Rafael Solorzano

V00838235

SENG360 Assignment 3 Docs

Compiling

In order to compile the program the following code can be run:

Javac *.java

Running and Using

To run the program the following steps must be taken:

1. Run the rmiregistry server by running:
    a. start rmiregistry
2. Run the server by running:
    a. java ServerOperation
3. Run the client by running:
    a. java ClientOperation

Additionally, if options were to be selected they should be specified as arguments to ServerOperation first and then to Client Operation. The arguments can be provided in any order. If no arguments are supplied the application will not use any security. The supported options are:

- c: provides confidentiality
- i: provides integrity
- a: provides authentication
    - Server login: user (admin), pass (seng360)
    - Client login: user (client), pass (iAmAClient)

An example of running the server with all the options and subsequently the client is shown below:

```
E:\Rafa\Documents\GitHub\SENG360\SENG360Asn3\src>java ServerOperation a i c
Authentication Set
Integrity Set
Confidentiality Set
```

```
E:\Rafa\Documents\GitHub\SENG360\SENG360Asn3\src>java ClientOperation a c i
Connection established with features:
Confidentiality
Integrity
Authentication
```

Technical Details

After they have been started the first message can be sent from the client, to which the server can respond and so on. Note, that a limitation of the application is that the client must send the first message and they are only able to respond one message a time.

The program supports Confidentiality, Integrity, and Authentication, this are explained below. Note that the below explanations are shown from the Server perspective, as the Client perspective is the same.

Confidentiality

When only confidentiality is specified, the server and client both generate public/private key pairs. A code example of how this is done is shown below:

```
private static void generateKeys() throws NoSuchAlgorithmException{
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(2048);

    KeyPair pair = keyGen.generateKeyPair();
    privateKey = pair.getPrivate();
    publicKey = pair.getPublic();
}
```

The private key is stored as private global variable, and the public key is stored as a public global variable to be able to encrypt using the public key from the client or server side.

After the keys have been generated the getPublicKey() function is called from the server/client to get the public key for encrypting the key for the message.

Once the key is obtained, the generation of the key, its encryption, and the message encryption is done.

This is shown in order in the example code below:

```
private static SecretKey generateKey() throws NoSuchAlgorithmException{
    KeyGenerator keygen = KeyGenerator.getInstance("AES");
    keygen.init(128);
    SecretKey aesKey = keygen.generateKey();
    return aesKey;
}
```

Generate Key

```
public static byte[] encryptKey(SecretKey key) throws InvalidKeyException, NoSuchAlgorithmException, I
    String ciphertext = Base64.getEncoder().encodeToString(key.getEncoded());
    Cipher encryption = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    encryption.init(Cipher.PUBLIC_KEY, clientPublicKey);
    byte[] encryptedKey = encryption.doFinal(ciphertext.getBytes());
    return encryptedKey;
}
```

Encrypt Key

```
private static byte[] encryptMessage(String text, SecretKey aesKey) throws

    Cipher aesCipher = Cipher.getInstance("AES");

    // Initialize the cipher for encryption
    aesCipher.init(Cipher.ENCRYPT_MODE, aesKey);

    // Our cleartext
    byte[] cleartext = text.getBytes();

    // Encrypt the cleartext
    byte[] ciphertext = aesCipher.doFinal(cleartext);
    return ciphertext;
}
```

Encrypt Message

Once the message is encrypted the encrypted key and the encrypted text are sent to the receiving party. Which in turn will decrypt it, print it and be able to send a response. This process is shown below:

```
public void sendMessageServerEncrypted(byte[] encryptedKey, byte[] encryptedText) throws
    Cipher aesCipher = Cipher.getInstance("AES");

    SecretKey originalKey = decryptKey(encryptedKey);

    aesCipher.init(Cipher.DECRYPT_MODE, originalKey);
    // Decrypt the ciphertext
    byte[] cleartext1 = aesCipher.doFinal(encryptedText);
    String decryptedText = new String(cleartext1);
    System.err.println("Client: "+ decryptedText);


    Scanner sc = new Scanner(System.in);
    String txt = sc.nextLine();
    sc.close();


    SecretKey key = generateKey();
    byte[] encodedKey = encryptKey(key);
    byte[] cipherText = encryptMessage(txt,key);


    client.sendMessageClientEncrypted(encodedKey, cipherText);

}
```

Decrypting the message

Once, the message is decrypted, shown, and a response is send, the process will repeat.

Integrity

When only Integrity is specified, the server and client both generate MAC key and MAC data on the input. This MAC key and MAC data are sent with each message.

The Mac Key and Bytes are stored in global variables and are generated as shown below:

```
private static void generateMACKey() throws NoSuchAlgorithmException{
    KeyGenerator keygen = KeyGenerator.getInstance("HmacMD5");
    SecretKey macKeyGen = keygen.generateKey();
    macKey = macKeyGen;
    byte[] keyBytes = macKey.getEncoded();
    macKeyBytes = keyBytes;
}
```

Once we have the MacKey and Bytes, a MacData is generated for each message that will be communicated between the clients. This is done as shown below:

```
private static byte[] generateMACData(String txt) throws
    Mac mac = Mac.getInstance("HmacMD5");
    mac.init(macKey);
    mac.update(txt.getBytes());
    byte[] macData = mac.doFinal();
    mac.reset();
    return macData;
}
```

Once all this is done, the plain text message is passed along with the macKeyBytes and the Mac Data for the plain text. This is then integrity checked as follows:

```
public void sendMessageServerIntegrity(String txt, byte[] macKey, byte[] macData) throws
    SecretKeySpec spec = new SecretKeySpec(macKey, "HmacMD5");
    Mac mac = Mac.getInstance("HmacMd5");

    mac.init(spec);
    mac.update(txt.getBytes());

    byte [] macCode = mac.doFinal();

    if (macCode.length != macData.length){
        System.out.println("ERROR: Integrity check failed, possible intercept");
    } else if (!Arrays.equals(macCode, macData)){
        System.out.println ("ERROR: Integrity check failed, possible intercept");
    } else {
        System.out.println("Client: " + txt);
    }


    Scanner sc = new Scanner(System.in);
    String toSend = sc.nextLine();
    sc.close();

    generateMACKey();
    client.sendMessageClientIntegrity(toSend, macKeyBytes, generateMACData(toSend));

}
```

Integrity Checking

Once the message has been validated, a response will be sent the exact same way and the process will repeat.

Integrity and Confidentiality

When both Integrity and Confidentiality are requested, a similar process to two previous is done. The difference consists in that this time both a public/private key pair and MAC key/data is generated. Additionally, the plain text is sent encrypted and will be integrity checked after it had been decrypted.

The code that handles Integrity and Confidentiality simultaneously is shown below:

```
public void sendMessageServerEncryptedIntegrity(byte[] encryptedKey, byte[] encryptedText, byte[] macKey, byte[] macData)
    /* First decrypt the text to plaintext */
    Cipher aesCipher = Cipher.getInstance("AES");

    SecretKey originalKey = decryptKey(encryptedKey);

    aesCipher.init(Cipher.DECRYPT_MODE, originalKey);
    // Decrypt the ciphertext
    byte[] cleartext1 = aesCipher.doFinal(encryptedText);
    String decryptedText = new String(cleartext1);

    /*Integrity check the decrypted text */
    SecretKeySpec spec = new SecretKeySpec(macKey, "HmacMD5");
    Mac mac = Mac.getInstance("HmacMd5");

    mac.init(spec);
    mac.update(decryptedText.getBytes());

    byte [] macCode = mac.doFinal();

    if (macCode.length != macData.length){
        System.out.println("ERROR: Integrity check failed, possible intercept");
    } else if (!Arrays.equals(macCode, macData)){
        System.out.println ("ERROR: Integrity check failed, possible intercept");
    } else {
        System.out.println("Client: " + decryptedText);
    }

    Scanner sc = new Scanner(System.in);
    String toSend = sc.nextLine();
    sc.close();

    SecretKey key = generateKey();
    byte[] encodedKey = encryptKey(key);
    byte[] cipherText = encryptMessage(toSend,key);

    generateMACKey();

    client.sendMessageClientEncryptedIntegrity(encodedKey, cipherText, macKeyBytes, generateMACData(toSend));
}
```

Authentication

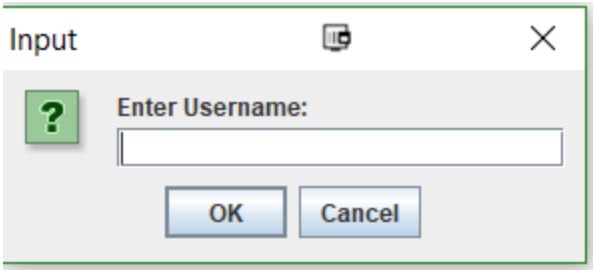
When authentication is specified, the server must first login providing a username and password.

To simulate storage of the logins in a database table, a txt file with the user and hashed password for server (server_auth.txt) and client (client_auth.txt) are used.

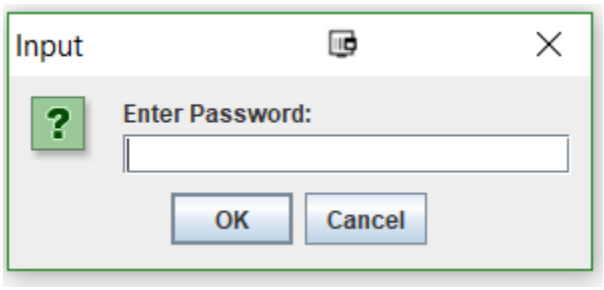
Server Login

When authentication is requested, the server will be the first to login. This will be done by providing a user and password prompt that will close the process after three failed attempts.

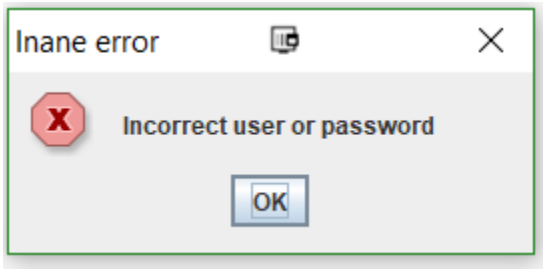
The login screens are shown below:



Username prompt



Password prompt



Incorrect prompt

```
E:\Rafa\Documents\GitHub\SENG360\SENG360Asn3\src>java ServerOperation a
Authentication Set
Too many incorrect tries... Exiting
```

Three failed attempts

After a success login the server will just proceed as normal and will enforce any of the other security properties specified.

To verify the user and password, the user and password from the “database” are read and compared against. For the password, the provided plaintext is hashed and compared against the hashed password stored in the database. This is done by the two functions (may be Server or Client) shown below:

```
private static int validateLogin(String usr, String pwd) throws NoSu
    String correctLogin[] = getLoginsServer();
    String correctUsr = correctLogin[0];
    String hashedCorrectPwd = correctLogin[1];


    MessageDigest sha = MessageDigest.getInstance("SHA-256");
    sha.update(pwd.getBytes());
    byte[] bytes = sha.digest();
    String hashedInput = Base64.getEncoder().encodeToString(bytes);


    if (usr.equals(correctUsr)){
        System.out.println("User matches");
        if (hashedInput.equals(hashedCorrectPwd)){
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}
```

Validate Login functions, calls function below for the pair logins.

```
private static String[] getLoginsServer() throws IOException{
    BufferedReader br;
    br = new BufferedReader(new FileReader("server_auth.txt"));
    String line = br.readLine();
    String[] parts = line.split(" ");
    br.close();
    return parts;

}
```

Get the pair login from the “database”

Password Hasher

To test hashes, and view them easily with different inputs a helper Password Hasher class was used. This class takes as an argument a single text and will return a hashed version of the plain text. This function was used to generate the hashes for the client and server passwords. The code is shown below:

```java
public class PasswordHasher {

    public static void main(String[] args) throws NoSuchAlgorithmExc
        String toHash = new String();
        if (args.length == 0 || args.length > 1){
            System.exit(0);
        } else {
            toHash = args[0];
        }
        System.out.println(toHash);
        MessageDigest sha = MessageDigest.getInstance("SHA-256");
        sha.reset();
        byte[] bytes = sha.digest(toHash.getBytes("UTF-8"));

        String hashed = Base64.getEncoder().encodeToString(bytes);

        System.out.println(hashed);

    }

}
```