ORIENTAL JOURNAL OF
COMPUTER SCIENCE & TECHNOLOGY
An International Open Free Access, Peer Reviewed Research Journal
Published By: **Oriental Scientific Publishing Co., India.**
www.computerscijournal.org

ISSN: 0974-6471
December 2012,
Vol. 5, No. (2):
Pgs. 315-320

SSL Based Transport Layer Security

J.VIJAYABHARATHI

Department of Computer Science, Mother Teresa Women's University
Kodaikanal. (India)

(Received: July 1, 2012; Accepted: June 04, 2011)

ABSTRACT

The last couple of years has seen a growing momentum towards using the Internet for conducting business. One of the key enablers for business applications is the ability to setup secure channels across the internet. The Secure Sockets Layer (SSL) protocol provides this capability and it is the most widely used transport layer security protocol. In this paper we investigate the performance of SSL both from a latency as well as a throughput point of view. Since SSL is primarily used to secure web transactions, we use the SPECWeb96 benchmark suitably modified for use with the SSL protocol. We benchmark two of the more popular web servers that are in use today and find that they are a couple of orders of magnitude slower when it comes to serving secure web pages. We investigate the reason for this deficiency by instrument the SSL protocol stack with a detailed profiling of the protocol processing components. Based on our findings we suggest two modifications to the protocol that reduce the latency as well as increase the throughput at the server.

Key words: SSL, Transport, Security

INTRODUCTION

Security is important on the Internet. Whether sharing financial, business, or personal information, people want to know with whom they are communicating (authentication), they want to ensure that what is sent is what is received (integrity), and they want to prevent others from intercepting their communications (privacy). The Secure Sockets Layer (SSL) protocol provides one means for achieving these goals at the transport layer. It was designed and first implemented by Netscape Corporation as a security enhancement for their Web servers and browsers. Since then, almost all vendors and public domain software developers have integrated SSL in their security sensitive client-server applications. At present, SSL is widely deployed on the intranet as well as over the public Internet in the form of SSL-capable servers and clients and has become the de facto standard for transport layer security. Recently, the Internet Engineering Task Force (IETF) has started an effort to standardize SSL as an IETF standard under the name of Transport Layer Security (TLS) protocol. In the rest of this paper, we refer to SSL as the Transport Layer Security protocol, or simply TLS. One of the reasons that TLS has outgrown other transport and application layer security protocols such as SSH, SMIME, and SET in terms of deployment is that it is application protocol independent. Conceptually, any application that runs over TCP can also run over TLS. There are many examples of applications such as TELNET and FTP running transparently over TLS. However, TLS is most widely used as the secure transport layer below HTTP. A large number of Web sites dealing with private and sensitive information, including all those engaged in E-Commerce, use TLS as the secure transport layer. This number is expected to grow exponentially as more and more businesses and users embrace electronic commerce. As security becomes an integral feature of Internet applications and the use of TLS rises, its impact on the performance of the servers as well the clients is going to be increasingly important.

The objective of this paper is to take a close and critical look at the TLS protocol with an eye on performance. The TLS protocol is composed of two main components: (1) the TLS Record Protocol responsible for data transfer, and (2) the TLS Handshake Protocol responsible for establishing TLS session states between communicating peers. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP [9]), is the TLS Record Protocol. The Record Protocol provides two basic security services: privacy and message integrity. It uses data encryption using symmetric cryptography (e.g., DES [2], RC4 [16], etc.) to provide privacy and a keyed message authentication digest to ensure message integrity. The TLS Record Protocol is used for encapsulation of various higher level protocol messages, including the TLS handshake protocol. The TLS Handshake Protocol is responsible for authenticating the server and the client to each other. It is also entrusted with the job of negotiating an encryption algorithm along with the required session keys before the application protocol transmits or receives its first byte of data. The Handshake protocol typically uses public key cryptography for exchanging secret information. TLS allows the session state to be cached for configurable amount of time. If a client needs to setup a new TLS session while its session state is cached at the server, it can skip the steps involving authentication and secret negotiation and reuse the cached session state to generate a set of keys for the new session. In the rest of the paper, we analyze the performance impact of TLS on HTTP and quantify the overhead associated with different components of TLS. To measure the performance impact of TLS on Web server performance, we have modified the SPECweb96 benchmark to generate client workload for servers and clients running HTTP transactions over TLS. Using this modified SPECweb96 benchmark, we evaluated the performance of two different secure Web servers with varying degrees of session reuse. Our results show that depending on the degree of session reuse the overhead due to TLS can decrease the rate at which the server can process HTTP transactions by up to two orders of magnitude. To identify the overhead associated with different components of TLS, we have instrumented and traced the TLS Handshake Protocol and TLS Record Protocol using timers with sub-micro second granularity. Our results indicate that for a typical HTTP transaction (10-15 Kbytes), the bulk of the overhead comes from the TLS Handshake Protocol unless the session state is reused. For very large HTTP transactions (1 Mbytes or more), the overhead due to data encryption and authentication is significant. We also observed that TLS handshake protocol adds significant latency to Web transfers due its four-way handshake. In light of these observations, we propose two techniques to improve the performance of the TLS Handshake protocol, namely caching of server certificates by clients and a three-way handshake protocol. As discussed later in the paper, by caching server certificates at the client, it is possible to reduce the number of messages exchanged during TLS handshake and consequently a round trip time. Certificate caching also reduces computational overhead at the client and the volume of data transferred during handshake. The second scheme is designed to offload the computationally expensive private key operations from the server.

Transport Layer Security

TLS provides the ability to setup private communication channels in a public network. Broadly, the operation of TLS can be split up along two major axes. One is the cryptographic techniques that it uses to provide security and the other is the operation of the protocol itself. First we review a few basic cryptographic operations that are critical to TLS and then describe the protocol.

A. Basic Mechanisms

TLS uses symmetric key encryption techniques, such as DES and RC4, to ensure privacy. In symmetric key encryption the sender and the receiver share a secret key which is used to encrypt or decrypt messages. However, this secret key must somehow be exchanged between the communicating parties before any secure communication can take place. During the TLS handshake process the client chooses a secret which it then sends to the server. Public key cryptography is used to protect this exchange. Unlike symmetric key cryptography, public key cryptography uses a pair of keys, a public key and a private key. As the name suggests, the owner of the key pair publishes the public component of the key and keeps the private component secret. If the public key is used to encrypt a message then only the private key can be used to decrypt it and vice versa. In TLS, the initiator of a session, typically the client, generates the secret and encrypts it with the public key of the peer, typically the server. The server, upon receipt of this message, uses its private key to decrypt it. Since the server is the only one who possesses the private key, from this point on, the client and the server share a secret which no one else knows. One of the main reasons why public key cryptography is used only to communicate the shared secret is the fact that it is computationally rather expensive and so in reality it can only be used to encrypt a few bytes of data. So the key exchange problem is solved, provided the client knows the server's public key. This can be supplied by the server, but the client has to be able to bind the public key with the true identity of the server. TLS makes use of X.509 certificates to associate a public key with the real identity of an individual, server, or other entity, known as the subject. A certificate is signed by a trusted agency, commonly referred to as a certificate authority (CA). The signature process again typically involves public key cryptography. The signing entity computes a hash function of the data to be signed and encrypts that with its private key. The signature can be verified by performing the corresponding decryption with the public component of the private key and then matching the result with the freshly computed hash of the data. Although encryption guarantees privacy, it does not ensure message integrity. An adversary may still alter the encrypted messages without the sender or receiver being aware of it. TLS ensures message integrity by sending a digest of the message to the receiver along with the original message. Digest algorithms, such as MD5and SHA, are one-way hash functions that output a unique digest for each input message. It is relatively easy to verify a digest given the original message. However reproducing the message given the digest is impossible. TLS guarantees message integrity by keying the message digests with a secret key shared between the sender and the receiver. Any modification to the message will result in a mismatch between the digests computed by the sender and the receiver, thus enabling the receiver to detect a compromised message.

Protocol Overview

Rather than defining a completely new transport layer protocol, TLS is layered on top of an existing reliable transport protocol *viz.* TCP/IP. This naturally introduces an in efficiency since the TLS negotiation cannot start until the TCP/IP handshake is completed. However, this clean separation between the transport and security operations, enabled a fairly rapid prototyping effort and partly contributed to the wide popularity of TLS. A TLS connection involves two stages. First, the communicating parties optionally authenticate each other and then exchange session keys. This phase is known as the TLS handshake. Once the handshake is complete, the two parties share a secret which can be used to construct a secure channel over which application data can be exchanged. TLS is an asymmetric protocol. It differentiates between a client and a server. The TLS handshake sequence may vary, depending on whether the RSA or Diffie-Hellman key exchange is used. Although TLS handshake allows both the client and the server to be authenticated to each other, most commonly, it is only the server that is authenticated. Client authentication is optional and is omitted in most cases. A typical TLS session makes use of the RSA key exchange with only the server being authenticated. We only consider this case in this paper. The client initiates the communication by sending a **Hello** message to the server. The Hello message includes a random number that is used in the handshake to prevent replay attacks. In response to the client Hello, the server replies with a Hello of its own, followed by a certificate that contains the server's public key. Optionally, it may also send a chain of certificates belonging to the authorities in the certification hierarchy. The client verifies the certificate (or chain of certificates) by verifying the identity of the server and checking the validity of the CA's signature. The client then generates a pre-master secret and encrypts it with the public key obtained from the server's certificate. This is sent to the server which does a decryption using its private key, thus obtaining the pre-master secret. The pre-master secret is used to generate a master secret that is now shared between the client and the server. The master secret is then used to generate symmetric keys for encryption and message authentication. In other words the master secret is a shared state between the client and server and this constitutes a TLS session. This session can identified by the session ID that was included in the initial server Hello message. In contrast to the initial handshake protocol, the reestablishment of a cached TLS session is relatively simple.

The server checks in its cache to determine if it has state associated with this session. If the session state still exists in the cache, it uses the stored master secret to create keys for the secure channel. The client repeats the same process and generates an identical set of keys. Note that multiple secure channels between the same pair of hosts can be established by reusing a single session state. This is a rather key feature of the TLS protocol that is particularly important in the context of the world wide web. A single secure web-page may be composed of multiple HTTP links and being able to reuse an existing session state to obtain the multiple links greatly reduces the latency and processing involved in setting up the secure channel.

Performance

Although TLS can be used with a variety of application protocols, such as TELNET and FTP, the most important and most common use of TLS has been to ensure privacy and authentication for HTTP transactions. All commercial web sites that require privacy and authentication use TLS. In this section, we benchmark the performance of secure Web servers and quantify the overheads of different components of TLS. We use the SPECweb96 benchmark as it attempts to capture real-world usage of a web-server and is based on the analysis of server logs from a few different Internet servers.

RESULTS AND DISCUSSION

The servers are configured with certificates for 1024-bit keys. In all of these experiments, we used RC4 for data encryption and MD5 for message authentication, since these are the most widely used in real life. Performance of other encryption and message authentication schemes are presented later in the section. We varied the degree of session reuse from 0-100%. When session reuse is 0% all TLS sessions setup between the server and the clients require a full handshake with the associated public and private key operations. When session reuse is 100%, only the first TLS session setup between the server and a client involves a full handshake. All subsequent connections reuse the already established session state between the server and the client. When the percentage of session reuse is in between 0 and 100, the clients reuse the same session for a certain number of times depending on the value of the reuse percentage. The way this is done is by maintaining a running counter of the number of connections that attempted to reuse session state. Whenever this counter drops below the desired fraction (reuse percentage) of total connections, the client attempts to reuse an existing session ID. For example, when the reuse percentage is set to 50, the sessions setup by a SPECweb client takes the form NRNR…, where N stands for a new session and R stands for a reused session. The Apache server can handle, at most, 15 requests per second when there is no session reuse. The Netscape server can only handle about 5 requests per second. At these operating points the latencies are extremely high in both cases with Apache coming in at around 300 msec and Netscape hovering above the 600 msec mark. We notice that as the amount of session reuse is increased the performance improves and with a 100% reuse the latency is fairly low even when the rate of connection requests is quite high. The numbers for 100% reuse are only provided as a reference since in all practicality a web-server is unlikely to experience such a large amount of session reuse. In comparison, the SPECweb96 numbers for Netscape and Apache for regular web-pages on the same server are around 300 and 250 requests a second, respectively.

The behavior of the Netscape server is fairly typical of what one would expect when the level of session reuse is varied. In Figure 2 we observe that the latency reduces and the sustainable throughput increases as the level of session reuse is increased. In contrast, with the Apache server at light loads there does not seem to be much difference in the latency results when the reuse is increased from 0 to 80%. This behavior may be a result of how session reuse is implemented in the Apache web server. Apache uses a process model in its web-server implementation. The web-server is composed of several, dynamically created server processes that serve web-requests. Rather than make a single entity responsible for dispatching the requests to each of the server processes, the creators of Apache chose to have each server process pick up a connection request and service it. This provides for some natural load-balancing features since a server process only picks up a request when it is free. When a TLS client wishes to reuse a session, it includes the session ID in the client Hello message. However at the time the connection is accepted by a server process, it has no knowledge of what the session ID will be since the Hello message is received only after the connection is accepted. Unfortunately, with most flavors of Unix, once a connection request is accepted there is no way to rescind it and so the server process is forced to serve the request, whether or not it has the session ID in its cache.

CONCLUSION

TLS is widely deployed on the intranet as well as over the public Internet in the form of TLS-capable servers and clients and has become the de facto standard for transport layer security. Although the security implications of TLS has been under the microscope ever since its inception, similar analysis of its performance has not been performed. In this paper, we have analyzed TLS from a performance perspective and quantified its impact on applications protocols and servers, specifically HTTP And Web servers. Using a modified SPECweb96 benchmark, we evaluated the performance of Web servers running HTTP transactions over TLS. Our results show that the overhead due to TLS can decrease the number of HTTP transactions handled by up to two orders of magnitude. Given the rather significant growth in the use of TLS particularly in the burgeoning field of E-commerce, it is not clear how secure web-servers of today will keep pace with this growth.

REFERENCES

1. C. Allen and T. Dierks. The TLS Protocol Version 1.0. Internet Draft, Internet Engineering Task Force, Work in progress (1997).
2. ANSI X3.106. American National Standard for Information Systems-Data Link Encryption. American National Standards Institute (1983).
3. T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0, (1995).
4. D. Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on The RSA Encryption Standard PKCS#1. In Advances in Cryptology-Crytpo '98 (1998).
5. D. Bleichenbacher, B. Kaliski, and J. Staddon. Recent Results on PKCS#1: RSA Encryption Standard. RSA Laboratories' Bulletin, 24 (1998).
6. W. Diffie and M. E. Hellman. New Directions in Cryptography. IEEE Transactions on Information Theory, IT-**22**(6):74–84 (1977).
7. A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 Protocol. Netscape Communications Corporation (1996).
8. V. International and M. International. Secure Electronic Transaction 1.0 specification (997). http://www.setco.org.
9. ISI for DARPA. Transport Control Protocol. RFC 793, (1981).
10. NIST FIPS PUB 180-1. Secure Hash Standard. National Institute of Standards and Technology, U.S. Department of Commerce, DRAFT (1994).
11. B. Ramsdell. S/MIME Version 3 Message Specification. Internet Draft, Internet

Engineering Task Force, May 1998. Work in progress.

12. R. Rivest. RFC 1321: The MD5 Message Digest Algorithm (1992).

13. R. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, **21**(2):120-126 (1978).