

Security Engineering: Cryptography

File of Documentation for SENG 360 Assignment 3

By

Nicola Paterson (V00839810)

Subah Mehrotra (V00855471)

Table of Contents:

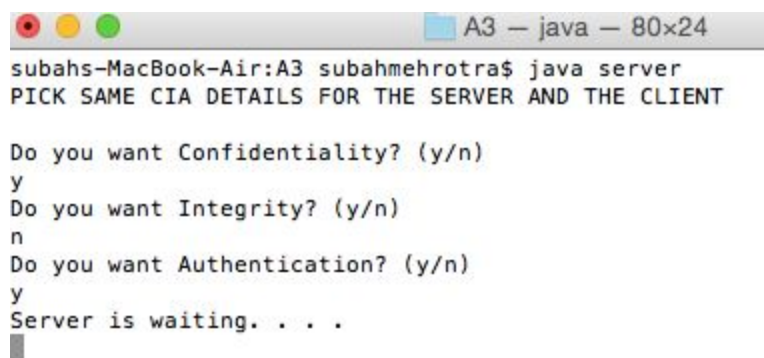
| | |
|--|---|
| Introduction..... | 2 |
| Confidentiality..... | 3 |
| Java Libraries Imported for Confidentiality..... | 4 |
| Integrity..... | 5 |
| Authentication..... | 6 |
| End Session and Reconnect..... | 7 |
| Compiling Instructions..... | 7 |
| Step-by-Step Instructions..... | 8 |
| Deliverable..... | 8 |
| Conclusion..... | 9 |

Table of Figures:

| | |
|---|---|
| Figure 1: Security properties on the server..... | 2 |
| Figure 2: Security properties on the client..... | 2 |
| Figure 3: Encrypt function in the code..... | 3 |
| Figure 4: Decrypt function in the code..... | 4 |
| Figure 5: Various functions used to maintain the integrity of the messages..... | 5 |
| Figure 6: Entering username and password on the client..... | 6 |
| Figure 7: Reconnect function in server.java..... | 7 |

Introduction

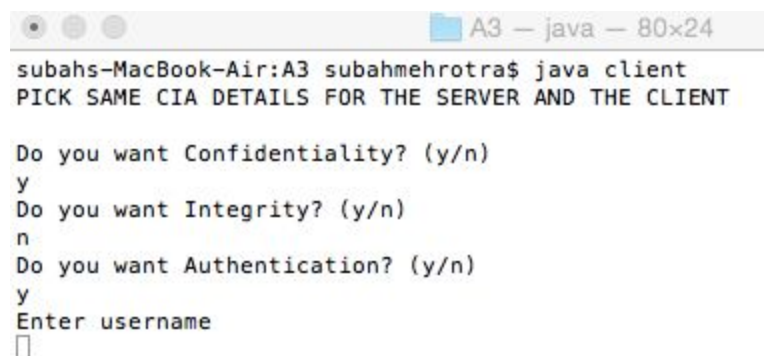
This document contains the details for an instant messenger, which uses java socket and threads to communicate between two parties. The parties, a client and a server, decide the security properties through answering yes or no to the security property prompts (Figure 1 and Figure 2). The security properties should have the same answers for both the client and the server. Then, the messenger system initializes to the waiting server. The messenger between the two parties continues until both client and server input 'END'. The following sections of this document detail the three security options, as well as how to compile the program, how to execute it step-by-step, and the files belonging to this project.

A screenshot of a macOS terminal window titled "A3 - java - 80x24". The prompt is "subahs-MacBook-Air:A3 subahmehrotra\$". The user has entered "java server". The output shows the server starting and asking for security preferences: "PICK SAME CIA DETAILS FOR THE SERVER AND THE CLIENT", "Do you want Confidentiality? (y/n)", "Do you want Integrity? (y/n)", and "Do you want Authentication? (y/n)". The user has responded with "y", "n", and "y" respectively. The server then says "Server is waiting. . . ." and shows a cursor.

```
subahs-MacBook-Air:A3 subahmehrotra$ java server
PICK SAME CIA DETAILS FOR THE SERVER AND THE CLIENT

Do you want Confidentiality? (y/n)
y
Do you want Integrity? (y/n)
n
Do you want Authentication? (y/n)
y
Server is waiting. . . .
█
```

Figure 1: Security properties on the server

A screenshot of a macOS terminal window titled "A3 - java - 80x24". The prompt is "subahs-MacBook-Air:A3 subahmehrotra\$". The user has entered "java client". The output shows the client starting and asking for security preferences: "PICK SAME CIA DETAILS FOR THE SERVER AND THE CLIENT", "Do you want Confidentiality? (y/n)", "Do you want Integrity? (y/n)", and "Do you want Authentication? (y/n)". The user has responded with "y", "n", and "y" respectively. The client then asks "Enter username" and shows a cursor.

```
subahs-MacBook-Air:A3 subahmehrotra$ java client
PICK SAME CIA DETAILS FOR THE SERVER AND THE CLIENT

Do you want Confidentiality? (y/n)
y
Do you want Integrity? (y/n)
n
Do you want Authentication? (y/n)
y
Enter username
█
```

Figure 2: Security properties on the client

Confidentiality

When the confidentiality property is agreed upon by both client and server, the instant messenger encrypts the message from the sender with the *encrypt* function and decrypts the message in the *decrypt* function. The client and server Java files have the same *encrypt* and *decrypt* functions.

To encrypt, the *encrypt* function stores the bytes of the message, creates an Advanced Encryption Standard (AES) cipher, and creates a secret key. The secret key, used to initialize the cipher, is the same for client and server because they share the same *key* variable. After the message is finished encrypting and is stored in the *encryptedData* variable, it is encoded with a Base64 encoding scheme and stored in the *encryptedByteValue* variable. Finally, the *encryptedByteValue* variable is sent to the other party (Figure 3).

```
//encrypt: Encrypt the message
public String encrypt(String data){
    try{
        byte[] dataToSend = data.getBytes();
        Cipher c = null;
        c = Cipher.getInstance(algorithm);
        SecretKeySpec k = new SecretKeySpec(key, algorithm);
        c.init(Cipher.ENCRYPT_MODE, k);
        byte[] encryptedData = "".getBytes();
        encryptedData = c.doFinal(dataToSend);
        Base64.Encoder encoder = Base64.getEncoder();
        byte[] encryptedByteValue = encoder.encode(encryptedData);
        return new String(encryptedByteValue);
    } catch (Exception e) {
    }
    return ("");
}
```

Figure 3: Encrypt function in the code

When the incoming message is obtained by the receiver and confidentiality has been agreed, the *decrypt* function is called. It creates the same AES cipher and secret key as the *encrypt* function because of their shared key. Then, it uses a Base64 decoding

scheme, and finally it decrypts the message. The message will then appear on the receiver's terminal (Figure 4).

If one user agrees to the property and the other does not, the user with the property on will attempt to decrypt files that are not encrypted and output a blank space. The user with the property off will not decipher the message and will output the Base64 encoded message.

```
//decrypt: Decrypt the message
public String decrypt(String data){
    try{
        Base64.Decoder decoder = Base64.getDecoder();
        byte[] encryptedData = decoder.decode(data);
        Cipher c = null;
        c = Cipher.getInstance(algorithm);
        SecretKeySpec k = new SecretKeySpec(key, algorithm);
        c.init(Cipher.DECRYPT_MODE, k);
        byte[] decrypted = null;
        decrypted = c.doFinal(encryptedData);
        return new String(decrypted);
    } catch (Exception e) {
    }
    return ("");
}
```

Figure 4: Decrypt function in the code

Java Libraries Imported for Confidentiality

Below is the list of java libraries imported in both client.java and server.java to code for confidentiality:

- java.security.Provider
- javax.crypto.Cipher
- javax.crypto.spec.SecretKeySpec
- java.security
- java.util.Base64
- java.util.Base64.Encoder
- java.util.Base64.Decoder

Integrity

When both client and server agree to the integrity property, the instant messenger uses the *Add_mac* function, *check_mac* function, and *Remove_mac* function to determine if the message has been tampered with between sending and receiving it. All functions are identical in the client and server Java files (Figure 5).

```
//check_mac: Checks if the message is sent from the client
public static boolean check_mac(String data){
    if (data.contains("!@#!@#$%")){
        return true;
    }
    return false;
}

//Add_mac: Add a message authentication key
public static String Add_mac(String data){
    return "!@#!@#$%"+data;
}

//Remove_mac: Remove the message authentication key
public static String Remove_mac(String data){
    return data.replace("!@#!@#$%", "");
}
```

Figure 5: Various functions used to maintain the integrity of the messages

When the message is being sent, the *Add_mac* function attaches an Message Authentication Code (MAC) to the message. If there was any encryption done to the file, the MAC is added afterwards.

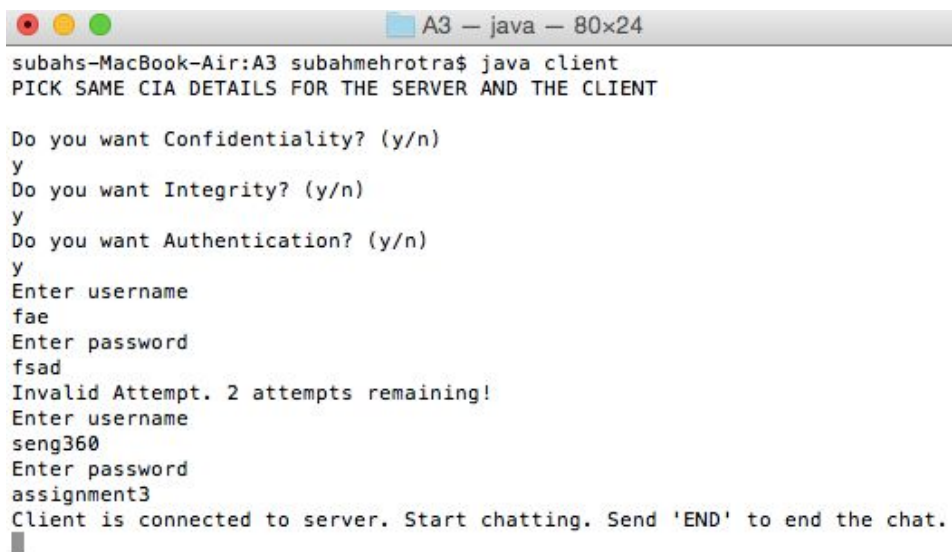
Once the message is received, the *check_mac* function finds whether or not the MAC has been tampered with. If there is tampering, then the output lets the message receiver know the integrity has failed. If there MAC remains untampered, the *Remove_mac* function removes the MAC and the message receiver can now see the message.

If the client and server do not agree on the integrity property setting, the user with the setting one will receive the outputs 'Integrity Failed'. The user with the setting off will have an output of a blank space.

Authentication

When the client agrees to the authentication property, they are prompted to input a username and password. The client has a maximum of three tries to successfully input the username and password, or else the client process exits. After the first failed attempt, there is a countdown to show the remaining attempts available to the client. Executing the client code after exiting, and agreeing to the authentication property, will restart the number of password attempts. Entering the correct username and password will connect the client to the server, starting the instant messenger (Figure 6). All of this happens within the *checkAuthentication* function.

The authentication property does contain limitations. If the server agrees to the authentication property but the client does not, there is no way to stop the client from connecting to the server. Also, if a user can read the client Java file, it is possible for them to find the required username and password.



```
A3 - java - 80x24
subahs-MacBook-Air:A3 subahmehrotra$ java client
PICK SAME CIA DETAILS FOR THE SERVER AND THE CLIENT

Do you want Confidentiality? (y/n)
y
Do you want Integrity? (y/n)
y
Do you want Authentication? (y/n)
y
Enter username
fae
Enter password
fsad
Invalid Attempt. 2 attempts remaining!
Enter username
seng360
Enter password
assignment3
Client is connected to server. Start chatting. Send 'END' to end the chat.
```

Figure 6: Entering username and password on the client

End Session and Reconnect

To end the session between the client and the server, both client and server terminals must enter 'END'. If, after entering 'END' on one terminal, the connection has ended on either client or server, the other, active terminal will still be able to send messages, but those messages will not be received.

If the connection between server and client is lost unexpectedly, the server will wait for a new connection and will show 'Server is waiting. . . .' on the terminal. Once the connection is accepted with a new client, the server and client can communicate again.

The *reconnect* function, shown in Figure 7, is present in `server.java` and stops the server from crashing if the client's connection drops without warning. This can be tested by quitting the present client session (control + C) and starting the client again ('java client').

```
//reconnect: Reconnects the client if the connection is lost
public void reconnect(){
    try {
        t1 = new Thread(this);
        t2 = new Thread(this);
        System.out.println("Server is waiting. . . .");
        socket = serversocket.accept();
        System.out.println("Client connected with Ip " + socket.getInetAddress().getHostAddress());
        t2.start();
        t1.start();
    } catch (Exception e){
    }
}
```

Figure 7: Reconnect function in `server.java`

Compiling Instructions

1. Have a command line or terminal open
2. Compile the server Java file by typing in 'javac server.java'
3. Compile the client Java file by typing in 'javac client.java'

Step-by-Step Instructions

1. Open two console windows (i.e.: Two command lines or two terminals)
2. Follow the compiling instructions
3. In one console window, execute the server by typing in 'java server'
4. Answer the property setting questions for the server
5. In the other window, execute the client by typing in 'java client'
6. Answer the property setting questions for the client (Note: These should be the same answers given to the server)
If Authentication is on, type in username and password (Username: seng360; Password: assignment3)
7. Continue with instant messenger program. Messages will now be sent between client and server.
8. When finished, type 'END' into client window and server window. (Note: until both client and server enter 'END', the one who has not sent it can still send messages)

Deliverable

The submission for this assignment is a zip file of the folder named Assignment3. It contains the following:

- server.java
- server.class
- client.java
- client.class
- technical_details.pdf
- authenticate.txt (contains the username and password)

Conclusion

All three security properties (Confidentiality, Integrity and Authentication) are accomplished for this assignment. The client and server communication is done using java socket and with the help of various threads to keep the client and server communicating with each other until “END” is entered on both sides. The only feature not accomplished for this assignment is the appropriate error message which should come if the client and server do not have the same property settings. It is hard for the server to know what all security properties are selected on the client side before having an established connection with the client.

This project, including all deliverables. is also available at <https://github.com/subahm/Cryptography>.