Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

Computer security, 7.5 credits
Semester Year e.g. Autumn Semester 2021
Faculty of Natural Science

Lab 1&2

Sandra Kaljula

Content

Introduction

This report is about lab 1 and 2 in the computer security course. The source code is in Appendix A.

Research questions:

- Why is it not a good idea to simply encrypt the plaintext with the receiver's public key? Why bother to generate Key1, IV, and encrypt them?

- Suppose the receiver (i.e. you) does not share any secret with the sender before she/he receives the encrypted keys in ciphertext.enc (i.e. the ciphertext + the encrypted symmetric keys). Does a verified correct message authentication code (MAC) (e.g. the one received by applying HmacMD5 in this exercise) authenticate the sender or can we trust the origin of the message in this case? Why or why not? (Note that we are assuming that digital signature is not used)

Method

First, the notes made during the lectures were read through together with the lecture slides.

Second, all the code examples from the lectures were imported to IntelliJ which is the environment that is used in these labs. The code was coded to be fully functional and runnable on my device and commented to give a better understanding to it.

Third, the lab1 and 2 were initialized by downloading the files needed for the labs and a pseudocode was written with the help of the instructions and diagrams in the instruction's sheets.

Fourth, the coding could be started and there were some problems that were encountered during this, which were solved by further research using the internet.

Finally, the code was analyzed, and the research questions were answered.

Results

Figure 1 shows the plaintext message retrieved from ciphertext using the keys.

```
Slip inside the eye of your mind
Don't you know you might find
A better place to play
You said that you'd never been
But all the things that you've seen
Will slowly fade away

So I'll start a revolution from my bed
Cos you said the brains I had went to my head
Step outside. Summertime's in bloom
Stand up beside the fireplace
Take that look from off your face
You ain't ever gonna burn my heart out

So Sally can wait, she knows its too late as we're walking on by
Her soul slides away, but don't look back in anger I heard you say

Take me to the place where you go
Where nobody knows if it's night or day
Please don't put your life in the hands
Of a Rock n Roll band
Who'll throw it all away

Gonna start a revolution from my bed
Cos you said the brains I had went to my head
Step outside the summertime's in bloom
Stand up beside the fireplace
Take that look from off your face
Cos you ain't ever gonna burn my heart out

So Sally can wait, she knows its too late as she's walking on by
Her soul slides away, but don't look back in anger I heard you say

So Sally can wait, she knows its too late as we're walking on by
Her soul slides away, but don't look back in anger I heard you say

So Sally can wait, she knows its too late as she's walking on by
My soul slides away, but don't look back in anger
Don't look back in anger
I heard you say
At least not today
```

Figure 1 Plaintext from the ciphertext.

Why is it not a good idea to simply encrypt the plaintext with the receiver’s public key?

The asymmetric encryption guarantees the confidentiality of the message. However, it does not guarantee the integrity of the message since the public-key encryption allows anyone that is eavesdropping to modify the message, even if the contents are not understood. To make it more secure it should be used together with a digital signature that uses symmetric key to guarantee the integrity of the message. Another alternative would be using a secret (symmetric) key together with the message authentication code. Using the receiver's public key only protects the data confidentiality, so that only the receiver and no one else can decrypt it with their private key.

Why bother to generate Key1, IV, and encrypt them?

It is encrypted with the receiver's public key to distribute the (symmetric)secret key to the sender, which then together with the IV can decrypt the next cipher block chains to plaintext. If the secret key and IV for this was sent in plaintext over to the sender, it would cause a big security risk. In case of a man in the middle attack the attacker would have key and IV to all the chains. So, they can decrypt them and read them themselves even in the future. They could also change the contents if they have all the keys and send it to the receiver pretending to be the sender.

The test for MAC(Message Authentication Code) and Digital signatures showed the following results seen in figure 2.

```
MAC2 is true
First signature is true and the second signature is false
```

Figure 2 Results correct MAC and digital signature.

Suppose the receiver (i.e. you) does not share any secret with the sender before she/he receives the encrypted keys in ciphertext.enc (i.e. the ciphertext + the encrypted symmetric keys). Does a verified correct message authentication code (MAC) (e.g. the one received by applying HmacMD5 in this exercise) authenticate the sender or can we trust the origin of the message in this case? Why or why not? (Note that we are assuming that digital signature is not used)

Yes, since we shared the secret key that is needed for message authentication code through using the private key that was stored in the password protected keystore. The key in the keystore is stored with the passwords that are written in plaintext in the pdf file, which could be a problem if the files were downloaded from a non-secure website. But since canvas is used, which is a very secure website I do not have to worry.

But if the website was not protected with any certificates or protocols and for example used http that is not secure at all. The man in the middle or even the website themselves could change the contents of this lab to be a threat to the downloader’s computer. The level of this threat really depends on what the user has in the computer. For example, if a keylogger is installed together with the documents the attacker might get all the passwords and information about the victim.

Discussion

I think these labs were very good to get an insight to encryption methods and computer security as a whole. I encountered some problems when using byte arrays and translating the numbers between different bases. But at the end I got it all figured out. It was a lot of new classes and methods that I had personally not used before and I am glad that I can now implement these in my future projects.

Appendix A

```java
package lab1;

//import org.graalvm.compiler.java.LambdaUtils; did not work

import javax.crypto.*;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.util.Locale;


/*       0. Get the private key to encrypt the other keys from the lab1Store
       1. EncKey 1: 128-bytes (1024 bits) of a symmetric key encrypted with RSA using the public key "lab1EncKeys"
       2. EncIV value: 128-byte of encrypted IV-value to be used in the decryption of the data
       3. EncHmacMD5 key: 128-byte containing encrypted key for a HmacMD5
       4. Ciphertext: Finally we have the encrypted data. The encryption is made with AES in CBC mode, using PKCS5 padding.*/
/*
       To get the keys and IV to be used to decrypt we need a private key.
       The key is stored in the keystore "lab1Store". The password to access lab1Store is "lab1StorePass". The
       alias for the private key is "lab1EncKeys" (bad name as already used to name its
       associated public key), and the password is "lab1KeyPass".*/

public class Main {
   public static void main(String[] args) throws IOException, BadPaddingException, IllegalBlockSizeException, NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException,
```

```
InvalidAlgorithmParameterException,
CertificateException, SignatureException {
        // Get the encrypted file
        File f = new File("src/lab1/ciphertext.enc");
        FileInputStream fis = new FileInputStream(f);

        //Get the private key from lab1Store  LAB1
encKey
        char [] storePassword =
"lab1StorePass".toCharArray();
        char [] keyPassword =
"lab1KeyPass".toCharArray();
        KeyStoreClass keyStore = new KeyStoreClass();
        PrivateKey Lab1EncKey =
keyStore.loadKey("src/lab1/lab1Store", storePassword,
"lab1EncKeys", keyPassword);
        //loadKey parameters:            loadKey(String
storeFilename, char[] storePassword, String alias,
char[] keyPassword)


        //Find keys 1, 2 and IV (decrypt with the
private key)
        //Key1 from EncKey1
        byte[] encKey1 = new byte[128];
        fis.read(encKey1); // Reads the encKey1.length
bytes of data
        Cipher rsaDec=Cipher.getInstance("RSA");
        rsaDec.init(Cipher.DECRYPT_MODE, Lab1EncKey);
        byte[] key1 = rsaDec.doFinal(encKey1);

        //IV from EncIV
        byte[] encIV = new byte[128];
        fis.read(encIV);
        byte[] IV = rsaDec.doFinal(encIV);

        //Key2 from encKey2 (EncHmacMD5)
        byte[] encKey2 = new byte[128];
        fis.read(encKey2);
        byte [] key2 = rsaDec.doFinal(encKey2);

        //Ciphertext
        byte[] ciphertext = fis.readAllBytes();
        fis.close();


        //Decrypt to plaintext byte array
        Cipher decoder =
Cipher.getInstance("AES/CBC/PKCS5Padding");
```

```
        SecretKeySpec keyOne = new SecretKeySpec(key1,
"AES");
        IvParameterSpec ivSpec = new
IvParameterSpec(IV);
        decoder.init(Cipher.DECRYPT_MODE, keyOne,
ivSpec);
        byte[] message = decoder.doFinal(ciphertext);

        String[] plain = new String[]{new
String(message, StandardCharsets.UTF_8)};
        String plaintext = plain[0];
        System.out.println(plaintext);

        //Verify MAC
        verifyMAC(message, key2);

        //Verify the digital signature
        verifyDigitalSignature(message);

    }

   public static void verifyMAC(byte[] message,
byte[]key2) throws IOException,
NoSuchAlgorithmException, InvalidKeyException {
        //comparing MAC
        File file1 = new
File("src/lab1/ciphertext.mac1.txt");
        FileInputStream fis1 = new
FileInputStream(file1);
        byte [] MAC1bytes = fis1.readAllBytes();
        char[] MAC1chars = new String(MAC1bytes, "UTF-
8").toCharArray();
        String MAC1value = String.valueOf(MAC1chars);

        File file2 = new
File("src/lab1/ciphertext.mac2.txt");
        FileInputStream fis2 = new
FileInputStream(file2);
        byte [] MAC2bytes = fis2.readAllBytes();
        char[] MAC2chars = new String(MAC2bytes, "UTF-
8").toCharArray();
        String MAC2value = String.valueOf(MAC2chars);


        Mac mac = Mac.getInstance("HmacMD5");
        SecretKeySpec keyTwo = new SecretKeySpec(key2,
"AES");
        mac.init(keyTwo);
        byte [] MACbytes = mac.doFinal(message);
```

```
        StringBuilder stringBuilder = new
StringBuilder();
        for (byte bytes: MACbytes){
            System.out.println("bytes"+bytes);
            stringBuilder.append(String.format("%02X",
bytes).toLowerCase(Locale.ROOT));
            System.out.println(String.format("%02X",
bytes));
        }
        String MACvalue = stringBuilder.toString();
```

```
        if (MACvalue.equals(MAC1value)){
            System.out.println("MAC1 is true");
        }
        if (MACvalue.equals(MAC2value)){
            System.out.println("MAC2 is true");
        }
    }
```

```
    public static void verifyDigitalSignature(byte[]
message) throws IOException, CertificateException,
NoSuchAlgorithmException, InvalidKeyException,
SignatureException {
        //Verify the digital signature
        //Source:
https://www.tabnine.com/code/java/methods/java.security
.cert.CertificateFactory/getInstance
        FileInputStream readPuKey = new
FileInputStream("src/lab1/lab1Sign.cert");
        CertificateFactory cf =
CertificateFactory.getInstance("X.509");
        X509Certificate certificate =
(X509Certificate)cf.generateCertificate(readPuKey);
        //Certificate certificate = (Certificate)
cf.generateCertificate(readPuKey);
        //PublicKey puKey = certificate.getPublicKey();
        PublicKey puKey = certificate.getPublicKey();
```

```
        //SHA1withRSA
        FileInputStream signature1 = new
FileInputStream("src/lab1/ciphertext.enc.sig1");
        byte[] signatures1 = signature1.readAllBytes();
```

```
        FileInputStream signature2 = new
FileInputStream("src/lab1/ciphertext.enc.sig2");
        byte[] signatures2 = signature2.readAllBytes();
```

```
        Signature myVerify =
Signature.getInstance("SHA1withRSA");
        myVerify.initVerify(puKey);
```

```
        myVerify.update(message);
        Boolean firstSignature =
myVerify.verify(signatures1);
        myVerify.update(message);
        Boolean secondSignature =
myVerify.verify(signatures2);
        System.out.println("First signature is
"+firstSignature+" and the second signature is
"+secondSignature+"");
    }
}
```

The keystore class was borrowed from the lecture slides:

```
package lab1;
```

```
import javax.crypto.SecretKey;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.KeyStore;
import java.security.PrivateKey;
```

```
public class KeyStoreClass { //Example: Create KeyStore
   public void createKeystore(String filename, char[]
password) {
        try {
            KeyStore myStore =
KeyStore.getInstance("JCEKS");
            myStore.load(null, null); //first time no
file or password
            FileOutputStream storeFile = new
FileOutputStream(filename);
            myStore.store(storeFile, password);
            storeFile.close();
        } catch (Exception e) {
            System.out.println("Next time I would maybe
throw the exception instead");
            e.printStackTrace();
        }
    }
```

```
   public void storeKey(String storeFilename, char[]
//Example: Store a key
            storePassword, SecretKey key, String alias,
char[] keyPassword) {
```

```
        try {
            KeyStore myStore =
KeyStore.getInstance("JCEKS");
            FileInputStream loadFile = new
FileInputStream(storeFilename);
            myStore.load(loadFile, storePassword);
//filename and password that protects the keystore
            loadFile.close();
            KeyStore.SecretKeyEntry skEntry = new
KeyStore.SecretKeyEntry(key);
            myStore.setEntry(alias, skEntry, new
                    KeyStore.PasswordProtection(keyPassw
ord));
            FileOutputStream storeFile = new
FileOutputStream(storeFilename);
            myStore.store(storeFile, storePassword);
            storeFile.close();
        }catch (Exception e){
            System.out.println(e);
        }
    }

   public PrivateKey loadKey(String storeFilename,
char[]
            storePassword, String alias, char[]
keyPassword) { //Example: Load a key
        try {
            //start by loading keystore
            KeyStore myStore =
KeyStore.getInstance("JCEKS");
            FileInputStream loadFile = new
FileInputStream(storeFilename);
            myStore.load(loadFile, storePassword);
//filename and password that protects the keystore
            loadFile.close();
            //load key
            PrivateKey theKey = (PrivateKey)
myStore.getKey(alias, keyPassword);
            return theKey;
        }catch(Exception e){
            System.out.println(e);
            return null;
        }
    }
}
```