confident in their solution on the topic of renegotiation for TLS, they mention that TLS still has “important open questions, including the security of other TLS ciphersuites and the formal analysis of other complex functionality such as alerts and error messages[3].” Therefore, they recommend research that targets the “security of TLS in increasingly realistic scenarios,” as opposed to theoretical scenarios[3].

Conclusion and Summary

Transport Layer Security, as well as its predecessor Secure Sockets Layer, are protocols that are used to provide security across distributed system communications, with the most common of those being the Internet. To achieve this, TLS adapts a modular design and is itself composed of several different protocols, such as the handshake and the heartbeat, which provide specific functions for the secure exchange of information. TLS is an integral component of the modern Internet and e-commerce economy that has been growing for much of the 21st century. The ability of TLS to provide end-to-end encryption between clients and servers has provided both consumers and companies with a sense of contentment about the privacy of their information. However, despite the relative success of TLS, it still faces challenges that include those relating to authentication, implementation of key exchange protocols, as well as cipher suites and alerts. Therefore, continued revision of the protocol, as well as the protocols that it consists of, are essential for maintaining its performance and effectiveness.

References

1. W. Stallings., “17.2 Transport Layer Security.” Cryptography and Network Security, Pearson Education, 2017, pp. 531–548.
2. S. Kyatam, A. Alhayajneh and T. Hayajneh, "Heartbleed attacks implementation and vulnerability," 2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT), Farmingdale, NY, 2017, pp. 1-6, doi: 10.1109/LISAT.2017.8001980.
3. F. Giesen, F. Kohlar, and D. Stebila. “On the Security of TLS Renegotiation”, ACM, 2013, doi:10.1145/2508859.2516694.
4. K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti and P. Y. Strub, "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS," 2014 IEEE Symposium on Security and Privacy, San Jose, CA, 2014, pp. 98-113, doi: 10.1109/SP.2014.14.
5. M. Carvalho, J. DeMott, R. Ford and D. A. Wheeler, "Heartbleed 101," IEEE Security & Privacy, vol. 12, no. 4, pp. 63-67, July-Aug. 2014, doi: 10.1109/MSP.2014.66.
6. S. Turner, "Transport Layer Security.," IEEE Internet Computing, vol. 18, no. 6, pp. 60-63, Nov.-Dec. 2014, doi: 10.1109/MIC.2014.126.