Security Lab – Cryptography in Java

VMware

- You will work with the **Ubuntu image**, which you should start in networking-mode **Nat**.

1 Introduction

In software development, there's often the requirement to perform cryptographic operations. For instance, consider a program that encrypts and decrypts data or a server application that uses the HTTPS protocol (HTTP over SSL/TLS).

In these situations, it's not reasonable to implement your own cipher or protocol. Instead, you should use available, well-established libraries and focus on using them in a secure way. In this lab, you'll use the cryptographic functions offered by Java to implement a program. The goal is that you get familiar with these functions and can apply them correctly.

2 Basics: Java Cryptography Architecture

The Java Cryptography Architecture[1] (JCA) is a framework to use cryptographic functions in Java. JCA offers various functions, including symmetric and asymmetric block and stream cipher, key generators, hash functions, message authentication codes (MAC), signatures, and certificates. Other security libraries of Java are often based on JCA, e.g. JSSE (for SSL/TLS) and JGSS (for Kerberos), but in this lab the focus is on JCA.

2.1 Cryptographic Service Provider

The Java Cryptography Architecture uses a provider architecture, which means the actual implementation of the cryptographic functions are offered by so-called Cryptographic Service Providers (CSP). If the user wants to use a specific cryptographic function, he requests it from the JCA class, which returns one of the registered implementations provided by a CSP. This means that a function to e.g. produce a SHA-1 hash can actually be provided by different providers (and different companies). Depending on the requirements, one can then choose e.g. a particularly fast or a certified implementation. Java itself includes several CSPs[2]. Today, these are integrated parts of the Java SE, but before Java version 1.4, they had to be additionally installed as so-called Java Cryptography Extensions (JCE). The provider names of the integrated CSPs are – depending on the cryptographic function – for instance *SUN* (e.g. for random number generators), *SunJCE* (for several encryption algorithms), and others. In addition, there are some third-party CSPs, among the most popular ones is the Open Source Bouncy Castle Crypto API[3].

2.2 Basic Usage

To use a cryptographic function in an application, it is requested using a static method of the corresponding factory class of the JCA. For instance, to get an object to compute SHA-1 hashes, this is done as follows:

```
MessageDigest md = MessageDigest.getInstance("SHA1");
```

The method returns a `MessageDigest` object to compute SHA-1 hashes from one of the installed CSPs – if at least one of them provides the function. If multiple installed CSPs support the function, the one with the highest priority is used. Today, this is typically one of the providers that are included in Java per default, which is also reasonable in most cases. On the image with which you are working,

---

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html

[3] http://www.bouncycastle.org

there are no extra CSPs installed besides the ones included in Java SE. Assuming you want to explicitly specify the CSP provider to be used, you must use a second variant of the static method. In the case of the Bouncy Castle CSP (which would have to be installed first), this would look as follows:

```
MessageDigest md = MessageDigest.getInstance("SHA1", "BC");
```

For every cryptographic function, there's a corresponding factory class. For this lab, the following are relevant:

- `SecureRandom`: Creates cryptographically strong random numbers.
- `Cipher`: Used to encrypt data symmetrically or asymmetrically.
- `MAC`: Serves to compute a HMAC.
- `KeyGenerator`: Generates symmetric `SecretKey`s.
- `CertificateFactory`: Converts certificates or CRLs in X.509 format to `Certificate`s, `CRL`s or `CertPath`s.
- `AlgorithmParameters`: Used to define additional parameters, that must be used in addition to the key with some algorithms, e.g. initialization vectors (IV).
- `AlgorithmParameterGenerator`: Creates `AlgorithmParameters` objects for a specific algorithm.

2.3 Classes

In the following, the classes listed above are described in detail. Addition information can be found in the Java API Specifications[4].

2.3.1 SecureRandom class

`SecureRandom` generates cryptographically strong random number. An algorithm that is often used has the name `SHA1PRNG`, which corresponds to a pseudo random number generator that is based on the hash function SHA-1. Creating a corresponding object works as follows:

```
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
```

You can also use the following line to create a `SecureRandom` object that uses the default algorithm, which is `SHA1PRNG`. The result is the same as above.

```
SecureRandom random = new SecureRandom();
```

Random numbers are generated using the function `nextBytes()`. The following line generates 20 random bytes and stores them in the array `bytes`:

```
byte bytes[] = new byte[20];
random.nextBytes(bytes);
```

2.3.2 Cipher class

A `Cipher` is used to encrypt data with an arbitrary algorithm. `Cipher` supports different symmetric and asymmetric algorithms and different padding schemes. The combinations that are supported by the providers that are included in Java per default are described online[5]. These combinations are named transformations and have the following form:

```
"Algorithm/Mode/Padding"
```

For instance, the following must be used for a DES cipher in CBC mode und PKCS5 padding (the method how the plaintext is increased to a multiple of the block length:

---

[4] http://docs.oracle.com/javase/7/docs/api

[5] http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html

```
Cipher c1 = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

Alternatively, one can also specify the algorithm name only. In that case, default values – depending on the used cipher – are used for mode and padding:

```
Cipher c2 = Cipher.getInstance("DES");
```

A `Cipher` can be used for different operations. Most relevant are `ENCRYPT_MODE` and `DECRYPT_MODE`. To use a `Cipher`, it must first be initialized using `init()`. The mode and a key (or a certificate in the case of asymmetric encryption) must be specified as parameters. Details about the key parameter (`key`) follow in section 2.3.4.

```
c1.init(Cipher.ENCRYPT_MODE, key);
```

If a cipher uses CBC mode, an initialization vector (IV) must be specified as well. This can be done when initializing the cipher by using a third parameter, which is an object of the class `AlgorithmParameter`. In this case, initialization of the cipher works as follows:

```
c1.init(Cipher.ENCRYPT_MODE, key, algParam);
```

Details about using `AlgorithmParameter` follow below in section 2.3.5.

After having initialized the `Cipher` object, it can be used to directly encrypt or decrypt data (stored in a byte array) using `doFinal()`. For instance, the following line encrypts the entire byte array `message1` and writes the ciphertext to `ciphertext`:

```
byte[] ciphertext = c1.doFinal(message1);
```

In the case of a block cipher, this includes correct padding of the final plaintext block.

Alternatively, it is also possible to encrypt step-by-step by calling the method `update()` repeatedly. With a block cipher, only complete blocks are encrypted, the rest remains "within the `Cipher` object" and is processed during the next call of `update`. With a stream cipher, all bytes are processed. With a block cipher, the last block must be processed using the `doFinal` method. `doFinal` always processes all remaining data in the `Cipher` object and makes sure the final plaintext block is padded correctly. As an example, the following three lines show twice a call of the `update` method and a necessary final call of `doFinal`.

```
byte[] ciphertext = c1.update(message2a);
ciphertext = c1.update(message2b);
ciphertext = c1.doFinal();
```

Decrypting basically works the same and in this case, `doFinal` removes the padding from the last plaintext block after decryption.

If only the first `n` bytes in a byte array should be passed to the `update` method, this can also be done:

```
c1.update(message2c, 0, n);
```

To encrypt or decrypt entire streams, there exist the decorator classes `CipherInputStream` and `CipherOutputStream`. Objects of these classes are constructed with an existing `InputStream` or `OutputStream` object and an initialized `Cipher` object. Subsequent `read()` or `write()` operations result in encrypting or decrypting the data from or to the underlying stream on-the-fly.

```
CipherInputStream cis = new CipherInputStream(
                             fileInputStream, c1);
```

2.3.3 Mac class

Using message authentication codes (MAC) works similar as using ciphers. After creating a `Mac` object, `init()` is used to initialize it with a key and `doFinal` can be used to compute a HMAC over the data:

```
Mac m = Mac.getInstance("HmacMD5");
m.init(key);
byte[] hmac = m.doFinal(message);
```

Note that the HMAC algorithm is always used together with a hash algorithm – in this case MD5 – which is why we specified `HmacMD5`. Another option would be `HmacSHA1`.

Additional information about the key parameter (`key`) follows in section 2.3.4.

The `Mac` class also offers the `update` method, but it works a bit different than with the `Cipher` class. The `update` method serves to “put” data (byte arrays) into the `Mac` object, but does not compute parts of the MAC. When all data has been “put in”, the HMAC over all data is computed using `doFinal`:

```
while ( ... ) {
  m.update(data-to-be-included-in-mac-computation);
}
hmac = m.doFinal();
```

Here again, the following variant can be used to only pass the first `n` bytes to the `Mac` object:

```
m.update(data, 0, n);
```

In contrast to `Cipher` and also `MessageDigest` (creates a hash without using any key) there are no decorator classes to use `Mac` with streams.

2.3.4 Key, KeySpec and KeyGenerator classes

Keys are a somewhat complex topic in JCA. Basically, there are two fundamental interfaces, the `Key` interface and the `KeySpec` interface. Classes implementing the `Key` interface are usually just “containers for key material” while classes implementing the `KeySpec` interface offer additional functionality, for instance to convert keys from one encoding to another. When initializing objects with keys, then objects that implement the `Key` interface (or its subinterfaces) are used.

Often used `Key`s are for instance `SecretKey` for symmetric encryption, `PrivateKey` and `PublicKey` (und their subinterfaces) for asymmetric encryption and `PBEKey` for password-based encryption.

Classes the implement the `KeySpec` interface or subinterfaces of `KeySpec` include for instance `SecretKeySpec` for symmetric keys, `RSAPrivateKeySpec` and `RSAPublicKeySpec` for RSA keys, `DESKeySpec` for DES keys, `DHPrivateKeySpec` and `DHPublicKeySpec` for Diffie-Hellman keys and so on. In addition, there are the classes `PKCS8EncodedKeySpec` and `X509EncodedKeySpec`, both subclasses of `EncodedKeySpec`, which serve to read encoded private and public keys.

To create new key material, the `KeyGenerator` class can be used. The following generates a 128-bit long AES key:

```
kg = KeyGenerator.getInstance("AES");
kg.init(128);
SecretKey key = kg.generateKey();
```

The created key object (`SecretKey`) can then be used in the `init` method of `Cipher` or `Mac` (see `key` parameter of the `init` method in sections 2.3.2 and 2.3.3).

If the key material is available as a byte array (e.g. the 16 bytes of an AES key), you can use the class `SecretKeySpec` (which implements the `KeySpec` und the `SecretKey` interfaces and therefore also the `Key` interface) to create a key object. In the case of AES, this works as follows (`keyData` is a byte array that contains the key):

```
SecretKeySpec sKeySpec1 = new SecretKeySpec(keyData, "AES");
```

Likewise, it is possible to generate a key for a MAC; this simply requires specifying e.g. `HmacSHA1` instead of `AES`:

```
SecretKeySpec sKeySpec2 = new SecretKeySpec(keyData, "HmacSHA1");
```

Because the class `SecretKeySpec` implements the `SecretKey` interface (and therefore its superinterface `Key`), the generated key objects can also be used in the `init` method of `Cipher` or `Mac`.

To get the byte array representation from a key object (e.g. `SecretKeySpec` or `SecretKey`) you can use the `getEncoded` method:

```
byte[] keyBytes = key.getEncoded();
```

2.3.5 AlgorithmParameters class

When using a cipher, additional parameters to the key are sometimes used. These parameters can be stored in an `AlgorithmParameters` object. In this lab, this is used for the initialization vector (IV) because all cipher modes except ECB require an IV. If this parameter is not specified when initializing a `Cipher` in `ENCRYPT_MODE`, `Cipher` generates its own IV. In `DECRYPT_MODE`, the IV must be explicitly specified.

In this lab, you'll include the IV in an encrypted file (see section 0). Therefore, it's reasonable to explicitly create the IV using `SecureRandom` (with AES and a 128-bit key, the IV has a length of 16 bytes) and use it in the `init` method when initializing the `Cipher`. The following lines show how an `AlgorithmParameters` object for an AES cipher is created, how it is initialized with an IV (which is stored as a byte array in `iv`), and how the `AlgorithmParameters` object is then used to initialize the `Cipher` in `ENCRYPT_MODE`:

```
AlgorithmParameters algParam = AlgorithmParameters.
                               getInstance("AES");
algParam.init(new IvParameterSpec(iv));
cipher.init(Cipher.ENCRYPT_MODE, key, algParam);
```

2.3.6 CertificateFactory class

`CertificateFactory` reads data in X.509 format and converts them in `Certificate`s[6], `CertPath`s or `CRL`s. This allows e.g. to verify certificates or to use the public key stored in a certificate for encryption. The following lines read a certificate from an `InputStream` and create a corresponding `Certificate` object.

```
cf = CertificateFactory.getInstance("X.509");
Certificate certificate = cf.generateCertificate(
                          certificateInputStream);
```

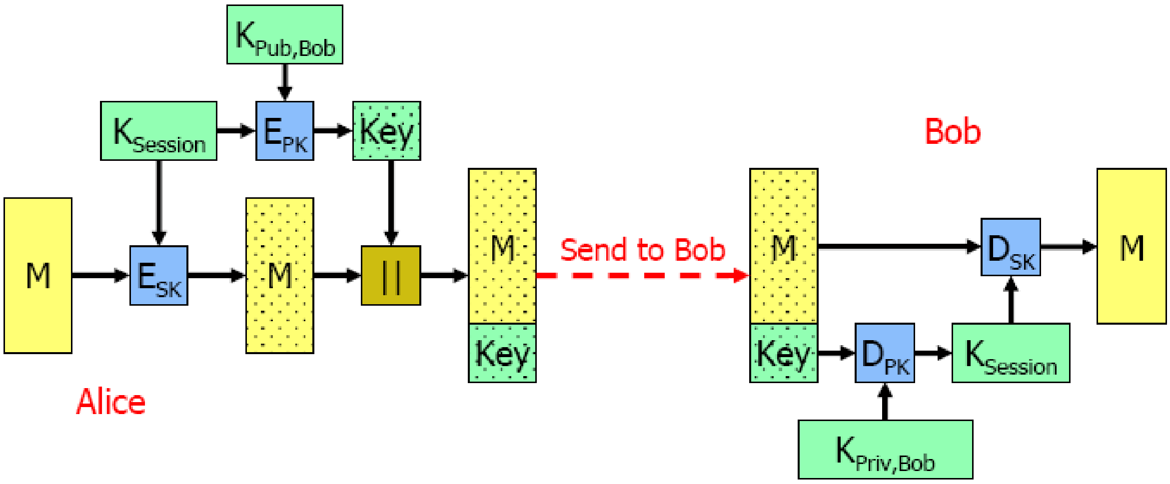
This certificate can then be used to e.g. initialize an RSA `Cipher`, which uses the public key in the certificate for encryption:

```
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, certificate);
```

[6] Note that there are two `Certificate` classes in Java SE, here you have to use `java.security.cert.Certificate`.

3 Task

To get familiar with the JCA you will develop an application to encrypt arbitrary data. The program – we name it SLCrypt (SL for Security Lab) – should be capable to encrypt data (using hybrid encryption) and decrypt it again.



With hybrid encryption, the data or message (M) is first encrypted symmetrically using a randomly generated key ($K_{Session}$). This session key is then encrypted with the public key of the recipient ($K_{Pub,Bob}$) and attached to the symmetrically encrypted message. The recipient first decrypts the session key using his private key ($K_{Priv,Bob}$), which is then used to decrypt the message.

Because pure encryption without authentication and integrity protection is problematic (as it allows attacks against the authenticity and integrity of the encrypted message), SLCrypt uses in addition a message authentication code (MAC). This MAC is computed over the plaintext message and added to the encrypted message. The recipient of the message has to check the MAC for correctness to verify the authenticity and integrity of the message. In SLCrypt, a password is used as MAC key.

3.1 Basis for this lab

There already exists an Eclipse project on the Ubuntu image that should be used as the basis.

- To start Eclipse, open a terminal, change to */home/user/eclipse* and enter *./eclipse*.
- The project directory is */home/user/workspace/SLCrypt*. Two directories *src* and *bin* are used for source and byte code.
- Running the program is best done in a terminal in directory *bin* as *user*.
- There's a directory *data* that contains a certificate (*certificate.cert*), the corresponding private key (*private_key.pkcs8*), and a test file (*testdoc.txt*) that can be used for encryption.

3.2 Program usage

You have to develop the encryption part of SLCrypt – `SLEncrypt` – as a command line program, which can be used as follows (in directory *bin* in a terminal):

```
java ch.zhaw.slcrypt.encrypt.SLEncrypt plain_file encrypted_file
     certificate_file mac_password
```

The four parameters have the following meaning:

- `plain_file` is the file name (relative or absolute path) of the plaintext document to be encrypted.
- `encrypted_file` is the file name where the encrypted document should be stored
- `certificate_file` is the file name of the X.509-encoded certificate (that contains the public key) of the recipient of the encrypted document.

- `mac_password` is the password used to compute the MAC.

3.3 Cryptographic operations and file format

In the following, the process how a document is protected is described. We start with the plaintext document that must be protected.

Document

In the first step, the MAC is computed over the document. We use HMAC with SHA-1, which is denoted as `HmacSHA1` in Java. For the key, the password that was specified on the command line is used. The MAC has a length of 20 bytes and is prepended (at the front) to the document. We name the resulting data structure *DocumentMAC*.

HMAC	Document	*DocumentMAC*

In the next step, *DocumentMAC* is encrypted symmetrically with the session key. Although the data format is basically able to cope with arbitrary ciphers (the corresponding transformation name is included in the header, see below), we use AES with a 128-bit key in CBC mode in this lab. As padding scheme, PKCS5Padding is used. The name of the corresponding transformation is `AES/CBC/PKCS5PADDING`. The result of the encryption is the following encrypted data structure (the padding is not depicted), which we identify as *EncryptedDocumentMAC*.

HMAC	Document	*EncryptedDocumentMAC*

The session key that was used for encryption above is included in a data structure we identify as *SessionKey*. This data structure not only contains the actual session key but consists of three parts: the name of the transformation, the IV (remember we are using CBC mode), and the key itself. The data structure *SessionKey* is defined as follows, the table describes the meaning of the fields:

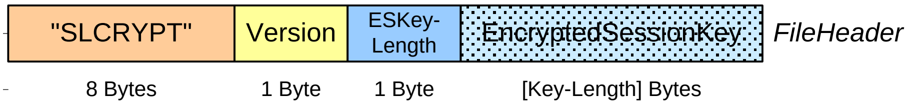
TName-Length	Transformation-Name	IV-Length	IV	Key-Length	Key	*SessionKey*
1 Byte	[Name-Length] Bytes	1 Byte	[IV-Length] Bytes	1 Byte	[Key-Length] Bytes	

TName-Length	1 Byte	Length of TransformationName.
TransformationName		Algorithm/Mode/Padding of the used encryption, which is required to configure the `Cipher` object. Examples are `AES/CBC/PKCS5PADDING`, `DES` etc. As mentioned above, you should use `AES/CBC/PKCS5PADDING` with a 128-bit key (and therefore also a 128-bit IV).
IV-Length	1 Byte	Length of IV in bytes. If no IV is used (e.g. with ECB-mode), it is set to 0.
IV		Initialization vector for the symmetric encryption.
Key-Length	1 Byte	Length of Key in bytes.
Key		The symmetric session key.

The entire data structure *SessionKey* is encrypted with the public key of the recipient. You get the public key from the certificate, which is passed to `SLEncrypt` on the command line. This encryption uses RSA in PKCS#1 format. The result is the data structure *EncryptedSessionKey*.

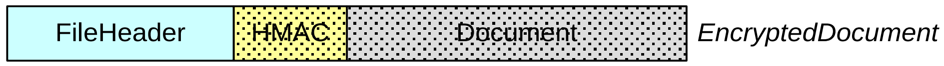
TName-Length	Transformation-Name	IV-Length	IV	Key-Length	Key	*EncryptedSessionKey*
1 Byte	[Name-Length] Bytes	1 Byte	[IV-Length] Bytes	1 Byte	[Key-Length] Bytes	

Finally, we need a file header that contains meta information and the *EncryptedSessionKey*. The format of the file header (we name this data structure *FileHeader*) is as follows, the meaning of the fields is given in the table below:



FormatString	8 Bytes	Identifier of the data format, is always `SLCRYPT`.
Version	1 Byte	Version of the SLCrypt formats. Here 0x01.
ESKey-Length	1 Byte	Length of *EncryptedSessionKey* in bytes.
EncryptedSessionKey		The Data structure *EncryptedSessionKey* described above (*SessionKey* encrypted with RSA in PKCS#1 format, using the public key from the certificate of the recipient).

The complete encrypted file (which we name *EncryptedDocument*) consists of the *FileHeader* (which among other information contains *EncryptedSessionKey*) and *EncryptedDocumentMAC* (the combination of HMAC and *Document*, encrypted with the session key):



3.4 Implementation

As mentioned above, there's an Eclipse project available on the Ubuntu image. We strongly recommend using the given program basis for your implementation.

General program components are located in package `ch.zhaw.slcrypt`. Classes that are specifically used for encryption can be found in package `ch.zhaw.slcrypt.slencrypt` and those for decryption in package `ch.zhaw.slcrypt.sldecrypt`. In the following, the classes that will be relevant for you are described on detail.

These two classes will help you to create the data structures *SessionKey* and *FileHeader*:

- `ch.zhaw.slcrypt.SessionKey` manages the data structure *SessionKey* (see above), which contains the transformation name, the IV and the session key itself. These three values can be set using `setTransformName()`, `setIV()` und `setKey()` and read using the corresponding `get`-variants. If the values are set, the method `encode()` can be used to return the data structure *SessionKey* as a byte array. The class also offers functionality to decode a *SessionKey* data structure, but you won't use this as you'll only implement the encryption part of `SLCrypt`.
- `ch.zhaw.slcrypt.FileHeader` manages the date structure *FileHeader* (see above). `setVersion()` sets the version number and `setEncryptedSessionKey()` sets the data structure *EncryptedSessionKey*. `encode()` returns the data structure *FileHeader* as a byte array. This class also offer decoding functionality, which you won't use.

To implement the encryption, the abstract class `ch.zhaw.slcrypt.HybridEncryption` is given. It's only non-abstract method is `encryptDocumentStream()`, which preforms the complete encryption of a document. This method calls five abstract methods, which you have to develop in a subclass of `HybridEncryption`. To do this, the class `HybridEncryptionImpl` is provided, which you have to complete. The five methods should offer the following functionality:

- `generateMAC(InputStream document, byte[] passwordMAC)` gets the document (`document`) and the MAC key (`passwordMAC`) and computes the MAC (HMAC-SHA1) over the document. The method returns an `InputStream`, from which the combined MAC and document can be read. This means that this method basically produces the data structure *DocumentMAC* as described above.

Hint: To create the `InputStream` return value, it's easiest to put the data you read from `document` (which you certainly have to do to compute the MAC) right back into a `ByteArrayOutputStream`:

```
ByteArrayOutputStream documentBackup =
                      new ByteArrayOutputStream();

int len;
byte[] tmpBuf = new byte[10];
while ((len = document.read(tmpBuf, 0, 10)) >= 0) {
    documentBackup.write(tmpBuf, 0, len);
    // here, you should also do something with tmpBuf
    // to compute the mac...
}
```

To create and return the combined `InputStream` consisting of the MAC and `documentBackup`, you can use `SequenceInputStream`:

```
// mac contains the byte-Array with the computed MAC
return new SequenceInputStream(new ByteArrayInputStream(mac),
    new ByteArrayInputStream(documentBackup.toByteArray()));
```

- `generateSessionKey()` should generate a `SessionKey` object (`ch.zhaw.slcrypt.SessionKey`) that contains all relevant information (transformation name, session key, IV). Use AES/CBC/PKCS5Padding as transformation name and 128 bits for the length of the key and the IV. Use cryptographically strong random numbers for the key and the IV.
- `encryptDocumentMAC(InputStream documentMAC, SessionKey sessionKey)` should symmetrically encrypt the combination of MAC and document (`documentMAC`) using the transformation name, key and IV generated above. These three values have been stored in a `SessionKey` object (see `generateSessionKey()` method above), which is passed to this method in parameter `sessionKey` , and can be read using the corresponding get methods. The method should return an `InputStream`, from which the encrypted data (MAC and document) can be read, this encrypted data corresponds to the data structure *EncryptedDocumentMAC*. For this `InputStream`, you should use the class `CipherInputStream` (see section 2.3.2). This means that you do not perform the actual encryption in this method and the encryption is done on the fly when reading data from the returned `InputStream`.
- `encryptSessionKey(SessionKey sessionKey, InputStream certificate)` should encrypt the *SessionKey* data structure (`sessionKey`) with the public key in the certificate (parameter `certificate`) using RSA. The result is the data structure *EncryptedSessionKey*, which is returned as a byte array.
- `generateFileHeader(byte[] encryptedSessionKey)` generates the file header using the *EncryptedSessionKey* data structure (`encryptedSessionKey`) and returns a `FileHeader` object (`ch.zhaw.slcrypt.FileHeader`). The version can be set to 1.

Finally, there's the class `ch.zhaw.slcrypt.SLEncrypt` that contains the main method. The class handles reading the files from the file system and storing the encrypted document. For encryption, the method `encryptDocumentStream()` that is described above and that uses the five methods you have to implement. You can use this class without having to adapt it.

3.5 Tests

To test the correctness of you encryption program, the decryption program `ch.zhaw.slcrypt.SLDecrypt` is provided. It reads an encrypted file and decrypts it using the private key of the recipient. It is used as follows:

```
java ch.zhaw.slcrypt.decrypt.SLDecrypt encrypted_file
     decrypted_file private_key_file mac_password
```

`encrypted_file` is the file name of the encrypted document to use and `decrypted_file` the file name where to store the decrypted document. `private_key_file` is the file name of the private key to be used (encoded in PKCS#8 format) and `mac_password` is the password to be used to verify the MAC. Of course, the content of the decrypted document should be the same as the original plaintext document that was encrypted using `SLEncrypt` (compare the contents of the files) and verifying the MAC should be successful (check the output of `SLDecrypt` in the terminal).

If you manage to encrypt files than can be correctly decrypted with `SLDecrypt` (including correct verification of the MAC), you are ready to collect the lab points.

Lab Points

For **3 Lab Points** you have to send an e-mail to the instructor that contains a file that was encrypted using your program. If it can be decrypted correctly by the instructor, you get 2 points and if the MAC can be correctly verified as well you get a third point. In addition, you must include the source code of your implementation of `HybridEncryptionImpl.java` (non-encrypted) in the e-mail.

The following rules apply:

- Use a plain ASCII text file. You can choose your own content, but there should be a connection to your names and your group (it's easiest to just include your group and the names).
- Use `GroupXY` depending on your group number as password for the MAC. That's an upper case ‚`G`' and the group number consists of two digits, e.g. `Group02` or `Group14`.

Use „SecLab - JCA - group X - name1 name2" as the e-mail subject; corresponding to your group number and the names of the group members.