T-409-TSAM-2016: Computer Networks
Programming Assignment 3: SSL

Lecturer: Marcel Kyas

October 5, 2016

Submission Deadline

You must submit the solutions to this assignment in MySchool until 23:59:59 GMT on November 9, 2016.

Intended Learning Outcomes

You should be able to:

- Design a network protocol.
- Implement the protocol in a client and a server application.
- Use OpenSSL to ensure privacy and authenticity.
- Explain why their implementation satisfies security properties.

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	10	10	20	10	20	20	10	0	0	100
Score:										

Questions 8 and 9 are bonus questions for 10 points, each. The maximum possible grade for this assignment is 12.

1 Instructions

This time, you shall implement both a client and a server for a secure chat protocol. Design a communication protocol and implement a server and a client program that implements the protocol. Use SSL to enforce privacy and authenticity.

The concepts needed for this assignments can be found in Chapter 8 of Kurose and Ross (2013).

Your grade will depend on the quality of the implementation, the explanation of the protocol, the justification of the security properties, and the number of possible attacks to the integrity and privacy of the communication that your solution can mitigate.

I strongly advise you to read this sheet, the documentation and web pages linked in this document and the man pages to OpenSSL carefully before you start coding.

If you get stuck, ask on Piazza. If you are unsure if you may ask your question, ask the question privately and anonymously. If a question is asked often, or we like the question, we reserve the right to publish private question, maybe with some edits.

2 A secure chat system

Examples of *insecure* chat service are:

Internet Relay Chat is specified in RFC 2812 `https://tools.ietf.org/html/rfc2812`

XMPP or Jabber is specified in RFC 6120 to 6122[1]

You are *not* asked to implement any of the protocols. They are referenced to inspire you. Pick what you like, find useful, or design your own.

The commands specified for the client below are taken from IRC.

2.1 Command Line Interface

It must be possible to run the server using this command:

```
[student16@skel pa3]$ make -C ./src
[student16@skel pa3]$ ./src/chatd $(/labs/tsam16/my_port) &
```

The client should be started by

```
[student16@skel pa3]$ ./src/chat hostname $(/labs/tsam16/my_port)
```

To obtain a port that you can use, call `/labs/tsam16/my_port`.

Since TCP will generate fresh ports for each connection, there is no guarantee that the port is free for you to use. Then, count the port number up until you find a free port.

2.2 Protocol Design

As part of the assignment, you shall design the line protocol used by the client and server. Explain the message format and the semantics of each message.

Hint: You can either send binary messages, like in TFTP (PA1), or text based messages, like in HTTP (PA2). If you choose a binary format, it is usually a good idea to include the length of a variable length data field before, because you know how much memory to allocate. However, attackers may try to crash the server by providing invalid data in these fields. If you choose a text base format, you have to be able to deal with streams of random data, i.e. deal with `'\0'` and other non-printable symbols in the stream.

A chat should either be point to point (i.e. two persons exchanging messages) or multicast (messages should be duplicated to users registered to a chat room). Access to chat rooms and point to point communication should be authenticated by maintaining a data base of registered users and credentials.

You have to decide if the communication is implemented in a client and server architecture or if the communication uses a peer-to-peer architecture.

[1] See `https://xmpp.org/xmpp-protocols/rfcs/` for the complete list.

2.3 Getting Started

The C Programming Language does not provide a well-rounded standard library. Do not implement your own data types. Instead, I suggest to use the following packages:

- GLib 2 (`https://developer.gnome.org/glib/`) for most common data structures, command line argument parsing and so on. Don't use the Fundamentals and the Core Application support functions except necessary (the general memory allocation functions are okay, however). The utility functions and especially the data types, however, are recommended.

Again, the use of any of these packages is optional. If you don't like them or have a better implementation, use this.

There is sample code in `src` to get you started. In the source code of the client program, `chat.c`, there is code for handling input from the terminal using GNU readline library. The code is neither complete nor correct, and you should modify it. Especially the code to parse the input from the terminal can be improved.

If the client crashes, your terminal may appear to be in an unusable state, because no input nor output is necessarily visible in the shell. If this happens, type the return key and then `reset` followed by return to reset the terminal into a usable state.

Also, to make things simpler to understand and to modify, the source code does not implement any windowing. What ever you print from the client may be interleaved with the input prompt, just line in the shell if you execute a job in the background with output while typing. If this bothers you, you can type CTRL-L to clear the screen and get the current prompt redisplayed. This will also erase the screen. This is not nice, but the alternatives would make the code too complex.

To have us answer efficiently, please provide a short snippet of code that you suspect to be wrong, a description of the behaviour you expect, a description of the behaviour you observe, and your question.

2.4 OpenSSL

Setting up secure connections is complex and involves cryptographic methods outside of the scope of this lecture. Instead, you are asked to use the OpenSSL library (`https://www.openssl.org/`). I am aware that there are many issues with OpenSSL and that that the library is criticized heavily.[2]

A tutorial introduction to OpenSSL is provided at this web page `http://h71000.www7.hp.com/doc/83final/ba554_90007/ch04s03.html`.

3 Tasks/Questions

Your hand-in must conform to the following to be graded:

[2]Mozilla's Network Security Services (NSS) library is considered the best alternative, but it is much more complex to use. GnuTLS is a runner up, but has a smaller developer base and had similar security issues that OpenSSL had. Other implementations of the library are not employed widely or limited to very narrow application domains.

- All necessary files need to be stored in an archive, especially the content of the `src/` directory. If unpacked, we expect to see a `./README` file and a subdirectory `./src/` containing the source code and a `Makefile`; do not have those files in a subdirectory.
- You must have the generated keys in the top-level directory.
- You must include a file called `./AUTHORS` that includes the name of each group member followed by the e-mail address *at ru.is* enclosed in angle brackets (see example) on separate lines.
- The `./README` file **must** contain a specification of the protocol, i.e. message format and how a client and a server should react to a message. It should contain notes about the implementation.
- The default rule in `Makefile` shall compile the `chat` and `chatd` programs.
- Do not forget to comment your code and use proper indentation.

Make sure that your implementation does not crash, does not leak memory and does not contain any security issues. Argue why communication is authenticated and private. Explain why there is no unexpected information flow. Try to keep track of what comes from the clients, which may be malicious, and deallocate what you do not need early. Zero out every buffer before use, or allocate the buffer using `g_new0()`.

1. (10 points) Key management

   To establish any connection via OpenSSL, you need to generate a certificate that is used to establish the connection. Read the OpenSSL cookbook at `https://www.feistyduck.com/books/openssl-cookbook/` (the HTML version is free to read) and follow the steps described in the cookbook to create a self-signed certificate.

2. (10 points) OpenSSL initialisation

   Modify `chat.c` and `chatd.c` to initialize OpenSSL and to load your self-signed certificate.

3. (20 points) Secure client server connection

   The point of this part is to establish an encrypted connection, not to authenticate the user to the server. Modify `chat.c` and `chatd.c` to establish an encrypted connection between `chat` to the server program `chatd` using TCP and OpenSSL.

   Upon success, the server should send the message `Welcome.` to the client indicating that the connection is established. For each established connection, the server shall log a line

   ```
   <timestamp> : <client ip>:<client port> connected
   ```

   where “timestamp” is in ISO 8601 format precise up to seconds, “client IP” and “client port” are the IP and port of the client sending the request.

   A connection shall be cleaned up and closed if the client issues a `/bye` command on the server. A closed connection shall be logged with

   ```
   <timestamp> : <client ip>:<client port> disconnected
   ```

4. (10 points) List of connected users

   Your server should handle multiple connections. Use `select()`, like in PA2.

   Implement a command `/who` to list the names of all users available on the system. It shall display the authenticated user name if available (see below on authentication), the IP address and the port number of the client, and the chat room the client is currently in if available. For now, assume that the user name and the chat room are `NULL`, i.e. they have not been set yet.

   Hint: You have to keep track of all file descriptors in the server, just like with PA2. OpenSSL supplies `SSL *` handles for the secure connection. The `SSL *` handle is created from a socket file descriptor (see the tutorial and `chat.c`) and maintains state of the encryption. The socket is still needed for calls to `select()` and `shutdown()`. The functions `SSL_read()` and `SSL_write()` receive or send encrypted messages, while `read()` and `write()` (and the receive functions) on the file descriptor of the socket receive and send data unencrypted. The underlying TCP stream will merge the data in weird ways and may cause unexpected behaviour or crashes. Follow the instructions of the tutorials linked above.

   A reasonable data structure to take maintain the logged in users is a binary balanced tree (GTree). You can create it with `g_tree_new(strcmp)` to map strings to any data object or `g_tree_new(sockaddr_in_cmp)` to map socket addresses to any data object. The function `sockaddr_in_cmp` is implemented in `chat.c` for your convenience.

   To iterate over the elements of such a tree, call `g_tree_foreach`. For example, to output all the keys in a tree, you may want to the function (data points to some memory location that you supply to `g_tree_foreach` and in which you can accumulate what you need):

```
void func(gpointer key, gpointer value, gpointer data)
{
    printf("%s\n", key);
}
```

5. (20 points) Chat room

   Implement a command `/list` to list the names of all available public char rooms and a command `/join name` to join one of the rooms. A client can only be member of one chat room at a time, a subsequent join will make a client to leave a chat room and join the next one. Remember that the client also leaves a chat room by termination.

   While a client is member of a chat room, it will receive all messages posted to the room. Every input that is not a command is a message posted to the chat room the client is currently a member of. The client sends an error message if the client is not member of a chat room.

   Keep track of chat rooms and the clients that are a member of the room. Identify the clients by the connection information (its IP address and the port number). Clients may use nick names that can change over the lifetime of the connection.

   You can maintain the chat rooms as a binary balanced tree (GTree), that maps a string, the chat room name, to a set of its members. Create the tree as `g_tree_new(strcmp)` to get a tree that compares on strings. The set of its members may be maintained in a doubly linked list (GList), but make sure not to insert duplicate names into the list.

Make sure that all messages are sent and received over the encrypted connection. Make sure that only members of the chat room receive the message and not any other user.

6. Authentication

   Upon establishing the connection to the server, the user should authenticate himself/herself to the server. That means, the user shall provide some kind of proof that he/she is the person it claims to be.

   6.1. (15 points) Implement a command `/user username` to register as username. Have the client ask for a password that is used as a shared secret between client and server. You may use `getpasswd()` in `chat.c` to get a password without it appearing on the screen.

   On the server side, a user shall be disconnected if the password does not match within three trials, there shall be a delay between each attempt, and the password shall be processed securely.

   If authentication is successful, a line

   ```
   <timestamp> : <client ip>:<client port> <user> authenticated
   ```

   shall be printed. For each unsuccessful try, a line

   ```
   <timestamp> : <client ip>:<client port> <user> authentication error
   ```

   shall be logged.

   Hint: You have two choices to send the password: as plain text or as a hash. An eavesdropper will not observe the communication in any case, because the connection is encrypted. A hashed password can be kept secret from the server, too.

   To make it harder for an attacker to obtain passwords, generate a random string, called *salt*, and store it for later referencing. Prepend the salt string to the password and hash that string with a strong cryptographic hash function (`sha256` is still considered safe). This hashed string is stored with the server. Whenever the client needs to ask for a password, prepend the salt string to the input password, hash it with the same hash function compare it to the stored has value.

   To protect against dictionary attacks and rainbow table attacks against the salted hashes, key spreading is suggested. That is, the hash function is reapplied to the hash multiple times. Depending on the source, $10,000$ to $200,000$ iterations are suggested. This method essentially slows an attacker down, if the number of iterations is known.

   Check the function `PKCS5_PBKDF2_HMAC` as a helper function for generating password hashes.

   Depending on the nature of your data, you may want to store it. The GKeyFile can be used to store and parse files that map keys to values. You can read the contents of a file into a key file using `g_key_file_load_from_file()`. You can convert a key file to a string `g_key_file_to_data()` and the string can then be saved to a file for later access.

   If you want to save binary data in a key file, e.g. the result of a hash function, you may encode it in Base64, a space efficient encoding of binary data in printable netascii characters.

   Example to create a key file containing a password (without error checking or error handling):

```
GKeyFile *keyfile = g_key_file_new();
gchar *passwd64 = g_base64_encode(passwd, length);

g_key_file_set_string(keyfile, "passwords", "robby", passwd64);
gsize length;
gchar *keyfile_string = g_key_file_to_data(keyfile, &length);
write(fd, keyfile_string, length);
g_free(keyfile_string);
g_free(passwd64);
g_key_file_free(keyfile);
```

Example to decode a password from a key file (without error checking or error handling):

```
GKeyFile *keyfile = g_keyfile_new();
g_key_file_laod_from_file(keyfile, "passwords.ini",
                          G_KEY_FILE_NONE, NULL);
gchar *passwd64 = g_key_file_get_string(keyfile, "passwords",
                                        "robby", NULL);
gsize length;
guchar *passwd = g_base64_decode(passwd64, &length);
```

You may use the key file like a small but inefficient data base. As long as there are not many users, you should not notice any difference.

6.2. (5 points) *Questions:* Where are the passwords stored? Where are the salt strings stored? Why do you send the plain text password/hashed password? Can someone read the password in transit? What are the security implications of your decision?

7. Private messages

7.1. (7 points) Any client may send a private message to another client using the command `/say username msg`. The message should be delivered to this user only instead of the chat room the user is currently in.

7.2. (3 points) *Logging of private messages:* Should private messages be logged? If so, what should be logged about private messages? What are the consequences of your decisions?

8. Idle timeouts

It is often a good idea to terminate connections to clients that were idle for some time. If a user leaves the system unattended, we avoid abuse by others.

8.1. (7 points (bonus)) Keep track of the clients idle time, i.e. the time that they did not send any commands, and disconnect the client after that time. If a client gets disconnected by a time out, the event shall be logged by a line

```
<timestamp> : <client ip>:<client port> timed out.
```

8.2. (3 points (bonus)) Explain the security implications of not idling out a unused connection. How do you make sure that your idle time out happens on time?

9. Dice

   9.1. (7 points (bonus)) Implement the following silly game for the chat service. Two clients may play a game of fortune by issuing the command `/game username`. The challenged user may accept or decline to play.

   Once a game started, `/roll` shall generate the sum of two pseudo-random numbers from 1 to 6. The winner is the one with the highest result.

   9.2. (3 points (bonus)) Explain the possible attacks or how one might try to cheat. Explain how you defend agains those attacks.

   Hint: The preferred way to generate pseudo=random numbers from 1 to 6 is this:

```
(int) floor(drand48() * 6.0) + 1
```

References

James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, 2013.