Security Lab – Cryptography in Java

VMware

- This lab can be solved with the **Ubuntu image**, which you should start in networking-mode **Nat**. The remainder of this document assumes you are working with the Ubuntu image.
- You can basically solve this lab also on your own system, provided you have an IDE and a current version of Java installed. Furthermore, you must install an additional Cryptographic Service Provider as Java 8 does not include support for SHA-3 (the Ubuntu image uses Bouncy Castle[1] for this) and you must also install the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files"[2] as Java does not allow using secret keys longer than 128 bits per default. All this is already installed on the Ubuntu image, so it's easiest to work with the image.

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.

You should first read the entire section 2 to deepen you knowledge about the Java Cryptography Architecture. With this information, you should then be ready to solve the task in section 3.

2 Basics: Java Cryptography Architecture

The Java Cryptography Architecture[3] (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[4]. 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

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

[2] Search the web for "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" and you'll find more information and the policy files.

[3] http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

[4] http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html

others. In addition, there are some third-party CSPs, among the most popular ones is the Open Source Bouncy Castle Crypto API.

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 SHA256 hashes (based on SHA-2), this is done as follows:

```
MessageDigest md = MessageDigest.getInstance("SHA256");
```

The method returns a `MessageDigest` object to compute SHA256 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. However, it may be that you need functionality that is not provided by the standard Java providers, e.g. SHA-3. In this case, you have to install an additional provider that supports this. On the image with which you are working, the Bouncy Castle CSP is additionally installed, so you can use SHA-3 using the `getInstance` method described above. You can also explicitly specify the CSP provider to be used for a specific cryptographic operation. This is done with a second variant of the static method. In the case of the Bouncy Castle CSP, this would look as follows:

```
MessageDigest md = MessageDigest.getInstance("SHA256", "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[5].

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();
```

[5] http://docs.oracle.com/javase/8/docs/api

Random numbers are generated using the function `nextBytes()`. The following line generates 32 random bytes and stores them in the array `bytes`:

```
byte bytes[] = new byte[32];
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[6]. These combinations are named transformations and have the following form:

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

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

```
Cipher c1 = Cipher.getInstance("AES/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("AES");
```

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();
```

[6] http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html

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("HmacSHA512");
m.init(key);
byte[] hmac = m.doFinal(message);
```

Note that the HMAC algorithm is always used together with a hash algorithm – in this case SHA512 – which is why we specified `HmacSHA512`.

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 **Error! Reference source not found.**). 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 Galois/Counter Mode

The Galois/Counter mode is a block cipher mode that was developed to combine encryption and integrity protection. It only needs one key to encrypt the plaintext and compute an Auth Tag (which basically corresponds to a MAC). In addition, it supports "additionally authenticated data", which is data that is (in addition to the plaintext) authenticated but not encrypted. This is very convenient to e.g. include a packet header into the authenticated data.

Just like CBC mode, GCM mode requires an IV. In addition, it requires the length of the Auth Tag, which can vary. To pass these parameters during initialization of the cipher, a `GCMParameterSpec` object must be created and used, similar to `AlgorithmParameters` above. The following creates such an object using an `iv` (a byte array) and 128 for the length of the `Auth Tag`:

```
GCMParameterSpec gcmParameters = new GCMParameterSpec(128, iv);
cipher.init(Cipher.ENCRYPT_MODE, key, gcmParameters);
```

Adding additional authenticated data (as byte array) to the cipher must be done before the data for encryption is fed into the cipher using the `updateAAD` method:

```
cipher.updateAAD(additionally-authenticated-data);
```

2.3.7 CertificateFactory class

`CertificateFactory` reads data in X.509 format and converts them in `Certificates`[7], `CertPaths` or `CRLs`. 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);
```

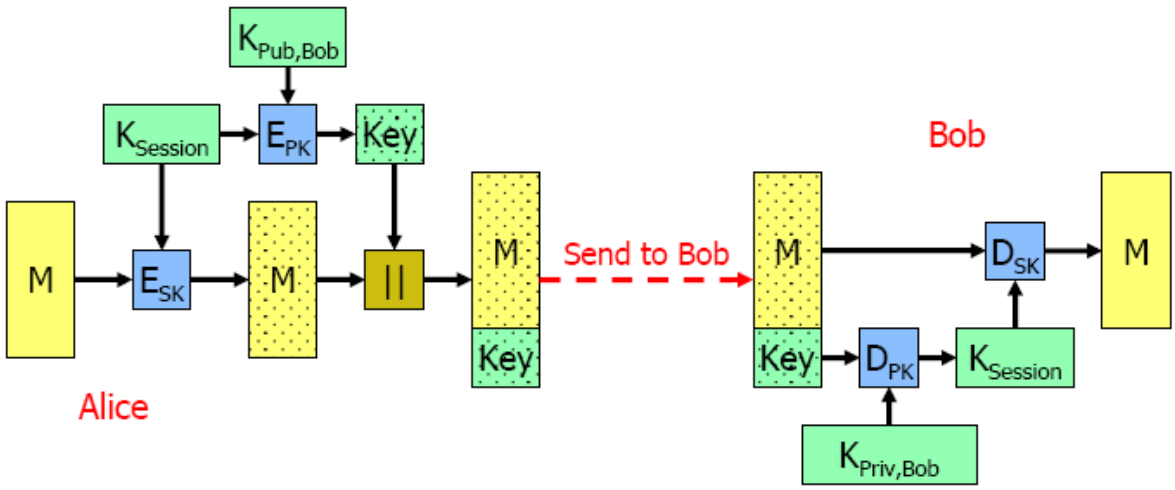
This certificate can then be used to e.g. initialize an RSA `Cipher`, which uses the public key in the certificate for encryption:

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

3 Task

This section describes the task you have to solve. Read through the entire section 3 (including all subsections) before you start implementing to understand correctly what you have to do.

To get familiar with the JCA you will develop an application to encrypt and integrity-protect arbitrary data using different algorithms. You don't have to develop the entire application from scratch as a significant portion of it is already provided as a basis. The program – we name it SLCrypt (SL for Security Lab) – should be capable to encrypt data (using hybrid encryption, the concept of which is illustrated in the image below) and decrypt it again. In addition, the data is integrity-protected to detect any tampering.



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.

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

Because raw 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 entire data (including the ciphertext) and appended to the data. 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.

Furthermore, SLCrypt supports Galois/Counter Mode (GCM). In this mode, MAC-computation is included in the cipher operation as described in Section 2.3.6 and therefore, no extra MAC must be computed and no password is needed.

To completely solve this lab, your application must support the following ciphers (if necessary, consult the IT-Sicherheit (IS) module, where all these ciphers and modes are explained):

- `AES/CBC/PKCS5Padding` with key lengths 128, 192, and 256 bits
- `AES/GCM/NoPadding` with key lengths 128, 192, and 256 bits
- A stream cipher, You can choose between the following:
  - `RC4` with a key length of 128 bits (note that RC4 is no longer considered to be very secure).
  - `AES/CTR/NoPadding` with key lengths 128, 192, and 256 bits (AES in counter mode, which is a mode to use a block cipher as the keystream generator for a stream cipher).

In addition, your program must support the following MACs:

- `HmacMD5`
- `HmacSHA1`
- `HmacSHA256` (based on SHA-2)
- `HmacSHA512` (based on SHA-2)
- `HmacSHA3-256` (based on SHA-3)
- `HmacSHA3-512` (based on SHA-3)

3.1 Basis for this lab

As mentioned above, a significant part of the program is already provided as a basis. Do the following to set up this basis:

- Download *SLCrypt.zip* from OLAT.
- Move the file to an appropriate location (e.g. in a directory *securitylabs* in the home directory */home/user*).
- Unzip the file. The resulting directory *SLCrypt* contains a Java project based on Maven. This should be importable in any modern IDE. The remainder assumes you are using *NetBeans*, which is installed on the Ubuntu image.
- Start *NetBeans* and open the project.
- To build the project, right-click *SLCrypt* in the *Projects* tab and select *Clean and Build*.
  - Whenever you do some changes, the project is rebuilt automatically, so it's usually not necessary to use *Clean and Build* again. However, if «something doesn't work as it should», do again a *Clean and Build*.
  - The executable code is placed in directory *SLCrypt/target/classes*.
- There's a directory *data* (just below *SLCrypt/target/classes*) 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.

- In general, run the program from the command line (not from within the IDE). Do this in a terminal as user *user*.

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 *SLCrypt/target/classes* in a terminal):

```
java ch.zhaw.securitylab.slcrypt.encrypt.SLEncrypt plain_file
encrypted_file certificate_file cipher_algorithm keylength
[mac_algorithm mac_password]
```

The parameters have the following meaning:

- `plain_file` is the file name (relative or absolute path) of the plaintext document to be encrypted. You can use the test file in *SLCrypt/target/classes/data* or any other document.
- `encrypted_file` is the file name (relative or absolute path) where the encrypted document should be stored. You can store it in *SLCrypt/target/classes/data* or in any other location.
- `certificate_file` is the file name of the X.509-encoded certificate (that contains the public key) of the recipient of the encrypted document. You can use the certificate in *SLCrypt/target/classes/data*.
- `cipher_algorithm` is the name of the cipher to use, e.g. `AES/CBC/PKCS5Padding` or `AES/GCM/NoPadding`.
- `keylength` is the length of the encryption key in bits, e.g. `128`.
- `mac_algorithm` is the name of the mac algorithm to use, e.g. `HmacSHA256`.
- `mac_password` is the password used to compute the MAC, e.g. `supersecret`.

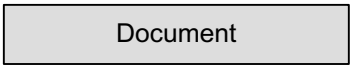
Note that the last two parameters are not needed if CGM is used (which already includes the MAC computation).

Below are two valid usage examples:

- ```
  java ch.zhaw.securitylab.slcrypt.encrypt.SLEncrypt plain_file
  encrypted_file certificate_file AES/CBC/PKCS5Padding 192 HmacSHA256
  supersecret
  ```
- ```
  java ch.zhaw.securitylab.slcrypt.encrypt.SLEncrypt plain_file
  encrypted_file certificate_file AES/GCM/NoPadding 128
  ```

3.3 Cryptographic operations and file format

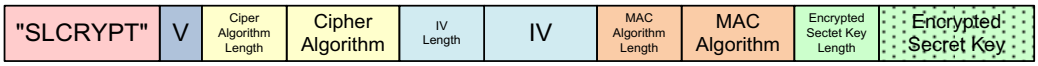
The protected files must follow a specific file format so the decryption part of SLCrypt can correctly decrypt and verify the protected files. Therefore, the file does not only contain the protected data, but also a file header with metadata. In the following, the process how a document is protected and the file format that is used for protected documents are described. We start with the plaintext document that must be protected.



To encrypt this document (this is described below), a secret key is created. To protect the secret key (remember that we are using hybrid encryption), the secret key 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 *EncryptedSecretKey*.

EncryptedSecretKey

Now, the file header that contains metadata about algorithms and parameters used for document protection is created, which we identify as *FileHeader*.



FileHeader

The fields in the file header are as follows:

SLCRYPT	7 bytes	Identifier of the data format, is always `SLCRYPT`.
Version (V)	1 byte	Version of the SLCrypt format. Here 0x01.
Cipher Algorithm Length	1 byte	Length of the cipher algorithm name (next field) in bytes.
Cipher Algorithm	> 0 bytes	The name of the cipher algorithm that is used. E.g. `AES/CBC/PKCS5PADDING`, `AES/GCM/NoPadding`, or `RC4`.
IV Length	1 byte	Length of the IV (next field) in bytes. 0 if no IV is used.
IV	≥ 0 bytes	The initialization vector that is used for encryption/decryption. If no IV is used (e.g. with RC4), the field is empty.
MAC Algorithm Length	1 byte	Length of the MAC algorithm name (next field) in bytes. 0 if no MAC is used.
MAC Algorithm	≥ 0 bytes	The name of the MAC algorithm that is used. E.g. `HmacSHA1` or `HmacSHA512`. If no MAC is used (e.g. with GCM), the field is empty.
Encrypted Secret Key Length	1 byte	Length of the Encrypted Secret Key (next field).
Encrypted Secret Key	> 0 bytes	The encrypted secret key (EncryptedSecretKey), corresponds to the secret key encrypted with RSA in PKCS#1 format, using the public key from the certificate of the recipient.

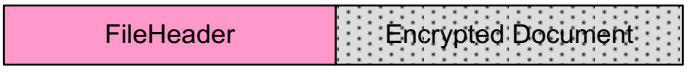
In the next step, the document is encrypted with the specified cipher algorithm using the secret key created before. Note that this is done differently depending on the cipher mode:

- With any mode other than GCM, the document is simply encrypted.
- With GCM, the document is also encrypted. But in addition, the file header is included as "additionally authenticated data" because we want to integrity-protect all data (in other modes, this will be done separately with a MAC, see below).

What results is what we identify as *EncryptedDocument*. Note that if we use GCM, this also includes the Auth Tag (i.e. a MAC), but this is not specifically illustrated.

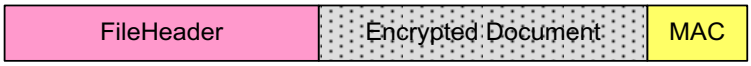
EncryptedDocument

Next, the header is prepended to the encrypted document. The resulting data structure is identified as *FileHeaderEncryptedDocument*:



FileHeaderEncryptedDocument

Finally, if another mode than CGM is used, the MAC is computed over FileHeaderEncryptedDocument and appended to the data. This results in the final data structure, *FileHeaderEncryptedDocumentMAC*:



Note this last step is only needed if another cipher mode than CGM was used.

Note also that security-wise, this is a sound approach. In particular, we use the "Encrypt then MAC" approach, which means data is first encrypted and only then integrity-protected, which is more secure than "MAC then Encrypt". For details, refer to the module IT-Sicherheit (IS).

3.4 Implementation

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

`ch.zhaw.securitylab.slcrypt.SLEncrypt` contains the main method for encryption and integrity-protection. The class handles reading the files from the file system and storing the protected document. For encryption, the method `encryptDocumentStream()` in the abstract class `ch.zhaw.securitylab.slcrypt.HybridEncryption` is used. `encryptDocumentStream()` performs the complete encryption and integrity-protection of a document. This method calls five abstract methods, which you have to implement in a subclass of `HybridEncryption`. To do this, the class `HybridEncryptionImpl` is provided, which you have to complete. Do only work with `HybridEncryptionImpl`, don't change any other class.

Before we discuss the five methods, we look at two additional classes you will use.

- `ch.zhaw.securitylab.slcrypt.FileHeader` manages the date structure *FileHeader* (see above). It provides getter and setter methods to get and set the cipher algorithm, the IV, the MAC algorithm, and the encrypted session key (`getCipherAlgorithm()`, `setCipherAlgorithm(...)` etc.). The class also has a method `encode()`, which – after having set all attributes – returns the FileHeader data structure as a byte array. This class also offers decoding functionality, but this is only used during decryption, so you won't use it
- `ch.zhaw.securitylab.slcrypt.Helpers` provides helper methods that will be useful. For instance, it provides methods `isCBC(...)` and `isGCM(...)` that return whether a cipher algorithm uses CBC mode or GCM. Furthermore, `getCipherName(...)` returns the raw cipher name of a cipher algorithm, e.g. it returns `AES` when `AES/CBC/PKCS5PADDING` is passed as parameter. Also, `getIVLength(...)` returns the length of the IV of a cipher algorithm in bytes.

In the following, the five methods you have to implement in `HybridEncryptionImpl` are described:

- ```
  byte[] generateSecretKey(String cipherAlgorithm,
                           int keyLength)
  ```

  This method takes the cipher algorithm name and the key length (in bits) as parameter and returns a secret key that can be used for encryption as a byte array.

- ```
  byte[] encryptSecretKey(byte[] secretKey,
                          InputStream certificate)
  ```

  This method takes the secret key and an input stream from which the certificate can be read as inputs. It uses the RSA public key in the certificate and uses it to encrypt the secret key. The encrypted secret key is returned as a byte array.

- ```
  FileHeader generateFileHeader(String cipherAlgorithm,
                                String macAlgorithm,
  ```
  ```

```
                                    byte[] encryptedSecretKey)
```

This method takes the cipher algorithm name, the mac algorithm name, and the encrypted secret key as input and creates a `FileHeader` object (using the setter methods provided by `FileHeader`). If the cipher algorithm uses CGM mode, the parameter `macAlgorithm` can be ignored and the MAC algorithm name in the `FileHeader` object should be set to the empty string. Likewise, if the cipher does not require an IV, the IV in the `FileHeader` object should be set to a byte array of length 0.

- ```
  byte[] encryptDocument(InputStream document,
                         FileHeader fileHeader,
                         byte[] secretKey)
  ```

  This method takes an input stream, from which the document to protect can be read, the pre-filled `FileHeader` object, and the secret key to use for encryption as input. It encrypts the document using the secret key and returns the encrypted document as byte array. In this method, you must distinguish between the different cipher algorithms (the one to use has been previously stored in the `FileHeader` object). For instance, if an IV is required, use the IV that was previously filled into the `FileHeader` object. In addition, if GCM is used, make sure to add the file header (use `encode()` to get it as byte array) as additionally authenticated data before encryption. Also, use 128 bits for the length of the Auth Tag if GCM is used. Furthermore, note that because the document is available as an input stream, it makes sense to use the class `CipherInputStream` (see section 2.3.2) to encrypt the document.

- ```
  byte[] computeMAC(byte[] dataToProtect,
                    String macAlgorithm,
                    byte[] password)
  ```

  This method receives a byte array with the data to protect, the MAC algorithm name to use and a MAC password as inputs and computes the MAC over `dataToProtect`. The MAC is returned as byte array. Note that with GCM, this method is never called, but this is already handled correctly in `encryptDocumentStream()` in the class `HybridEncryption`.

3.5 Tests

To test the correctness of your encryption program, the decryption program `ch.zhaw.securitylab.slcrypt.SLDecrypt` is provided. It reads an encrypted file and decrypts it using the private key of the recipient. It is used as follows (in directory *SLCrypt/target/classes* in a terminal):

```
java ch.zhaw.securitylab.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, you can use the private key in *SLCrypt/target/classes/data*) and `mac_password` is the password to be used to verify the MAC. This password is optional and only provided if another mode than GCM is used. 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 decryption is successful, an output as follows is shown:

```
MAC:       HmacSHA256
MacDoc:    5a7a40e33e07ccaabbc994895f8b650129bf1341958d03009704baa3f449814f
MacComp:   5a7a40e33e07ccaabbc994895f8b650129bf1341958d03009704baa3f449814f
MAC:       Successfully verified
```

```
Cipher:    AES/CBC/PKCS5Padding
Keylength: 128
Key:       2f73cbb58235413f959439b0f7fbc8b7
IV:        b28133cb66dec83990874b24c0b96b91

Plaintext (69 bytes): The ultimate test document for the Java cryptography
security lab!!!
```

If you manage to encrypt files than can be correctly decrypted with `SLDecrypt` (including correct verification of the MAC) – for all ciphers and MAC algorithms listed at the beginning of Section 3, you are ready to collect the lab points.

Lab Points

For **3 Lab Points** you have to do the following:

- Create an ASCII text file that is used as plaintext. 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).
- Encrypt and integrity-protect the file with your program 3 times to produce 3 different protected documents. You must use the following algorithms and key lengths to create the protected files:
  - One file must use `AES/CBC/PKCS5Padding`
  - One file must use `RC4` or `AES/CTR/NoPadding`
  - One file must use `AES/GCM/NoPadding`
  - With respect to key lengths, one file must use 128, one must use 192, and one must use 256 bits. You can decide on your own what to use with what file.
  - With respect to MAC algorithms, one file should use HMAC based on SHA-2 and one should use HMAC based on SHA-3. Use either 256 or 512 bits for the hash length.
  - For the MAC password, choose anything you want, but make sure to include the password in the e-mail (see below).
- Send an e-mail to the instructor that contains the 3 protected files and the MAC password. If the files can all be correctly decrypted (including correct MAC verification), you get 3 points (1 point for each file).
- In addition, you must include the source code of your implementation of `HybridEncryption-Impl.java` (non-encrypted) in the e-mail.
- Use *SecLab - Java Crypto - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members.