ClosedSSL, a Modified Implementation of RFC 2246

Shona Seema Hemmady, Amelia Grace Holcomb, Vicky Tu

Advisor: Professor Richard Yang

CPSC 433/533 Computer Networks

10 May 2016

Table of Contents

Introduction

Approach

Background Research
- OpenSSL
- RFC 5246

Features
- Handshake
  - *Helo*
  - *Cert*
  - *Key*
  - *Finished*
- Encryption/Decryption

Obstacles and Solutions
- Calling ssl_connect multiple times
- Calling ssl_accept multiple times
- sendPacket

Final Result

Acknowledgements

Introduction

Cryptographic communication has risen to unprecedented importance, paralleling the rise of the World Wide Web. Every client communicating to a server wishes to know that 1) the messages the two parties send are encrypted using a key only the two share, 2) the messages the client receives are coming from the server unaltered, and 3) the messages the client sends are reaching the server unaltered. A simple public-private key encryption scheme is insufficient to solve this problem, because the associated computation costs are too large for a transport-layer encryption. To that end, Secure Sockets Layer (SSL) and it's successor Transport Layer Security (TLS) were created. For this project, we endeavored to create closedSSL, a modified version of TLS that implements the main aspects of the original cryptographic protocol outlined in RFC 2246.

Approach

To make the project manageable, we partitioned the project into the following components:

- Background research: We researched OpenSSL and read RFC 2246 to understand the components of SSL and TLS.
- Diagramming handshake flow: After learning about the protocols, we created a diagram of the handshake flow, undoubtedly the most complicated part of closedSSL. We included the parts that we believed were critical to the protocol.
- Functions: We implemented the functions necessary to complete the SSL handshake in SSLlib.java, namely sendHelo/parseHelo, sendCert/parseCert, sendKey/parseKey, and sendFinished/parsedFinished.

Background Research

OpenSSL

OpenSSL is an open source implementation of the SSL/TLS protocol specified in RFC 2246. The library is hosted on Github and is written in the C programming language. As we read through the source code we realize it was not the best resource to model our cryptographic protocol on. Thus we looked into RFC 2246 to understand the main ideas of the protocol.

RFC 2246

This protocol is an earlier and somewhat simpler version of TLS than is currently distributed, but in nonetheless captures the main points. It describes the information that client and server need to decide upon together (version, cipher), the information the server must send the the client (rand_s, public key, domain information) and the information the client must send to the server (rand_c, premaster secret). It also described in detail how the master secret is generated from the premaster secret, rand_c, and rand_s.

Features

The features below describe our implementation specifically, and may not be the same as those in RFC 2246 or those implemented by the current version of SSL.

Handshake

The handshake flow has 4 main parts, detailed below:

Helo

The HELO, first sent by the client, tells the server what version of SSL/TLS the client supports, the kind of cipher the client supports, the sessID the clients wants (always 0) and a random 32-byte rand_c value that will be used later. The server, upon receiving this information, parses and stores it. When ssl_accept is called and the server already has a Helo parse, it will send its Helo, in which it tell the

client the version, cipher, and sessID it has selected for the secure connection. For closedSSL, the server always mirrors what the client wants. The server also sends to the client a random 32-byte rand_s value. In our implementation, only one version and cipher is available for use.

Cert

Immediately after sending the HELO, the server sends its certificate. In openSSL, the client and server use a trusted third party, called a Certifying Authority, in order to ensure that the server's certificate, which contains its public key, comes from the real server and has not been tampered with. For our implementation, we did not use a separate certifying authority. Instead, we simulated a certifying authority by having the server sign its own certificate. This works as follows: first, we generate a public and private key for the certifying authority and save them into files. Upon initialization, each connection socket on the server side reads the private key from the file and saves it locally; the client does the same with the public key. When the server finishes writing its certificate, it signs it with the saved private key. The client will then verify the signature with the saved public key.

Key

Upon receiving S_DONE, the client needs to generate and send the premaster secret that will be used to generate the master key. The premaster secret is chosen at random and sent encrypted with the public key to the server. The client then uses the premaster secret, rand_c, and rand_s to generate the symmetric key (also referred to as a master key) by splitting the premaster secret in half and using a combined hashing and data expansion function to create two parts to then XOR, finally ending up with the symmetric key. The server uses the same premaster secret and random bits and goes through identical steps to generate its copy of the symmetric key. It was in generation of the master key that we most closely followed RFC2246. See page 11 and 12 of that RFC for a full explanation of the function genSymKey, which generates the master key. After the handshake is finished, every message sent between client and server is encrypted with the symmetric key.

Finished

The client would like to verify the server had received the messages it sent unaltered by an interfering third party, so it sends an encrypted digest of the messages it has send to the server. The server computes a digest of the messages it has received and checks it against the decrypted digest. Then the server repeats these steps to verify that the client had received the messages it sent unaltered.

Encryption/Decryption

With the handshake established, the actual encryption and decryption of data is quite simple and lightweight, as intended. When the application layer calls TCPSock's write,

the write function simply calls ssl_encrypt to replace its buffer with an encrypted buffer immediately before packaging the buffer into transports. When TCPManager receives a data packet, meanwhile, it calls ssl_decrypt to decrypt the buffer awaiting reads. In order to keep the encryption and decryption process so simple, a stream cipher must be used, so that bytes can be decrypted independent of the packet in which they arrive. We used RC4, which was used in early iterations of TLS precisely because it could be implemented so simply, but it has since been deprecated because it is susceptible to attack. In the future, we could improve the security of our SSL implementation by 1) using Message Authentication Codes to protect against bit flipping attacks (which all stream ciphers are susceptible to) and 2) using the stream version of AES instead of RC4.

Obstacles and Solutions

Calling ssl_connect and ssl_accept multiple times

Since ssl_connect and ssl_accept drive the entire SSL handshake, we need them to be called multiple times. First we attempted busy-waiting, with both in a while loop each, but this caused the thread carrying the socket to block. Instead, we changed each loop to if statements that returned if ssl_connect and ssl_accept were not done. Since the thread execute is called repeatedly, this effectively allowed ssl_connect to be called repeatedly without blocking. For ssl_accept, we used a similar strategy, but the function needed to be called inside of a TransferWorker thread (not TransferServer) so that it associated with a single connection socket.

Different Packet types for SSL Handshake

While we added different Transport types into the Transport class, the code we had did not accommodate these different types. Thus, we had to write a new sendPacket function that would accommodate the types of packets needed for the SSL handshake, namely HELO, CERT, S_DONE, C_KEYX, FINISHED, and ALERT.

Java Cryptography Library Obscurity

Java's cryptography library was very obscure, and using it was more difficult than we had anticipated. Seemingly small details like padding and Base64 encoding regularly broke our program, and on top of that there were inconsistencies between the library's capabilities and the RFC's demands.

Final Result

Our implementation of closedSSL has been submitted with this report. It assumes that every connection made will be an SSL connection. It also assumes zero loss.

Instructions for Compiling and Running the Code

*** code requires Java8. No additional downloads are necessary ***

```
$ cd fishnetJava-1.8-link
$ make
$ perl fishnet.pl simulate 2 scripts/transfertest.fish
```

Sample Output

```
Node 0: started
Node 1: started
Node 0: server started, port = 21
SS:Node 0: time = 1005 msec
Node 0: connection accepted
!.Node 1: connecting...
Server state == ESTABLISHED
Node 0: shaking hands ...
Client sock state == HANDSHAKE
Client:
.HELO sent
Node 1: shaking hands ...
HELO received
HELO received and parsed
Server sending HELO, CERT, and S_DONE
Server:
.HELO sent
.......CERT sent
.S_done sent
Node 0: shaking hands ...
Client state == HELO
Node 1: shaking hands ...
HELO received
HELO received and parsed
CERT received
CERT received
CERT received
CERT received
CERT received
CERT received
CERT received
parsing cert
CERT parsed and received
S_DONE received
sending key
..sent key
in genSymKey
.FINISHED sent
C_KEYX received
C_KEYX received
in genSymKey
FINISHED received
checking messages  from server
Digests match
.FINISHED sent
FINISHED received
```

```
Digests match
Client state == DONE

Node 1: time = 4010 msec
Node 1: started
Node 1: bytes to send = 3000
..sent 107 bytes of data
Node 1: bytes sent = 107
.::..sent 107 bytes of data
Node 1: bytes sent = 107
.::...sent 214 bytes of data
Node 1: bytes sent = 214
.::.::...sent 214 bytes of data
Node 1: bytes sent = 214
.::.::.....sent 321 bytes of data
Node 1: bytes sent = 321
.::.::.::.....sent 428 bytes of data
Node 1: bytes sent = 428
.::.::.::.::.....sent 428 bytes of data
Node 1: bytes sent = 428
.::.::.::.::.....sent 535 bytes of data
Node 1: bytes sent = 535
.::.::.::.::.::.....sent 642 bytes of data
Node 1: bytes sent = 642
.::.::.::.::.::..sent 4 bytes of data
Node 1: bytes sent = 4
.::Node 1: time = 14010
Node 1: sending completed
Node 1: closing connection...
Faccept error
Node 1: time = 15010 msec
Node 1: connection closed
Node 1: total bytes sent = 3000
Node 1: time elapsed = 11000 msec
Node 1: Bps = 272.72727272727275
accept error
Node 0: time = 16005 msec
Node 0: connection closed
Node 0: total bytes received = 3000
Fishnet exiting after time: 500020 msec.
Number of packets sent: 79
Number of packets dropped: 0
Number of packets lost: 0
```

Extra Credit

We have included a closedSSL specification that can be used for subsequent iterations of CPSC 433/533.

Acknowledgements

We would like to thank Professor Yang for his support and guidance on this project. This assignment has taught us a great deal about cryptography, SSL and TLS protocols, and TCP/IP!