Security Lab – Cryptography in Java

VMware

This lab can be done with the **Ubuntu image**. The remainder of this document assumes you are working with this image.

You can do this lab also on your own system, provided you have an IDE and a current version of Java installed.

1 Introduction

When developing software, there's often the requirement to perform cryptographic operations. For instance, this may be the case in a program that encrypts and decrypts data or in a server application that uses the HTTPS protocol (HTTP over TLS). In these situations, it's not reasonable to completely implement ciphers or secure communication protocols on your own. Instead, you should use available, well-established components and focus on using them in a secure way.

This is exactly what you'll do in this lab. Your task is to use the cryptographic functions offered by Java to implement a program that can be used to encrypt and integrity-protect files. The goal is that you get familiar with the cryptographic functions of Java and that you can apply them correctly.

You should first read the entire section 2 to deepen your 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[1] (JCA) is a component of Java SE that provides various cryptographic functions including secret key block and stream ciphers, public key ciphers, key generators, hash functions, message authentication codes (MAC), digital signatures, and certificates. Other security components of Java are often based on the JCA, e.g., JSSE (for SSL/TLS) and JGSS (for Kerberos), but in this lab the focus is on the JCA.

2.1 Cryptographic Service Providers

The JCA uses a provider-based architecture, which means the actual implementations of the cryptographic functions are provided (in a plug-in manner) by software components identified as Cryptographic Service Providers[2] (CSP). Java SE includes several such CSPs[3] per default and as a result of this, Java SE supports virtually all of the widely used cryptographic functions «out of the box». The 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 exist some CSPs that are provided by 3rd parties. If you want to use such a 3rd party CSP, you have to install it manually (see section 3). Usually, 3rd party CSPs are only used if Java SE does not support a specific cryptographic algorithm you want use. One of the most popular 3rd party CSPs is the Bouncy Castle CSP.

2.2 Basic Usage

To use a cryptographic function in a program, it is usually required to use the static method *getInstance* 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:

---

[1] https://docs.oracle.com/javase/10/security/java-cryptography-architecture-jca-reference-guide.htm

[2] Basically, a CSP is a software component (i.e., a library) that can be plugged into the JCA and that contains classes that provide the functionality of one or more cryptographic algorithms.

[3] https://docs.oracle.com/javase/10/security/oracle-providers.htm

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

The method returns a *MessageDigest* object from one of the installed CSPs – assuming at least one of them supports SHA-2 – and this object can then be used to compute SHA256 hashes. If multiple installed CSPs support the function, then the one with the highest priority is used.

When using the *getInstance* method of any of the factory classes of the JCA (e.g., *MessageDigest*, *Cipher*, *Mac*, *KeyGenerator* etc., details see below), then the returned object is typically from one of the CSPs that are included in Java SE per default, as they support a wide range of cryptographic operations. However, it may be that you need a cryptographic algorithm (e.g. an only recently published secret key cipher) that is not supported by the CSPs included in Java SE. In this case, as already mentioned above, you have to install a 3rd party CSP that supports the desired algorithm (e.g., the Bouncy Castle CSP). Once this has been done, the *getInstance* method can be used in the same way as above and will return an object from the 3rd party CSP.

If you want to explicitly specify the CSP to be used for a specific cryptographic operation, you can use a second variant of the method *getInstance*. So assuming that the Bouncy Castle CSP is installed and that you want to specifically use the SHA-2 implementation provided by this CSP (and not the one which is part of Java SE), this would be done as follows:

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

2.3 Classes

In the following, several classes of the JCA are described in detail, in particular also the ones that you need to use to successfully complete this lab. Additional information can be found in the Java API Specifications[4].

2.3.1 SecureRandom class

SecureRandom generates cryptographically strong random numbers. Java supports several random number generator (RNG) algorithms, depending on the underlying operating system. If possible, *SecureRandom* uses the random sources provided by the underlying operating system, e.g., */dev/random* or */dev/urandom* on Linux/Unix/macOS-like systems. These random sources either use a hardware random number generator (also called true RNG (TRNG)) if available on the system or a pseudo random number generator (PNRG) that is seeded with random material collected by the operating system (inputs from mouse, network, keyboard,...). In addition, *SecureRandom* also supports general PRNGs such as DRGB and SHA1PNRG, which are seeded by the random sources provided by the OS. In general, it's best to create *SecureRandom* objects without specifying the specific algorithm to use as this uses «the best» RNG (TRNG or PRNG) depending on the operating system. This is done as follows:

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

If you really want to use a specific RNG, e.g., *SHA1PRNG*, the *SecureRandom* object is created as follows (but as mentioned about, you usually shouldn't do this):

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

Random numbers are generated using the method *nextBytes()*. The following two lines generate 32 random bytes and store 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

[4] https://docs.oracle.com/en/java/javase/11/docs/api/index.html

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 an 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 (in the case of AES, the default values are *ECB* (which is insecure!) and *PKCS5Padding*):

```
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);
```

In many cases (e.g., when using CBC, CTR or GCM mode or when using the cipher CHACHA20), an additional parameter must be specified to initialize the cipher (e.g., an initialization vector (IV)). This can be done by using a third parameter when initializing the cipher, which is an object of any of the JCA classes that provide parameter specifications (e.g., *IvParameterSpec*, *GCMParameterSpec*, *ChaCha20ParameterSpec* etc.) In this case, initialization of the cipher works as follows:

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

Details about using this third parameter follow below in sections 2.3.5 - 2.3.7.

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 stores the ciphertext in *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 usually processed. In general, the final operation must always be a call to *doFinal* (with or without additional data as parameter), as only this guarantees that the final block is padded correctly. With stream ciphers, the final call of *doFinal* is usually not «strictly» required as no padding is done, but it should still be done due to best practice as it is not guaranteed that update processes all bytes that could be processed, e.g. because of optimization reasons. As an example, the following three lines show twice a call of the *update* method and a necessary final call of *doFinal*. Note that the content of the byte array *ciphertext* will of course be overwritten with each call, so it's important to store the received ciphertext after each call, e.g. be writing (appending) it into a file or into a larger byte array.

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

Decrypting basically works the same (the main difference is that the *init* method must use *Cipher.DECRYPT_MODE*) and in this case, *doFinal* removes the padding from the last plaintext block after decryption.

[5] https://docs.oracle.com/javase/10/docs/specs/security/standard-names.html

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 by using 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. As an example, the following line constructs a *CipherInputStream* object:

```
CipherInputStream cis = new CipherInputStream(inputStream, 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 differently 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, *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* and 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. *HmacSHA256* instead of *AES*:

```
SecretKeySpec sKeySpec2 = new SecretKeySpec(keyData, "HmacSHA256");
```

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 IvParameterSpec class

Many ciphers, e.g., when a block cipher is used in CBC mode, require an initialization vector (IV). If a cipher needs an IV and if no IV is specified when initializing the corresponding *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 the header of an encrypted file (see section 3.3). Therefore, it's reasonable to explicitly create the IV using *SecureRandom* and use it in the *init* method when initializing the *Cipher* for encryption. The following lines show how an *IvParameterSpec* object is created based on an IV value *iv* (a byte array) and how it is then used to initialize the *Cipher* in *ENCRYPT_MODE*:

```
IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
cipher.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
```

2.3.6 Galois/Counter Mode

The Galois/Counter Mode (GCM) is a block cipher mode that was developed to combine encryption and integrity protection. It only needs one key as an input and uses this to encrypt the plaintext and to compute an Auth Tag for integrity protection (the Auth Tag basically corresponds to a MAC). In addition, it supports «additionally authenticated data», which is data that is (in addition to the plaintext) also integrity-protected by the Auth Tag but that is not encrypted. This is very convenient to provide integrity-protection also for, e.g., a file header or a packet header.

Just like, e.g., 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 *IvParameterSpec* above. The following creates such an object using an *iv* (a byte array created using *SecureRandom*) and 128 for the length of the Auth Tag:

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

Adding additionally authenticated data (as byte array) to the cipher must be done with the *updateAAD* method (this must be done *before* the data to be encrypted is processed with *update* or *doFinal*):

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

2.3.7 CHACHA20 Cipher

CHACHA20 is a relatively novel stream cipher. CHACHA20 requires two initialization parameters: a 12-byte nonce and a counter. The nonce basically corresponds to an IV. To pass these parameters during initialization of the cipher, a *ChaCha20ParameterSpec* object must be created and used, similar to *IvParameterSpec* and *GCMParameterSpec* above. The following creates such an object, assuming *nonce* contains the 12-byte nonce (a byte array created using *SecureRandom*) and *counter* contains the counter:

```
ChaCha20ParameterSpec chaCha20ParameterSpec = new
    ChaCha20ParameterSpec(nonce, counter);
cipher.init(Cipher.ENCRYPT_MODE, key, chaCha20ParameterSpec);
```

According to RFC 8439, *counter* is typically set to 1, which we will also do in this lab.

2.3.8 CertificateFactory class

CertificateFactory reads data in X.509 format and creates *Certificate*[6], *CertPath* or *CRL* objects from this data. 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, e.g., to initialize an RSA *Cipher*, which uses the public key in the certificate for encryption:

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

Make sure to use *RSA/ECB/OAEPPadding* for the algorithm parameter as this guarantees usage of a secure RSA padding scheme.

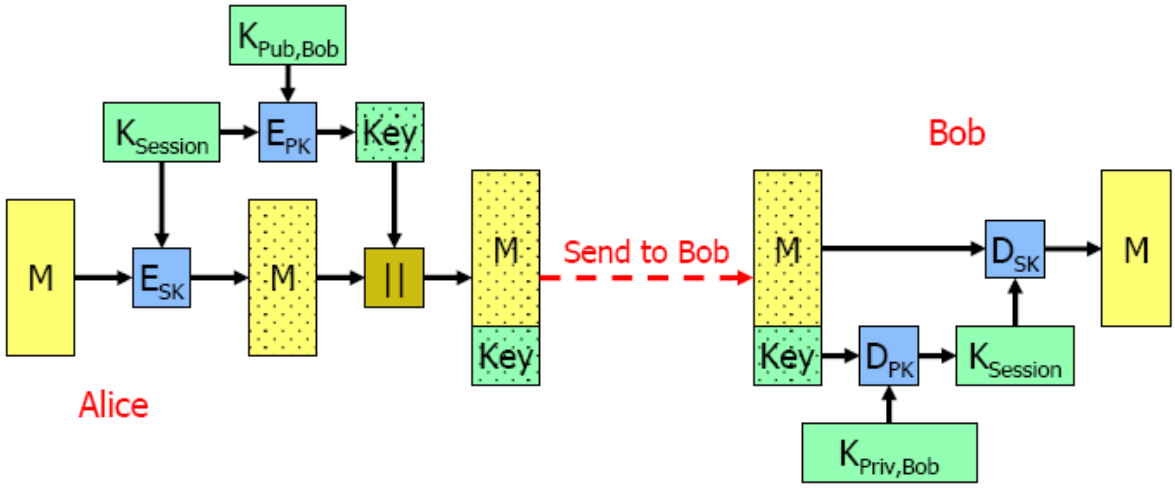
[6] Note that there are multiple *Certificate* classes in Java SE, here you have to use *java.security.cert.Certificate*.

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.

Your task is to develop a program to encrypt and integrity-protect arbitrary data using different cryptographic algorithms. The name of the program is SLCrypt (SL for Security Lab). You don't have to develop the entire program from scratch as a significant portion of it (including the entire decryption part) is already provided as a basis.

To encrypt the data, hybrid encryption is used. The concept of hybrid encryption (and decryption) is illustrated in the following image, where Alice sends an encrypted message to Bob:



With hybrid encryption, the sender Alice first encrypts a message (M, this can be any data) with a secret key cipher using a randomly generated key ($K_{Session}$). This secret key is then encrypted with a public key cipher using the public key of the recipient Bob ($K_{Pub,Bob}$) and attached to the encrypted message. The encrypted message and the encrypted secret key are then sent to Bob, who first decrypts the secret key using his private key ($K_{Priv,Bob}$). Next, Bob uses the secret key to decrypt the message.

Because raw encryption without authentication and integrity-protection is problematic (as it allows attacks against the authenticity and integrity of the encrypted data), SLCrypt uses in addition a message authentication code (MAC). This MAC is computed and appended when encrypting the data and checked for correctness when decrypting the data. In SLCrypt, a password is used as MAC key.

Furthermore, SLCrypt also supports Galois/Counter Mode (GCM). In this mode, integrity-protection is «already included» as described in section 2.3.6 and therefore, no MAC is used and no password is needed.

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

- *AES/CBC/PKCS5Padding with* key lengths 128, 192, and 256 bits. AES uses an IV of 16 bytes in all modes (expect ECB mode, which is not considered here).
- *AES/GCM/NoPadding* with key lengths 128, 192, and 256 bits. GCM uses an Auth Tag, for which a length of 128 bits is used in this lab.
- *AES/CTR/NoPadding* with key lengths 128, 192, and 256 bits. This means AES in counter mode, which is a mode to use a block cipher as the keystream generator for a stream cipher.
- *RC4* with a key length of 128 bits. Note that RC4 is no longer considered to be secure, but it's still included here as an example of a stream cipher that does not use an IV.
- *CHACHA20*, which is quite a novel stream cipher with a key length of 256 bytes. CHACHA20 uses two initialization parameters: a 12-byte nonce (which basically corresponds to an IV) and a counter (for which the value 1 is used in this lab).

In addition, your program must support the following MACs:

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

Note that the Java version available on the Ubuntu image (Java SE 11) does not support HMAC with SHA-3 (it may be supported if you are using another Java version). Therefore, you must additionally use the 3rd party Bouncy Castle CSP, which provides HMAC with SHA-3. To do this, do the following:

- Download the latest version of the Bouncy Castle CSP library from *https://www.bouncycastle.org/latest_releases.html*. The name of the file is *bcprov-jdk15on-xyz.jar*, e.g., *bcprov-jdk15on-165.jar*.
- Open */usr/lib/jvm/java-11-openjdk-amd64/conf/security/java.security* in an editor (as *root*) and add the Bouncy Castle CSP at the end of the listed CSPs, using «the next free» priority number (e.g., 13 in the example below):

  security.provider.11=JdkSASL
  security.provider.12=SunPKCS11
  security.provider.13=org.bouncycastle.jce.provider.BouncyCastleProvider

- When running the program (see section 3.2), include the CSP library in the classpath, e.g.:

```
java –cp bcprov-jdk15on-xyz.jar:. ...
```

3.1 Basis for this Lab

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

- Download *SLCrypt.zip* from OLAT.
- Move the file to an appropriate location (e.g., into 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.

3.2 Program Usage

SLCrypt is a command line program. Run it in a terminal (not in the IDE) as user *user*. SLCrypt consists of two main components (each has its own *main* method): *SLEncrypt* for encryption and *SLDecrypt* for decryption. In the following, the usage of *SLEncrypt* is explained because your task is to complete the encryption component of SLCrypt. Usage of *SLDecrypt* will be explained in section 3.5.

SLEncrypt is 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 name (relative or absolute path) of the file that contains the plaintext document to be encrypted. Note that in the remainder of this lab, the term *document* is used to identify the actual data to be encrypted, i.e., the content of the file specified with this parameter. For this parameter, you can use the test file in *SLCrypt/target/classes/data* or any other file.
- *encrypted_file* is the file name (relative or absolute path) where the encrypted (and integrity-protected) 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*.
- *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*.

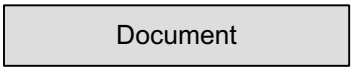
Note that the last two parameters are not needed if CGM is used (which already includes the MAC computation).

Below there 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

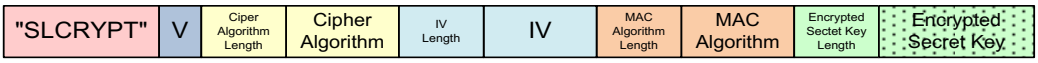
The protected files must follow a specific file format so the decryption part of SLCrypt can correctly decrypt and verify the protected files. Therefore, the file does not only contain the protected data, but also a file header with metadata. In the following, the process how a document is protected 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 v2 format (use algorithm *RSA/ECB/OAEPPadding* to do this). The result is the data structure *EncryptedSecretKey*.

EncryptedSecretKey

Next, 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*, *RC4* or *CHACHA20*.
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. In the case of CHACHA20, this field is used for the nonce.
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 v2 format, using the public key from the certificate of the recipient.

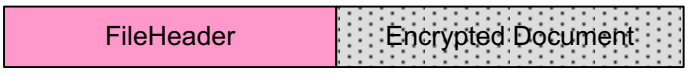
In the next step, the document is encrypted with the specified cipher algorithm using the secret key created before. Note that this is done differently depending on the cipher mode:

- 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 module IT-Sicherheit.

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 to encrypt and integrity-protect documents. The class checks the command line parameters, reads the plaintext document and the certificate with the public key from the file system, encrypts and integrity-protects the document, and stores the protected document in the file system. For the actual encryption and integrity-protection, the method *encryptDocumentStream()* in the abstract class *ch.zhaw.security-lab.slcrypt.HybridEncryption* is used. *encryptDocumentStream()* performs the complete encryption and integrity-protection of a document by calling five abstract methods in the same class. To complete the program, you have to implement these five methods in a subclass of *HybridEncryption*. For this, the class *HybridEncryptionImpl* is provided, which already contains skeletons of the methods. 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 and which are already completely implemented:

- *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()*, *isGCM()* and *isCHACHA20()* that return whether a cipher algorithm uses CBC mode or GCM or whether the cipher is a CHACHA20 cipher. Likewise, *hasIV()* returns whether a cipher algorithm uses an IV (this method also returns true if CHACHA20 is used, as the nonce used by CHACHA20 is basically an IV). 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 (such as *AES/CBC/PKCS5Padding*) and the key length (in bits) as parameters and creates 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 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 inputs and returns a corresponding *FileHeader* object. To do this, the method first creates a *FileHeader* object and next, it sets the cipher algorithm name, the IV (which must first be created randomly, using the correct length), the MAC algorithm name, and the encrypted secret key in the *FileHeader* object (using the setter methods provided by class *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 inputs. 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 (or a nonce) 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 a byte array) as additionally authenticated data before encryption. Also, use 128 bits for the length of the Auth Tag if GCM is used. If CHACHA20 is used, use the value 1 for the counter. 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* (note that *dataToProtect* corresponds to the data structure *FileHeaderEncryptedDocument* described above). 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 Testing

To test the correctness of the encryption component you developed, the decryption component *ch.zhaw.securitylab.slcrypt.SLDecrypt* is provided. It reads an encrypted document, decrypts it using the private key and verifies its integrity, and stores the decrypted document in the file system. 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 needed if another mode than GCM is used. When decryption is completed, *SLDecrypt* shows an output as illustrated below. This allows you to easily check whether the document could successfully be decrypted and whether integrity verification was successful.

```
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 with all ciphers and MAC algorithms listed at the beginning of section 3 and if all variants can be correctly decrypted with *SLDecrypt* (including correct verification of the integrity), you are ready to collect the lab points.

Lab Points

In this lab, you can get **3 Lab Points**. To get them, you have to do the following:

- Create an ASCII text file that is used as plaintext file, i.e., that contains the document to protect. You can choose any content you want, but there should be a connection to your names and your group number (an easy way to do this is to supplement the test file in *SLCrypt/target/classes/data* with your group number and the names).
- Encrypt and integrity-protect the document with your program 5 times to produce 5 different protected files. You must use the following algorithms and key lengths to create the protected files:
  - Each of the following encryption algorithms must be used exactly once: *AES/CBC/PKCS5Padding*, *AES/GCM/NoPadding*, *AES/CTR/NoPadding*, *RC4*, *CHACHA20*
  - Of the 3 files encrypted with AES, one must use a key length of 128 bits, one a length of 192 bits, and one must use a length of 256 bits. You can decide on your own which key length to use with which file.
  - The file encrypted with RC4 must use a key length of 128 bits and the file encrypted with CHACHA20 must use 256 bits.
  - With the 4 files that have a MAC (the one encrypted in GCM mode does not use a separate MAC), use each of the following 4 MAC algorithms exactly once: *HmacSHA1*, *HmacSHA256*, *HmacSHA3-256*, *HmacSHA3-512*.
  - 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 5 protected files and the MAC password. If the files can all be correctly decrypted (including successful verification of the integrity), you get 3 points.
- In addition, you must include the source code of your implementation of *HybridEncryptionImpl.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.