Charles Schmitter
Crypto I – Prof. Yener
White Hat Implementation Write Up

Project Overview

As a brief overview, our team implemented a message exchange network program using the SSL protocol. The program begins with the SSL handshake protocol, in which the client and server negotiate which cryptographic primitives will be used for the duration of the connection. They then establish a symmetric key and proceed to exchange messages, utilizing the cryptographic primitives that were agreed upon. The cryptographic primitives were implemented by my partners, Nick and Ryan. I was responsible for implementing the network protocol that integrated these cryptographic primitives into our own implementation of the SSL protocol.

SSL Handshake Protocol

The handshake protocol is split between two files, client.py and server.py, and naturally these files represent the client and server in the protocol as mentioned above. The handshake begins with an exchange of hello packets, which initiates the connection. Once hello packets have been exchanged, the client sends a packet containing the cryptographic algorithms it has available to the server. Specifically, the client will specify the algorithms it has for symmetric

key encryption, key exchange, and hashing. The server will receive this packet and determine which algorithms it has in common with the client's algorithm listing. Once determined, the server sends a reply packet containing the algorithms that both parties have available. These algorithms are then used for the duration of the connection.

The key exchange algorithm we implemented, namely RSA, uses a set of public and private keys. Each party in the connection must have the other party's public key for the key exchange to be effective. Thus, after the cryptographic algorithms have been chosen, the client and the server exchange public keys. Once public keys have been exchanged, the key exchange algorithm can be used to send the symmetric key for the session from the client to the server.

The client randomly generates a symmetric key of 168 bits using Python's built-in secrets library, which allows for cryptographically secure random number generation. This symmetric key is encrypted with the server's public key, and then it is sent to the server. The server sends an acknowledgement message back to the client, which completes the handshake protocol.

SSL Message Passing Protocol

Charles Schmitter
Crypto I – Prof. Yener
White Hat Implementation Write Up

As mentioned in the overview, our program allows the client and the server to exchange messages. Once the handshake protocol is completed, message exchange can begin. The client begins the message exchange; on the client's terminal, there will be a prompt to enter a message. Once the message is entered, it is encrypted with the symmetric key algorithm selected during the handshake protocol. However, the encrypted message is not sent by itself to the server.

In addition to the encrypted message, the message authentication code (MAC) is also sent. Rather, a hash of the MAC is sent (HMAC). The HMAC is computed as follows:

```
# || is concatentation
MAC_HALF_2 = SHA1(sym_key
    || compressed.length
    || compressed)
```

```
MAC = SHA1(sym_key || MAC_HALF_2)
```

Note that the computation requires a compressed version of the plaintext. In our implementation, we compress the plaintext using Python's built-in gzip library. Once computed, the HMAC and the encrypted message are bundled together and sent to the server.

Once the server receives the message, it must verify the message's integrity and make sure the message has not been tampered with.

SSL Message Verification

Once the server receives a message from the client, it will decrypt the encrypted message with the symmetric key that was agreed upon during the handshake. However, it is possible that an adversary tampered with the message, and therefore, the server must verify that the HMAC of the decrypted message aligns with HMAC sent alongside the message. Note that decrypting the message will leave us with the compressed plaintext, so the server is readily able to use the compressed plaintext to compute the HMAC. The server computes the HMAC as mentioned in the previous section, and determines if the computed value is equivalent to the HMAC sent by the client. If it is equivalent, the server knows that the message was not tampered with, and it will continue communication. However, if the HMAC values are not equivalent, the server will terminate the connection.

Application Architecture

As briefly mentioned in the overview, the application we developed on top of the SSL protocol is a simple message exchanging program. The program logic is divided into a client and

server file, and each file contains functions that handle each packet needed in the aforementioned protocols. The client and server both utilize an AlgHandler object which holds the algorithms chosen during the handshake. This way, the client and the server can encrypt, decrypt, and hash without needing to handle each possible combination of cryptographic algorithms.

We utilized several built-in Python modules as well. For starters, we used the built-in secrets module for cryptographically secure random number generation. We also used the sockets library for the networking component.