Assessment 4 - Protocol

COSC540 - Networks and Information Security

Andrew Weymes

2023-05-03

Introduction

This client-server model utilises elliptic curve cryptography in conjunction with SSL and BIO from the C implementation of the OpenSSL library. The overall protocol has not change substantially since assessment 2, rather, it has had a TLS layer added to mitigate some network security risks.

The protocol as described below, outlines the server implementation of the current working protocol, what it expects to receive, and what can be expected to be returned.

SSL/TLS Protocol (The server)

1. Generates an elliptic curve key pair using the NID_X9_62_prime256v1 curve.
2. Uses the key pair to generate a self-signed SSL/TLS certificate that is valid for one year.
3. Initializes an OpenSSL SSL context object.
4. Loads the SSL/TLS certificate and the private key into the SSL context object.
5. Initializes an OpenSSL listen socket (BIO) to listen for incoming connections.
6. Binds the listen socket to the specified IP address and port and waits for a client to connect.
7. When a client connects, the server accepts the incoming connection and retrieves the client BIO.
8. Initializes an SSL object for the connection using the SSL context.
9. Sets the client BIO for the SSL object and performs the SSL/TLS handshake.
    a) During the SSL/TLS handshake, the server sends its SSL/TLS certificate to the client for verification (this implementation uses a self signed certificate for development and testing purposes).
    b) After the SSL/TLS handshake, the server and client can exchange application data securely using the SSL/TLS protocol.

SSL/TLS Protocol Implementation in C

1. Generate a key pair using elliptic curve cryptography algorithm (NID_X9_62_prime256v1 curve).

```
// Generate EC keys
EVP_PKEY *keys;
EVP_PKEY_CTX *key_ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);
EVP_PKEY_keygen_init(key_ctx);
EVP_PKEY_CTX_set_ec_paramgen_curve_nid(key_ctx, NID_X9_62_prime256v1);
EVP_PKEY_keygen(key_ctx, &keys);
EVP_PKEY_CTX_free(key_ctx);
```

2. Use the key pair to generate a self-signed SSL/TLS certificate.

```
// Generate and self-sign certificate
X509 *cert = X509_new();
X509_set_version(cert, 2);
ASN1_INTEGER_set(X509_get_serialNumber(cert), 1);
X509_gmtime_adj(X509_get_notBefore(cert), 0);
X509_gmtime_adj(X509_get_notAfter(cert), (long) 60 * 60 * 24 * 365); // valid for 1 year (in seconds)
X509_set_pubkey(cert, keys);
X509_sign(cert, keys, EVP_sha256());
```

3. Initialize an OpenSSL SSL context object and load the SSL/TLS certificate and the private key into it.

```
SSL_CTX *ctx = SSL_CTX_new(TLS_server_method());
SSL_CTX_use_certificate(ctx, cert);
SSL_CTX_use_PrivateKey(ctx, keys);
SSL_CTX_check_private_key(ctx);
```

4. Initialize an OpenSSL listen socket (BIO) to listen for incoming connections and bind the listen socket to the specified IP address and port.

```
BIO *bio = BIO_new_accept("host:port");
BIO_do_accept(bio);
```

5. Wait for a client to connect, accept the incoming connection and retrieve a client BIO. Note that ‘BIO_do_accept(bio); is called immediately after it was called previously.

```
BIO_do_accept(bio);
BIO *client_bio = BIO_pop(bio);
```

6. Initialize an SSL object for the connection using the SSL context and set the client BIO for the SSL

object and perform the SSL/TLS handshake. Note that the SSL context will automatically negotiate the SSL/TLS handshake upon accept.

```
SSL *ssl = SSL_new(ctx);
SSL_set_bio(ssl, client_bio, client_bio);
```

7. Following this, the server utilises multithreading to handle concurrent clients which is out of the scope of this protocol description. For secure reading and writing of messages to and from a client, the server will use `SSL_read()` and `SSL_write()`. Both of which take the same argument, that is, a client BIO, a buffer array, and the maximum size of the buffer if reading, otherwise, the length of the array if writing.

Messaging Protocol

The messaging protocol will allow a maximum of five (5) concurrent users to interact with the server. User data is stored in a server defined data structure that holds key:value pairs. These structures are held in a dynamic array that is stored in another server defined data structure that holds information about the user session such as client id, the number of stored key:pairs, and an array of key:pairs.

The following keywords should be passed in capital letters.

CONNECT

The server requires the CONNECT keyword to be the first argument it receives. It should be sent in conjunction with a user ID.

E.g.,

```
CONNECT myUserName
```

If successful, the server should respond with `CONNECT OK`, otherwise, `CONNECT: ERROR`. On a connect error, the server will close the connection, and the client will need to reconnect.

PUT

The PUT keyword takes two arguments in succession as a key:pair for storage. First the PUT argument is passed with a key value. After pressing enter, a newline should appear and the value to be stored should be entered followed by pressing enter.

E.g.,

```
PUT this
here
```

If successful, the server will respond with `PUT: OK`, otherwise, `PUT: ERROR`. NOTE: if the key:pair already exists and the same key is used to store a new value, the old value will be overwritten with the new value.

GET

The GET keyword takes a single keyword argument. The keyword should correspond to the key used to store a value in the PUT command.

E.g.,

```
GET this
```

If successful, the server should respond with the value stored with this key, in the case of this example, the server should respond with `here`. On errors, the server will respond with `GET: ERROR`.

DELETE

The DELETE keyword takes a single keyword argument. the keyword should correspond to the key used to store a value in the PUT command.

E.g.,

```
DELETE this
```

If successful, the server should respond with `DELETE: OK` and the stored key pair will be deleted. On error, the server will respond with `DELETE: ERROR`.

DISCONNECT

The DISCONNECT keyword takes no arguments. Entering this keyword will delete all client data, return `DISCONNECT: OK` to the client, and gracefully close the connection.

Should any of the keyword arguments, `CONNECT`, `PUT`, `GET`, `DELETE`, or `DISCONNECT` be entered incorrectly, the server will close the connection and all stored data will be deleted.