Project: Secure Sockets Layer

Jakob Struye
20120612

December 13, 2016

1 Overview

I implemented the project in two parts. First there is the Somewhat Secure Sockets Layer (SSSL) implementing the general protocol. This part of the code is not aware of anything within the use cases (sending images etc.) and could be used within another project without having to modify the code. The application itself is SSSL_web, written in Python (and very little HTML and javascript) using the Django backend and a SQLite database. The next sections go over some implementation details and design decisions.

2 SSSL

The setup packets use the fields as described in the assignment. The only additional field is a length field present in all packets. This is mainly done for consistency with the payload packets as described later. Processing the setup packets and replying to them is all fairly straightforward. When a client opens a connection to the server, the server starts a new thread to listen to that client, so that the server can serve multiple clients simultaneously. The server stores all connection-specific variables in dictionaries indexed by the connection object to further allow serving multiple clients. Setup occurs in the main thread for the client, but listening for payload packets is performed on a second thread.

Payload packets have the following structure:

(NOT ENCRYPTED)

- Message type (1 byte): '0x07', indicating this is a payload-carrying packet
- Encrypted message length (4 bytes): The length of the remainder of the packet, in bytes. Using 4 bytes for this allows for very large payloads (up to 4GB). The receiver needs this value to know what part of the following byte stream to decrypt.

(ENCRYPTED)

- Session ID (2 bytes): Same as in the setup packets.
- Payload length (4 bytes): The length of the actual payload field, in bytes. This differs from the previous length field in that it refers to the unencrypted length. It is needed to know what part of the following byte stream should be sent to the application as payload.
- Payload (variable): The actual payload, as provided by the layer above SSSL.
- EndOfMessage (2 bytes) : '0xF0F0' as with the setup packets.

The first two fields are not encrypted as the receiver needs them to know if and how the rest of the packet should be decrypted.

2.1 Error management

As soon as any of the errors listed in the assignment is detected, the detecting end prepares an ErrorMessage. This again follows the format in the assignment, plus a length field for consistency. During the setup phase, the sending end immediately closes the socket after successfully sending the error packet. During the secure phase, the number of sent errors is tracked, and the socket is closed after the tenth error sent. When receiving an error, the error is logged and only in the setup phase the socket is closed. Upon receiving an invalid ErrorMessage (e.g. too short) the receiver will log this but will not reply with another error, to avoid ErrorMessage loops.
Most of the errors can be triggered by simply changing the sent or expected value in code. Bad encryption errors can be triggered by manually raising an exception in the try block where decryption occurs.

2.2 Handling TCP streams

One socket.send() does not necessarily translate to one socket.recv() on the other end. This is especially noticeable when sending multiple small packets in rapid succession (due to Nagle's algorithm). The code was designed to handle this. When receiving any amount of data, all received data is stored in a buffer. As soon as 5 bytes have arrived, the length field is available. With this it's possible to check if a full packet has arrived. If not, the receiver waits for a next .recv() before trying to process the packet. Each .recv() appends to the receive buffer and starts a new processing attempt. After a successful packet process the packet is removed from the buffer. If any data remains in the buffer, the peer attempts to process this immediately.

2.3 Using the code

The SSSL code is included within the SSSL_web project, in the SSSL subdirectory. The main functions in the server.py and client.py classes show how to use the code. To setup the server the user must supply the certificate and private key files. Then the user can add a number of accounts. This can be done with either a plain text password (for convenience) of a pre-hashed user ID + password (which is more secure). To set up the server the user must supply a certificate file, a user ID and a password (plain text). To connect to a server the user supplies a host and port. The client object immediately sets up the connection and the .connect() function returns as soon as a secure connection is established. Then send_payload() can be invoked. To handle received payloads, the user invokes add_payload_listener(listener). Every time a payload is received, the .callback(payload) function of this object is invoked, sending the payload to the user. This method can then return an error code to be sent back in an error message, or one or more payloads to be sent back in payload packets. With this mechanism, any application can send any type of payloads without having to modify the protocol code. This mechanism exists for both server and client.

3 SSSL web

My application uses a simple, fairly ugly web interface with the Django backend. As this is a security class, I did not focus on things such as layout and user-friendliness. The application can be started using 'python manage.py runserver' for localhost and 'python manage.py runserver 0.0.0.0:8000' to make it accessible from other interfaces. The url is ip:8000/app for the application and ip:8000/admin for database access (credentials admin/sssl1234)

3.1 SSSL setup and management

Only one SSSL server is used in the application, started in SSSL_web/urls.py. Note that this will only occur when a page is accessed for the first time. All accounts in the User table in the database are added to the server. Whenever a new user accesses the page (i.e. one who does not have a session key cookie), a new SSSL client is opened in app/views.py. In following page accesses by the same user, the Client object can be fetched from a static dict using the session key. Note that both the SSSL server and client are run in the webserver. To see that communication does in fact occur only via the SSSL messages and not e.g. via shared variables, run two separate instances of the webserver on two linked machines. Then on one machine just change the host that the SSSL client connects to, to the other machine's , in app/views.py.

3.2 Packet structure

A few different payload structures are defined for this application. They are created and parsed in app/ServerCallback.py and app/ClientCallback.py. The structures are consistent with the full SSSL packets' structures where useful.

Note that "message" is used for both a message entered by a user to be stored in the database, and a network message.

MessagePayload: sends message to server

- MessageType (1 byte): 0x01
- MessageID (1 byte): 0-255, identifying this message. This is included in the ack sent back by the server, so that the client can easily tell which message is being ack'd. 1 byte should be enough, as it's unlikely a user will send another 255 messages before the first one is ack'd (or even that many in total)
- Length (4 bytes): Length of the actual message in bytes.
- Message (variable): The actual message
- EndOfMessage (2 bytes): 0xF0F0

ImagePayload: sends image to server

- MessageType (1 byte): 0x02
- ImageID (1 byte): See MessageID
- NameLength (1 byte): Length in bytes of the filename. 1 byte should be enough for file name length
- FileName (variable): The file name of the uploaded image
- Length (4 bytes): The length of the binary representation of the image file. 4 bytes allows for images up to 4GB.
- Image (variable): Binary representation of the image
- EndOfMessage (2 bytes): 0xF0F0

Note that these do not include the user ID or a timestamp. The ServerCallback processing these payloads receives the user ID from the SSSL implementation, and the timestamp is created server-side.

ShowPayload: Lets the server know the clients wants to retrieve all data

- MessageID (1 byte): 0x03
- EndOfMessage (2 bytes): 0xF0F0

(Each show request + reply is the same, no need for a further ID)

AckPayload: Acknowledges properly received message or image to the client

- MessageType (1 byte): 0x04
- ID (1 byte): Either a MessageID or an ImageID. The same type of ack is sent for both. This means the client should keep their MessageID and ImageID values mutually exclusive (until they wrap).
- EndOfMessage (2 bytes): 0xF0F0

The show reply is more complicated. As they may contain an arbitrary number of images and messages, each is sent in a separate payload. This way the payload structure can be simpler, as it won't require arrays of values.

ShowReplyInitialPayload: Lets the client know a stream of messages and images is about to arrive

- MessageID (1 byte): 0x05
- ReplyID (1 byte): 0-255. Shared by all following messages and images in this reply, used to identify them as part of this reply.
- Length (2 bytes): Indicates how many images and messages are to follow. This way the receiver knows when all have been received. 2 bytes allows for over 65.000 items.
- EndOfMessage (2 bytes): 0xF0F0

ShowReplyMessagePayload: One message part of a larger show reply

- MessageID (1 byte): 0x06
- ReplyID (1 byte): see above
- Length (4 bytes): Length of the message to follow
- Message (variable): The actual message
- EndOfMessage (2 bytes): 0xF0F0

ShowReplyImagePayload: One image part of a larger show reply

- MessageID (1 byte): 0x06
- ReplyID (1 byte): see above
- Length (4 bytes): Length of the image to follow
- Image (variable): The actual image
- EndOfMessage (2 bytes): 0xF0F0

(The filename is not sent as it is not used. This could easily be added though)

Note that there is no explicit order of images and messages defined in the structure. The message and images are sent from most to least recent by the server. As TCP preserves order, they are also received in this order.

3.3 Web page

The page offers the 5 required buttons, along with a text field and image chooser. Only some buttons are enabled at any time. Whenever a button is clicked, the appropriate action is taken (usually sending a payload and waiting for a reply). As I really didn't want to write any non-trivial javascript, the page load blocks until an acknowledgement or list of items to show is received. This is checked every 0.1s through a shared variable. If nothing is received after 10 seconds an error is shown to the user. If it is successful this is also displayed. The page is created in HTML in app/templates/app/index.html. Some very basic Django-specific if/else clauses are used to decide which data to show. For images, an HTML <img>-tag is used to which the image in binary form is added as src. This way the client never has to store these to a temporary file.

4 Replay Attack

Most data sent in the SSSL setup can be generated and read by anyone. The client never needs to use its private key: no data sent to the client is only interpretable by the client. The only data secret to the client is its unhashed password. As it is always hashed in the same way, anyone that received a ClientKeyExchange can execute a replay attack pretending to be that client. The attacker can just generate a new client_random (sent plain text) and pre_master_secret (encrypted with server's public key, which the attacker has). The only data that has to be taken from an intercepted ClientKeyExchange is the client certificate and the hashed credentials. Both can just be copied into the forged ClientKeyExchange. The ClientHello and FinishedClient are sent just like in a regular connection setup. The attacker has the server_random (received plain text), pre_master_secret (which it generated and encrypted itself) and client_random (which it generated itself) so they can easily calculate the master_secret. This was implemented in a separate SSSL branch, in SSSL_replay.zip: the client does not have the certificate file or the password, but only an intercepted ClientKeyExchange, which I copied from a WireShark dump.

Note that a full-on replay attack (replaying the entire ClientKeyExchange) would not work. The pre_master_secret is already encrypted there and can only be encrypted using the server's private key. Without a decrypted pre_master_secret, the attacker cannot calculate the master_secret and cannot send or decrypt any traffic in the secure phase.

5 Replay Attack Fix

As we did not have the client's private key, no fixes using that were possible. I opted for a simpler fix. The main weakness here is that the attacker can just replay the hashed password without knowing the plain text password. This can be fixed by applying a second round of hashing to this value, this time with a nonce. I simply use the server_random: I append the server_random to the regular hashed credentials and hash that again. Both a legitimate client and server can easily perform this. Assuming the server does not reuse server_randoms, there is no way for the attacker to calculate this hash (ignoring SHA1 weaknesses and assuming the client does not lose their password). This is a rather convenient fix as it does not require any changes to the message structure. It is implemented in the main branch of SSSL.

6 Other attacks

The main point of this protocol is to counteract the release of message contents, so this type of passive attack is clearly not possible. Assuming AES-128-CBC is secure, an eavesdropper cannot read any of the encrypted traffic without knowing the master secret, which is never transmitted. Traffic analysis is still possible as an eavesdropper can rougly tell size and frequency of packets being sent. This is hard to defend against and would require a significant overhaul of the protocol.

Considering active attacks, the password hashing fix above patches the weakness to replay attacks. Masquerading as the server side (e.g. through a man in the middle attack or by replaying server messages) would not work as the attacker would not have the server's private key, needed to decrypt the pre_master_secret. The attacker also should not be able to get their own certificate for the same domain as the legitimate server. Some types of modification of messages are possible. While an attacker cannot (reliably) modify the message contents, they can be reordered (e.g. by modifying TCP sequence numbers) or delayed. The reordering could be counteracting by adding a SSSL-level sequence number in the encrypted part of the message. Finally denial of service attacks are definitely possible. Attacks equivalent to the TCP SYN attack could open up a lot of new SSSL connections without finishing the setup phase. Furthermore, the protocol supports very large payload sizes (up to 4GB) making it rather simple for an attacker to generate out-of-memory errors at the server.

Considering the application, there is also an obvious sercurity hole. The user passwords are stored in plain text in the database. This is however easy to replace with the already hashed values. It was done this way for convenience. Also note that if multiple images are uploaded with the same name, the newest will overwrite any older ones. This can be fixed by generating new, unique filenames at the server.