Java SE Security
Lab 2

1 Lab setup

Make sure you have the following items installed

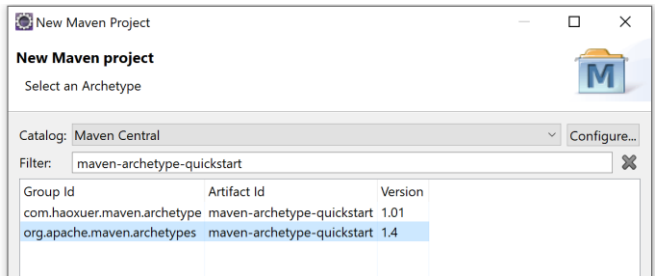
- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Eclipse Enterprise Edition or Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A suitable hex editor (HxD Hex Editor)
- A utility to extract zip files

2 Using SecureRandom as a CSPRNG

The source code for this lab is in the `Encryption-Basics-Demo` folder.

Switch to Java EE perspective.

Start with File -> New -> Maven Project. Select Next and type `maven-archetype-quickstart` in the filter. Eclipse may freeze temporarily while it attempts to filter through all the archetypes available at Maven Central. Select the entry with group id: `org.apache.maven.archetypes` and click Next



Enter in the following details and click Finish.

Group id: `com.workshop.basics`
Artifact id: `EncryptionBasicsDemo`
Version: `0.0.1-SNAPSHOT`
Package: `com.workshop.basics`

Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`. Right click on the project, select Maven -> Update Project, and click OK.

In the package, `com.workshop.basics` place the following new files:

```
RandomNumberGenDemo
BasicUtils
```

The SecureRandom class is useful for producing cryptographically strong random values by using a cryptographically strong pseudo-random number generator (CSPRNG). Randomness is an extremely important quality in cryptography, as public or private keys are expected to be truly random otherwise they are susceptible to cryptanalytic attacks. Avoid using other classes to generate random numbers such as `Math.random` or `java.util.Random` as these are based on substandard PRNG algorithms.

The list of all SecureRandom algorithms are at:
https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#SecureRandom

On Solaris, Linux, and macOS, if the entropy gathering device in java.security is set to file:/dev/urandom or file:/dev/random, then NativePRNG is preferred to SHA1PRNG. Otherwise, SHA1PRNG (the default) is preferred. On Windows, Windows-PRNG is preferred to SHA1PRNG.

Every instance of SecureRandom is created with an initial seed, which is provided from the environment if none is specified. The seed works as a base for providing random values and changes every time we generate a new value. If we create two SecureRandom objects with exactly the same initial seed, they will always generate the same set of random numbers.

We can use the random byte sequence generated by one SecureRandom object to periodically reseed another SecureRandom object, for additional cryptographic randomness.

Run the app to verify the functionality of the SecureRandom class.

To configure details for SecureRandom algorithms, check the main security configuration file:
`%JAVA_HOME%\jre\lib\security\java.security`

3 Symmetric encryption with AES in ECB mode

In the package, `com.workshop.basics` place the following new files:

```
AESECBDemo
CryptoUtils
```

Advanced Encryption Standard (AES) secret keys can only be in 3 different lengths: 128, 192 or 256 bits (16, 24 and 32 bytes, respectively). Attempting to use any other key length will result in a run time exception.
A key is simply a sequence of bytes, so any array of bytes in the specified length can be used as a key. However, in practice, the randomness of the key is extremely important for its cryptographic strength, so we will use the KeyGenerator class or the SecureRandom class for that purpose.

In addition to this, we can also use Key Derivation Functions or password hashing functions such as PBKDF2 or Argon2 to generate a secret key, which we will examine later in another lab.

We can generate secret keys for different types of symmetric encryption algorithms:
https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyGenerator

The different types of symmetric encryption algorithms available for use in Java:
https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names

A Cryptographic Service Provider (provider) refers to a package (or a set of packages) that supply a concrete implementation of a subset of the cryptography aspects of the JDK Security API. Multiple providers may be configured at the same time, and are listed in order of preference. When a security service is requested, the highest priority provider that implements that service is selected.

The list of existing providers is in the main Java security configuration file:
`%JAVA_HOME%\jre\lib\security\java.security`
Look for
`security.provider.x=yyyy`

Encryption algorithms involve the use of different types of transformations related to a particular algorithm. There are some standard set of transformations which are supported:
https://docs.oracle.com/javase/9/docs/api/javax/crypto/Cipher.html

In this example, we use the ECB mode of operation in the transformation with PKCS5 padding. This mode of operation is the simplest of all and is not recommended for production grade encryption. The plaintext is divided into blocks with a size of 128 bits. Then each block is encrypted with the same key and algorithm. Therefore, it produces the same result for the same block.

Run the app.

The encryption algorithm operates on a byte array (representing the plaintext string) and produces a byte array as an output (the encrypted sequence). The decryption algorithm is nearly identical, except that it takes an encrypted sequence and produces a plaintext output.

The first important thing to note is that attempting to decrypt an encrypted sequence (the ciphertext) with a different key than the one that was originally used to encrypt it will result in an error shown below:

```
Attempting to decrypt the ciphertext using an incorrect key
javax.crypto.BadPaddingException: Given final block not properly
padded. Such issues can arise if a bad key is used during decryption.
```

This is the case for the statement:

```
CryptoUtils.decryptMessage(ALGORITHM_TRANSFORMATION, encryptedSequence,
secondKey);
```

Comment this out and run the app again.

Now notice that just changing one of the bytes in the encrypted byte sequence (the ciphertext) will result in a plaintext with some undecipherable characters when it is decrypted. In other words, we are able to detect any kind of tampering or manipulation of the ciphertext during the process of decryption.

Finally, we run an experiment to show the length of the encrypted message for different secret key lengths. Since AES has a block size of 16 bytes, the encrypted byte array will have a length that is a multiple of 16. So if the original plain text byte array is between 1 - 15 elements in length, the encrypted output is always 16 elements: if the original is between 16 - 31 elements in length, the encrypted output is always 32 elements, and so on. This is true regardless of the secret key lengths.

4 Symmetric encryption with AES in CBC mode

In the package `com.workshop.basics` make the following changes:

`CryptoUtils-v2`

and place this new file

`AESCBCDemo`

In order to overcome the ECB weakness, CBC mode uses an additional Initialization Vector (IV) to augment the encryption process. It is considered best practice to utilize different IVs for different keys.

The code implementation for encryption / decryption is nearly identical to that of ECB, with the exception of the need to include the IV in the encryption / decryption process.

Run the app.

Notice that the length of the encrypted message (check the Base64 string version) is exactly the same for CBC and ECB mode. The use of the IV does not increase the length of the encrypted content, as

this is solely dependent on the cipher block size (which in both modes is 16-bit). IV is only used to enhance the randomness of the encryption process, which increases its cryptographic strength.

5 Symmetric encryption with AES in GCM mode

In the package `com.workshop.basics` make the following changes:

`CryptoUtils-v3`

and place this new file

`AESGCMDemo`

The Galois/Counter Mode (GCM) is secure alternative to CBC that provides authentication as well, removing the need for an HMAC SHA hashing function. It is also slightly faster than CBC because it uses hardware acceleration (by threading to multiple processor cores). It is a very widely used cipher for SSL / TLS in browsers such as Firefox.

GCM requires an IV as well, and this is called a nonce as it will only be used once for each particular key.

Run the app.

6 Encrypting and decrypting an object

In the package `com.workshop.basics` make the following changes:

`CryptoUtils-v4`

and place these new files:

```
ObjectEncryptionDemo
Developer
```

The object to be encrypted must implement Serializable.
We can use any existing encryption algorithm for the object: here we use AES/CBC

Run the app.

7 Encrypting and decrypting file contents

In the package `com.workshop.basics` place these new files:

```
FileEncryptionDemo
FileUtils
```

Open File Explorer in the project root folder. To do this, right click on the project, select Show in -> System Explorer.
In the root folder, place this file:

`plaintext.txt`

We have already seen how to write / read byte array and Strings to / from files.
Since cryptographic keys and encrypted sequences are simply byte arrays, we can simply write them to files in the usual manner.
We can also encode the byte arrays holding cryptographic keys and encrypted sequences using Base64 or Hex Strings, as we have seen before. We can then read / write these to files in the usual manner.

Run the app.

Examine the contents of the files with the binary and Base64 contents for the secret key and the cipher text respectively. You may need to use the HxD Hex editor to view the contents of the binary files properly.

Verify that the secret key and encrypted content are correctly read back from their respective Base64 file contents to reconstruct a new secret key which is used to decrypt the encrypted content successfully.

Up to this point, we have read the entire contents of the file before encrypting and then writing the encrypted content back to disk. This is feasible for small file sizes, but intractable for large files that may consume more memory than that available in the JVM. In that case, we may wish to incrementally read from that file first into a buffer, encrypt the portion in the buffer and write the encrypted content out to disk incrementally. This is done using the `cipher.update` method for incremental encryption, and then calling `cipher.doFinal` to complete the last block with the required padding size.

Next, we demonstrate the use of CipherInputStream and CipherOutputStream to perform encryption / decryption of files. We can also write the IV to the start of the encrypted file, and then read this to initialize the cipher when we perform decryption of that file. Notice that when we read from a file as an array of bytes (instead of strings), we skip the endline for each line, which results in a byte array representing a single long string that is encrypted and subsequently decrypted.

The last part of the app encrypts an object, writes the encrypted content to a file, reads back this content and decrypts it to obtain the original object. For this case, we read and write using byte arrays instead of a Base64 string, but you can easily perform this using a Base64 format as we have just done previously. If you check the contents of file, you will see some plaintext information indicating that it is a SealedObject and the particular encryption algorithm / transformation used to create it.

Run the app again. You can attempt to modify the secret key sequence and/or encrypted content sequence in the saved files to check the effect on the decryption process. We have already seen that if the key sequence that is being used to decrypt is not exactly identical to the key sequence used to encrypt, a run time exception occurs. Also, if a specific portion of the encrypted content is randomly changed, this is likely to appear as unreadable characters in the decrypted content: which allow us to detect unauthorized tampering with the encrypted contents.

8 Asymmetric encryption with RSA

In the package `com.workshop.basics` make the following changes:

`CryptoUtils-v5`

and place these new files:

`RSADemo`

For asymmetric encryption, note that we now generate a key pair using the KeyPairGenerator class. We can use the DSA algorithm here as well if we wish by getting an instance of it with that value.

Run the app.

Notice that the private key and public key have different lengths, and neither is exactly the same as the key bit size. RSA keys are made of Modulus and Exponent (either encryption or decryption exponent). The key size refers to the bits in modulus. The public key needs slightly more than the required 256 bytes to store a 2048-bit modulus because it also needs to store the exponent. The private key actually contains the entire key pair (private and public), along with all other values (p, q) that form the core of the RSA algorithm.

The main principle behind asymmetric encryption is that encryption is performed using one particular key (either private or public), while decryption is performed using the other key. Typically, encryption is performed with the public key and decrypted with the private key.

The code implementation demonstrates a run time exception occurs if any attempt is made to decrypt an encrypted sequence using the same key that was used to encrypt it.

```
javax.crypto.BadPaddingException: Decryption error
```

You can comment out the relevant statements to avoid this error from occurring.

We can write both the private and public key sequences to a file in binary or String (Base64, Hex) format and read it back again, just as we have done earlier. However, the process of reconstituting a private key from a byte sequence (using PKCS8EncodedKeySpec) is different from that of a public key (using X509EncodedKeySpec).

9 Using the BouncyCastle provider

In the package `com.workshop.basics` place these new files:

`BouncyCastleEncryptionDemo`

In addition to the list of preinstalled cryptographic providers shown in the main Java security configuration file:
`%JAVA_HOME%\jre\lib\security\java.security`

We can also use providers of our own for the Java Cryptography Extension (JCE) and the Java Cryptography Architecture (JCA). A very popular open source provider is BouncyCastle (https://www.bouncycastle.org/)

We have added the BouncyCastle dependency in the project POM:

```
<dependency>
	<groupId>org.bouncycastle</groupId>
	<artifactId>bcprov-jdk15on</artifactId>
	<version>1.60</version>
</dependency>
```

You can statically add the BouncyCastle provider by adding this line in the correct sequence in the list of existing providers in `%JAVA_HOME%\jre\lib\security\java.security`

```
security.provider.xx                                                    =
org.bouncycastle.jce.provider.BouncyCastleProvider
```

Alternatively, we can do this programmatically instead via:

```
Security.addProvider(new BouncyCastleProvider());
```

Which is what we do in this class.

Run the app.

Notice that the BouncyCastle provider (BC) is listed right at the end of the list of providers, which a very broad list of algorithms supported.

Here we are implementing encryption / decryption using AES/CBC. The code here is identical to that of the previous example with the only difference that we explicitly specify the provider name (BC) when we get a new instance of the Cipher object.

Verify that the encryption / decryption works in a similar manner as the previous example.