CS 558 Introduction to Computer Security, Spring 2024
Programming Assignment 4

Out: 4/23/2024 Tue.
Due: 5/11/2024 Sat. 23:59:59

Introduction

In this project, students will enhance their understanding of secure communications by integrating OpenSSL into the basic client and server programs. Initially, students will be given traditional socket programming code that facilitates non-secure communication between a client and a server. Along with this, students will receive several key and certificate files necessary for establishing secure communications. Students are required to make modification to the existing code to convert it into the secured version.

Included Files

Before going forward, download the compressed files from Brightspace. In this section, explanations of the provided file will be given.

1. `server.c` : Implements a traditional, unsecured server that handles basic socket communication. It reads an integer message from the client, increments this number by 1, and sends the result back to the client. The server terminates the connection once the client disconnects.

2. `client.c` : This file contains the implementation for the client program. The client reads an input integer from the user via the terminal, sends it to the server, and waits for a response. Once received, it displays the result on the terminal. The client connects to the server assuming it is hosted on the same machine (`localhost`).

3. `Makefile` : A script to simplify the building of the client and server programs. Executing `make` within the source folder will compile both programs automatically.

4. `server.key` : Contains the server's private key. This key is used by the server to sign communications, confirming its identity during interactions with clients.

5. `server.crt` : Holds the server's public certificate, used by clients to encrypt data sent to the server or to verify signatures created with the server's private key. This certificate is signed by a trusted Certificate Authority (CA), which, for this project, is represented by the instructor. This validates the authenticity of the certificate.

6. `self_signed_server.crt` : This file is an alternative certificate of the server. It is self-signed by the server instead of the CA(instructor), thus viewed as untrusted. It will be used to test the authentication process.

7. `ca.pem` : This is the file that contains the CA's certificate. Usually, the client will have access to this file and use it to verify the server's certificate, which is signed by the CA(instructor). This verification establishes trust that the server is indeed who it claims to be. This verification process can also be applied in reverse.

8. `client.key, client.crt, self_signed_client.crt` : These files are relevant only for the bonus mutual authentication feature. They are described in more detail in the bonus section of the project documentation.

To summarize the the key and certificate usage:

- At initialization, the server loads its private key (`server.key`) and its certificate (`server.crt`).
- The client loads the CA's certificate (`ca.pem`) to verify the authenticity of the server's certificate received during the connection setup.

Basic Client & Server Programs

In this section, it demonstrate how the given client server programs work.

1. **Build the program**: Navigate to the root of the source directory and run the command `make`. This will compile the source files and generate executable files named `client` and `server`.
2. **Start the Server**:
   - Launch the server by executing "`./server 8999`" in the terminal. Here, 8999 represents the port number the server will listen on. You can replace 8999 with any other available port number that does not conflict with well-known service ports.
   - Upon starting, the server will display "`Listening on port 8999`" , indicating that it is ready to accept connections.
3. **Launch the Client**:
   - Open a new terminal window on the same machine and start the client by running "./client 8999", using the same port number as the server.
   - Once started, the client prompts you with "`Input a number:`" for input. Simultaneously, the server will print "`Connection from 127.0.0.1:52322`" (or another client-specific address and port), showing that a client has successfully connected.
4. **Iteract with the Client and Server**:
   - Enter a number, for example, `10`, at the client prompt. This number is sent to the server, which then logs "`Received: 10`".
   - The server processes this input by adding one to the number, resulting in 11, and sends this back to the client. The client then displays "`Received from server: 11`".
   - To end the session, the client can disconnect by typing "`exit`" or by using keyboard shortcuts `ctrl+D` or `ctrl+C`. The server will print "`Client closed the connection`".

Secured Client & Server Programs

In this section, it shows what you are required to implement in this project.

- **Familiarize Yourself with OpenSSL**: Start by exploring OpenSSL through available tutorials and resources such as those found on opensource.com and the HP documentation. Pay attention to how secure contexts are initialized, and how keys and certificates are managed within these contexts.
- **Initialize the Secure Context**: Use TLS for initializing the secure context, avoiding outdated protocols like SSL2 or SSL3.
- **Server Argument Expansion**: Modify the server to accept four arguments instead of the previous one. The arguments should include:
  1. The port number.
  2. Authentication mode (`oneway` or `mutual`), with `oneway` meaning only the client authenticates the server.
  3. Server's private key (e.g., `server.key`).
  4. (**optional**)Server's certificate (e.g., `server.crt`). If not provided, the server should still function without it.

  Example command for running the server could be: "`./server 8999 single server.key server.crt`" or "`./server 8999 single server.key`" if no server certificate is provided.
- **Server Authentication Mode**: Depending on the `mode` argument, the server should decide how to authenticate clients. For `oneway` mode, client authentication isn't required.
- **CA Certificate Loading**: The server optionally loads a CA certificate (`ca.pem`) based on the authentication mode. For base implementation, no CA certificate is needed for the server.
- **Secure Socket Operations**: Replace regular socket read/write operations with their secure counterparts: `SSL_read() and SSL_write()`.
- **Client Modification**:
  - The client accepts a single argument, the port number.
  - The client always loads the CA certificate `ca.pem`.
  - The client uses this CA certificate to verify the server's certificate.
  - The client verifies the server public key using peer mode, and treat no certificate provided from server as failure. The peer mode will let the client verifies the server during the communication.

  - The client should only communicate with server with a valid certificate signed by the CA. If the server certificate is not signed by the CA, the client should print "`Server certificate verification failed`". If the server does not provide a certificate, then the client print "`Server certificate missing`". Then, the client should disconnect and shutdown the SSL connection correctly before terminating itself.
  - The client also use secured version `SSL_read(), SSL_write()` for reading/writing the socket.

- **Testing Scenarios**:
  - Start the server with 3 cases:
    * Use the CA signed server certificate `server.crt` to properly authenticate itself to the client.
    * Utilize `self_signed_server.crt` to check if the client correctly rejects a server with an untrusted certificate.
    * Test client behavior when no server certificate is provided.
  - Start the client normally with command "`./client 8999`"

- **Maintain Original Program Logic**: While you can modify the code freely, ensure that the core functionality and logic of the client and server programs remain consistent with their original design. You can also create additional key and certificate files for testing purposes.

Bonus: (Extra 20 points)

To get the bonus points, you have to implement the **mutual** authentication for this secured client server communication.

- **Server Modification**:
  - **Support Mutual Authentication**: Modify the server to handle a **mutual** authentication mode. In this mode, the server should load the CA certificate (**ca.pem**) to verify client certificates.
  - **Client Certificate Verification**: The server program should only communicate with clients with a valid certificate signed by the CA. If the client certificate is not signed by the CA, the server should print "`Client certificate verification failed`". If the client doesn't provide a certificate, then the server print "`Client certificate missing`".
  - **Connection Shutdown:** Ensure that the server properly closes the connection if the client's certificate is invalid or missing.

- **Client Updates**:
  - Argument Expansion: The client now takes three arguments instead of one:
    1. The port number.
    2. The client's private key (e.g., client.key).

3. (**optional**)The client's certificate (e.g., client.crt). If not provided, the client should still attempt to connect, triggering the server's error for missing certificates.

Example command for running the client could be: "`./client 8999 client.key client.crt`" or "`./client 8999 client.key`" if no client certificate provided.

- **Testing Scenarios**:
  - Start the server in mutual authentication mode with "`./server 8999 mutual server.key server.crt`".
  - Start the client with 3 cases:
    * Use the CA signed client certificate `client.crt` to properly authenticate itself to the server.
    * Utilize `self_signed_client.crt` to check if the server correctly rejects a client with an untrusted certificate.
    * Test server behavior when no client certificate is provided.

For this project, you have to implement it in C programming language. Ensure your code functions correctly on the department's machines, as grading will occur in this environment.

Log and submit your work

Log your work: besides the source files of your assignment, you must also include a README file which minimally contains your name, B-number, programming language you used and how to compile/execute your program. Additionally, it can contain the following:

- The status of your program (especially, if not fully complete).
- Implemented the bonus or not. If you implement the bonus, explain why your code work for bonus.
- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Any other material you believe is relevant to the grading of your project.

Compress the files: compress your README file, all the source files in the same folder, and any additional files you add into a ZIP file. Name the ZIP file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the zip file should be "proj4_abc.zip".

Submission: submit the ZIP file to Brightspace before the deadline.

Grading guidelines

(1) Prepare the ZIP file on a Linux machine. If your zip file cannot be uncompressed, 5 points off.

(2) If the submitted ZIP file/source code files included in the ZIP file are not named as specified above (so that it causes problems for TA's grading scripts), 10 points off.

(3) If the submitted code does not compile or execute:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(4) If the code is not working as required in this spec, the TA should take points based on the assigned full points of the task and the actual problem.

(5) Late Penalty: Day1 10%, Day2 20%, Day3 40%, Day4 60%, Day5 80%

(6) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with your fellow students, but code should absolutely be kept private.