

SENG 360 Assignment Three – Cryptography **Instant Messaging App**

Phil Denhoff and Jackie Gorman

Technical Details

This application consists of four separate classes: the Messenger, Message, MessagePackage, and CIA classes. The javadoc files for each class can be found in the InstantMessaging directory. These provide the technical details of method headers, parameters, what occurs in the method, and what is returned.

The Message Template

This template is what the Client and Server implement. This template extends the RMI Remote interface, and allows for remote communication.

The MessagePackage Object

This class is what the Server and Client pass to each other to securely transmit information, including Public Keys, Symmetric Keys, signatures, initialization vectors, and the name of the device.

The Messenger Class

Both the Client and Server are instances of the Messenger class. This class implements many of the required methods for allowing client-server communication using RMI.

When the messenger class boolean for authentication is selected, this class:

- Prompts the user for a password
- Hashes the received password using the `hashString(message)` method in the CIA class
- Compares the hashed password with the hashed stored password for that user
- If they match you are authenticated, else the program displays the error “Password does not match” and the program exits

When both the messenger class booleans for confidentiality and integrity are selected, this class:

- Sends messages encrypted using the `encryptSymmetric(message)` method in the CIA class
- Receives messages and decrypts them using the `decryptSymmetric(message)` method in the CIA class
- Signs messages using the `sign(message)` method in the CIA class
- Verifies signed messages using the `verifySignature(message)` in the CIA class

When the messenger class boolean for confidentiality is selected, this class:

- Sends messages encrypted using the `encryptSymmetric(message)` method in the CIA class
- Receives messages and decrypts them using the `decryptSymmetric(message)` method in the CIA class

When the messenger class boolean for integrity is selected, this class:

- Signs messages using the sign(message) method in the CIA class
- Verifies signed messages using the verifySignature(message) in the CIA class

The CIA Class

The CIA class is used to implement the selected confidentiality, integrity, and authentication options. When a new CIA object is invoked, the constructor takes a boolean for the confidentiality and integrity options. If either confidentiality or integrity is selected, asymmetric keys are generated. Asymmetric encryption and decryption uses RSA with key sizes of 2048 bits.

To implement confidentiality this class is used by the Messenger class to:

- Generate a symmetric key for AES CBC with PKCS5 padding
- Encrypt and decrypt using an established symmetric key
- Encrypt using the other person's public key
- Decrypt using the corresponding private key

To implement integrity this class is used by the Messenger class to:

- Sign messages using a private key
- Verify the signed message using the corresponding public key (the other person's)

To implement authentication this class is used by the Messenger class to:

- Hash a string password to be compared to the stored hashed password for the user

Vulnerabilities

During initialization, our method of implementation is vulnerable to a man in the middle attack when each user's public key is being passed to the other user. If we were truly implementing this project, public keys would be stored in an access controlled location that they could be retrieved from or they would otherwise be previously securely passed/exchanged.

Additionally, the password used for authentication is stored in plaintext, rather than stored as a hash. The password would be stored as a hash if this application were truly implemented. In this case, every user's password is also "password." This should be resolved by creating accounts with usernames and passwords.

How to Compile

This program can be compiled using the following command from within the correct directory:

```
javac *.java
```

How to Use It

You must add this snippet:

```
grant {  
    permission java.security.AllPermission;  
};
```

To the system policy file at

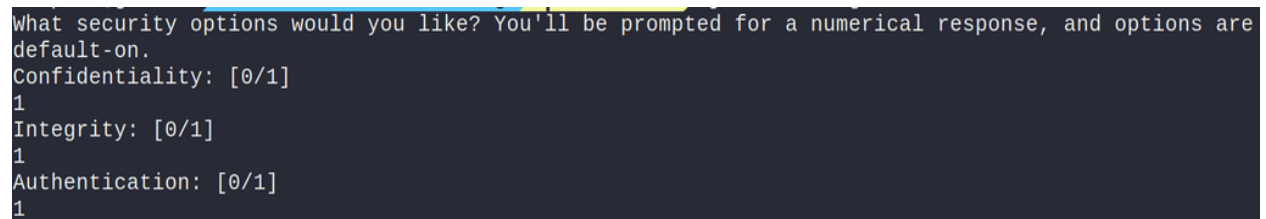
```
java.home/lib/security/java.policy  (Solaris/Linux)  
java.home\lib\security\java.policy  (Windows)
```

in order for this program to work.

Once that has been added and the program is compiled, the program can be initialized using the following command:

```
java Messenger
```

The program will then prompt the user for the options of confidentiality, integrity, and authentication. The user responds with a 0 or 1 for no or yes, respectively, to each presented parameter. This can be seen in the screenshot below. In this case, the user has selected all of confidentiality, integrity, and authentication.



```
What security options would you like? You'll be prompted for a numerical response, and options are  
default-on.  
Confidentiality: [0/1]  
1  
Integrity: [0/1]  
1  
Authentication: [0/1]  
1
```

If the user selects authentication, they will be prompted to enter a password (for this application the password is always password).

If the password does not match the program will output "Password does not match" and the program quits. Else, if the password matches the user's stored password, the user will be prompted to enter whether they are a client or the server. To select client, enter a C, to select server, enter a S. The case of the C or S does not matter. If you select client before a server has been started in another window, it will error and say "Error: No server started." Therefore, the first device to connect must take the identity of the server. The error message can be seen below.

```
What security options would you like? You'll be prompted for a numerical response, and options are
default-on.
Confidentiality: [0/1]
1
Integrity: [0/1]
1
Authentication: [0/1]
1
Enter password:
password
Are you a client or the server? [C/S]
c
ERROR: No server started.
```

If the device takes the identity of the server, they will be prompted to enter the server name, which may be any unicode character. This should then return the message

```
>>> DEVICE READY
>>> NAME: <name>
```

The server is now ready for use. This can be seen below.

```
What security options would you like? You'll be prompted for a numerical response, and options are
default-on.
Confidentiality: [0/1]
1
Integrity: [0/1]
1
Authentication: [0/1]
1
Enter password:
password
Are you a client or the server? [C/S]
s
(SERVER MODE) Enter the server name:
seng-360-chat
>>> DEVICE READY
>>> NAME: seng-360-chat
```

Once a server has been started, then the user can become a client, or they may select server. If they select server, no connection will be made until a client initializes and selects the server. If they select client, they will receive a list of available servers and are prompted to select the server name that they wish to connect to. If the client selected the same security options as the selected server, this will return the message

```
>>> DEVICE READY
>>> NAME: client
<selectedServer> connected to you!
```

This can be seen in the next image below.

```

What security options would you like? You'll be prompted for a numerical response, and options are
default-on.
Confidentiality: [0/1]
1
Integrity: [0/1]
1
Authentication: [0/1]
1
Enter password:
password
Are you a client or the server? [C/S]
c
Available servers:
    seng-360-chat
(CLIENT MODE) Enter the server name:
seng-360-chat
>>> DEVICE READY
>>> NAME: client
seng-360-chat connected to you!

```

The server can only connect to one client at once. If a second client initializes connection with a server that is already connected to a client, the server will disconnect from the first client and only respond to the second.

Assuming the first user takes the identity of the server and the second takes the identity of the client and they select the same security options, a successful connection will be established. The two parties can now communicate back and forth by typing messages, using enter to send the message to the other party. The other party's messages are displayed <other_party_name>: <message>. An example of this is shown below.

This is the server's window:

```

What security options would you like? You'll be prompted for a numerical response, and options are
default-on.
Confidentiality: [0/1]
1
Integrity: [0/1]
1
Authentication: [0/1]
1
Enter password:
password
Are you a client or the server? [C/S]
s
(SERVER MODE) Enter the server name:
seng-360-chat
>>> DEVICE READY
>>> NAME: seng-360-chat
client: Jens is my favourite Prof!
And his TAs are awesome too!
client: glorious secure-msg for the win

```

This is the corresponding client's window:

```
What security options would you like? You'll be prompted for a numerical response, and options are
default-on.
Confidentiality: [0/1]
1
Integrity: [0/1]
1
Authentication: [0/1]
1
Enter password:
password
Are you a client or the server? [C/S]
c
Available servers:
    seng-360-chat
(CLIENT MODE) Enter the server name:
seng-360-chat
>>> DEVICE READY
>>> NAME: client
seng-360-chat connected to you!
Jens is my favourite Prof!
seng-360-chat: And his TAs are awesome too!
glorious secure-msg for the win
```