VPN Project General Information

In this project, you will implement a server and a client that implement a VPN – Virtual Private Network. A VPN is a secure network that runs of top of the existing Internet infrastructure. You can think of a VPN as a network of secure links, or *tunnels*, established over a public infrastructure. By connecting themselves to the VPN, clients can communicate with each other in a secure way through the tunnels.

This project includes many pieces of what is covered in this course:

* Encryption keys
* Symmetric encryption
* Public-key encryption
* Certificates

Organisation and Requirements

The project is organised in two parts. The first part consists of a number of smaller preparatory assignments, "tasks", which gradually introduce the concepts and where you implement basic functionality that you need for your VPN. Each task has its own due date.

In the second part, you take what that you have implemented in the first part, and put it together into a VPN system. This VPN consists of a client and a server implementation. The due date for submitting your implementation of the VPN system is the main due date for the entire project assignment.

The project is intended to be implemented in Java using JCA (Java Cryptography Architecture).  There are licensing restrictions for Java that you need to be aware of, and for this project, it is assumed that you follow the recommendations at KTH. See <https://intra.kth.se/it/programvara-o-system/system/oracle-java-new-license-1.862964> for more information about using Java at KTH.

Requirements

The requirements for completing the project assignments are:

* Submit solutions to all preparatory assignments before their respective due dates
* Submit a VPN implementation, consisting of a client and server, before the project's due date

More details about the requirements can be found in the instructions for each component.

Make-up

If you make a serious attempt at solving the the project before the due date, but do not pass the requirements, you will get a second chance to make up. You will receive feedback from the teachers so that you know what it is that you need to take care of. There is a separate due date for the make-up.

If you failed any of the preparatory tasks, you can also use the make-up opportunity to take care of that. However, then you miss the main point with the preparatory tasks, namely to help you prepare for the project. So do not plan for submitting preparatory tasks during the make-up – then you are not using your time efficiently, and you will have a harder time completing the project assignment.

# Software Resources

## Java

The project is intended to be implemented in Java using JCA (Java Cryptography Architecture).  There are licensing restrictions for Java that you need to be aware of, and for this project, it is assumed that you follow the recommendations at KTH. See <https://intra.kth.se/it/programvara-o-system/system/oracle-java-new-license-1.862964> for more information about using Java at KTH.

There are no particular requirements on Java version; Java 8 or later should work.

Junit 5

As part of the material for this assignment, you will get a set of [unit tests Links to an external site.](https://en.wikipedia.org/wiki/Unit_testing)that you can use to test your code. This is an important resource, so make sure to learn how to install and run the unit tests. The tests are for [Junit 5 Links to an external site.](https://junit.org/junit5/). If you are using an IDE, it is very likely that Junit 5 is pre-installed in your environment. If you want to run the tests in a terminal window, install the [Junit 5 ConsoleLauncher](https://canvas.kth.se/courses/36226/pages/junit-5-consolelauncher).

Note that the unit tests are provided as help for you to test your code. The responsibility for testing and debugging still lies with you, and the tests are provided without any further guarantees. In particular, if you pass the unit tests, it does not mean that you have passed the assignment!

### Skeleton Code and Tests

You get a substantial amount of code to work with as part of the material for this assignment. A personal git repository has been created for you on KTH GitHub (see below)  with skeleton code for your submissions, and test code with unit tests that you can use for debugging.

Download this repository, and use it as a starting point for your work. Clone the repository, and then use git for version control during the software development.

#### KTH GitHub

If you have not used KTH GitHub before, start by visiting it on the web to activate your account. Here is a link: <https://gits-15.sys.kth.se/>

There are plenty of online tutorials for git, and many IDEs have support for it.

Junit 5 ConsoleLauncher

For the project assignments, you get a set of unit tests to help you with the testing. The unit tests are for Junit 5, which comes pre-installed in many IDEs. However, if you instead want to run Junit 5 tests in a terminal window (shell, command prompt, etc), you need to install "ConsoleLauncher" for Junit 5. Here are the instructions:

1. Visit the [Junit 5 webLinks to an external site.](https://junit.org/junit5/)
2. Go to the "Platform" installation under "Latest Release".
3. Download `junit-platform-console-standalone` as a JAR file.

The file you download will be called something like `junit-platform-console-standalone-x.x.x.jar`, where `x.x.x` is a version number  (at the time of writing, the latest version is 1.9.1).

Assume that you download this file to the directory where your Junit 5 test program is. For example, say that you want to run the unit tests for the SessionKey assignment, and that you are using version 1.9.1 of ConsoleLauncher. The unit tests are in a Java file called SessionKeyTest.java. Then, to compile and execute the tests, run the following commands:

$ javac -cp .:junit-platform-console-standalone-1.9.1.jar SessionKeyTest.java  
  
$ java -jar junit-platform-console-standalone-1.9.1.jar --select-class=SessionKeyTest -cp .

Java Cryptography Architecture

Java Cryptography Architecture (JCA), together with Java Cryptography Extension (JCE), is a security framework for Java. It contains all cryptographic functionality that you need for the project assignment.

You are expected to learn about JCA through self-studies – there are no lectures on JCA. You can find plenty of good material about JCA on the Internet.

A suitable starting point is the [JCA Reference Guide.  Links to an external site.](https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html)You should mainly use it as a reference, but the [Introduction Links to an external site.](https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Introduction) briefly explains the basic structure of JCA and is therefore a good starting point.

Otherwise, you are recommended to search on the Internet for materials about JCA. Here are two popular tutorials with plenty of code examples:

* [http://tutorials.jenkov.com/java-cryptography/index.htmlLinks to an external site.](http://tutorials.jenkov.com/java-cryptography/index.html)
* <https://www.tutorialspoint.com/java_cryptography/index.htm>

Notes about learning JCA:

Task1: The SessionKey class

This task is about creating encryption keys, and converting the keys to a portable format that can, for instance, be communicated over the network. You will create a class SessionKey, which will be used in the final project assignment as a session key to encrypt traffic in a communication session.

There are two constructors for SessionKey:

SessionKey(Integer keylength)  
  
SessionKey(byte[] keybytes)

The first constructor creates a random SessionKey of the specified length (in bits). The second variant creates a SessionKey from a byte array. The byte array contains an existing key, represented as a sequence of bytes (more about this later).

The actual encryption key that is created is a symmetric key (or secret key) for AES. JSA and JCE use the class [SecretKey Links to an external site.](https://docs.oracle.com/javase/8/docs/api/javax/crypto/SecretKey.html)to represent symmetric keys. Therefore, it is a good idea to use the SecretKey class internally in your SessionKey as well.

In order to retrieve the SecretKey from a SessionKey object, there is a getSecretKey method to retrieve the corresponding SecretKey:

SecretKey getSecretKey()

In the final part of the project assignment, you will create session keys and transmit them over the network. But we cannot just take a Java object and transmit it over the network – what happens if the other party is not implemented in Java? Moreover, we do not want to transmit a symmetric key in cleartext over the network – that would put the confidentiality of the key at serious risk.

This means that we should encrypt the symmetric key before we transmit it over the network. Encryption is an operation that works on binary data – it takes a sequence of bytes and transforms it to another sequence of bytes. Hence, in order to encrypt our symmetric key, we need to first convert it to a sequence of bytes.

So, long story short, the SessionKey class should export a key as a sequence of bytes. For this, we use the getGeyBytes method:

byte[] getKeyBytes()

Assignment

Implement the SessionKey class in Java, with the methods and constructors described above.

To help you get started, there is a file with a skeleton declaration of the SessionKey class in the project repository on KTH GitHub. Use that file as the basis for your solution, and fill in the code.

Key Randomness

The quality of encryption depends a lot on the quality of the keys. What does that mean? How can you check that the key that you have generated is a "good" key? Presumably a key is generated as a random key. What constitutes a good random number generator is something that researchers have given a good deal of thought.

Surely you cannot look at a single number and decide if it is random or not? But think of a key not as a single number, but as a sequence of bits, where you decide the value of each bit by flipping a coin. So a 256-bit key is the result of flipping a coin 256 times. Then you might be able to do some analysis to determine the quality of the coin-flipping process.

On a lighter note, you may get some inspiration from these comic strips:

* [http://www.dilbert.com/strip/2001-10-25Links to an external site.](http://www.dilbert.com/strip/2001-10-25)
* [https://xkcd.com/221Links to an external site.](https://xkcd.com/221/)

You may also find this website interesting; it is about using atmospheric noise to achieve randomness: [https://www.random.org/randomness.Links to an external site.](https://www.random.org/randomness/)

Resources

There is a personal repository created for you on KTH GitHub with skeleton code and test programs. The repository is called "<username>-SessionKey", where "<username>" is your KTH user name.

Clone the repository and use it as a starting point for your work.

Testing

How can you test that your code works correctly? If it doesn't work as it should, you will get into debugging problems in the final assignment, so it is much better to spend some time now making sure that your code does what it is supposed to do.

Here are some suggestions for testing:

You could examine the key that is created. It should consist of 128 bits (or 192 or 256, depending on what key length you used). Print out the key (which is a byte array). Check that the output is indeed 128 (192, 256) bits, and that the bits appear random.

Unit Tests

We have put together a small set of basic tests that you can run on your code. It will help you check the basic functionality, and also to verify that the code can be integrated in a larger environment. There is a test class, SessionKeyTest, which uses your definition of the SessionKey class, and contains a number of test.

The tests are done for the JUnit test framework for Java. You can find the tests in your personal repository on KTH GitHub.

Submission

In order to complete your submission, you need to do two things:

* Commit your changes in git and push to your personal repository on KTH GitHub.
  + **Note:** the file should not contain any package declarations. If your IDE uses packages and inserts them for you, you need to remove them.
* Submit the URL of your personal KTH GitHub repository here, in this assignment, to notify us about the submission.
  + The URL looks something like this: "https://gits-15.sys.kth.se/IK2206HT22/<user>-SessionKey.git"

**Important**: We will not automatically check your personal KTH GitHub repository for updates. Whenever you make an update that you want us know about, you have to make a submission here. Submit the URL of your personal KTH GitHub repository, as described above.

Task2: Session Cipher

The next step is to use the session key you created in the previous assignment, and use it to create a cipher to encrypt and decrypt data. A cipher represents the parameters needed to specify encryption/decryption operations. For the purpose of encrypting a communication session using symmetric encryption, a cipher has five main components:

|  |  |
| --- | --- |
| **Parameter** | **Remark** |
| Cryptographic algorithm | Use AES |
| Mode of operation | Use CTR (counter) mode |
| Padding algorithm | Use no padding |
| Secret key | All possible lengths allowed (128, 192, and 256 bits) |
| Initial Vector (IV) | Initial counter value. Same length as an AES block (128 bits). |

## The SessionCipher class

SessionCipher is a class that performs encryption and decryption on a stream of data. The data can come from a file, for instance or, as in the final project, from a network socket.

There are two constructors for SessionCipher. The first takes a SessionKey as parameter:

SessionCipher(SessionKey key)

With this constructor, the SessionCipher itself creates the IV.

The second constructor takes a SessionKey and an existing IV as parameters. The IV is represented as a byte array.

SessionCipherer(SessionKey key, byte[] ivbytes)

There are two "getter" methods to fetch the key/IV encryption parameters from a SessionEncrypter as byte arrays:

SessionKey[] getSessionKey()  
byte[] getIVBytes()

For the actual encryption, we use Java I/O Streams to communicate with the SessionCipher in the following way. Data to encrypt is sent to the SessionCipher  via a [CipherOutputStream Links to an external site.](https://docs.oracle.com/javase/8/docs/api/javax/crypto/CipherOutputStream.html) object associated with the SessionCipher. The encrypted output from the SessionCipher goes to another [OutputStream Links to an external site.](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html)object, which can be a file, socket, etc. The following SessionCipher method sets up this arrangement, with a SessionCipher that is streaming data from a CipherOutputStream to an OutputStream:

CipherOutputStream openEncryptedOutputStream(OutputStream output)

So the idea is that after calling openEncryptedOutputStream, the caller can write data to the returned CipherOutputStream, and the data will then be encrypted and written to the output OutputStream. This may sound complicated, but hopefully it will be clearer if you look at the testing code (see below).

For decryption, there is a corresponding method to set up an InputStream.

CipherInputStream openDecryptedInputStream(InputStream input)

After calling openDecryptedInputStream, the caller can use the returned CipherInputStream to read plaintext data from it. Again, look at the testing code below.

## Assignment

Implement the SessionCipher class in Java, with the methods and constructors described above.

To help you get started, there are skeleton files with all the required declarations in your personal repository on KTH GitHub.

Tip: study JCA carefully, read online tutorials and look at examples. If you do this right, The class will only be a few lines of codes.

## Testing

The primary test is straightforward: take a piece of text and encrypt it into a ciphertext. Decrypt the cipher text, and check that the result matches. You find a Junit 5 test program in your personal repository on KTH GitHub. It also servers as an example of the SessionCipher in action.

## Submission

In order to complete your submission, you need to do two things:

* Commit your changes in git and push to your personal repository on KTH GitHub.
  + **Note:** the file should not contain any package declarations. If your IDE uses packages and inserts them for you, you need to remove them.
* Submit the URL of your personal KTH GitHub repository here, in this assignment, to notify us about the submission.

**Important**: We will not automatically check your personal KTH GitHub repository for updates. Whenever you make an update that you want us know about, you have to make a submission here. Submit the URL of your personal KTH GitHub repository, as described above.

[Previous](https://canvas.kth.se/courses/36226/modules/items/560978)[Next](https://canvas.kth.se/courses/36226/modules/items/579669)

Task3: Handshake Digest

This assignment is about hashing. We will use hashing to compute digests, or fingerprints, of data exchanged during handshake negotiations between client and server. Those digests will be used for integrity protection and for authentication.

The hashing algorithm we use is SHA-256. There is already a class in JCA that suits our needs very well, the MessageDigest class, so there is not much implementation work you need to do to support digest computations.

## HandshakeDigest Class

Implement the HandshakeDigest class. When an instance is created by calling the constructor, which does not take any parameters, it is initialised for SHA-256 hashing. The constructor looks like this:

public HandshakeDigest()

The HandshakeDigest class has two methods:

public void update(byte[] input)

The input to a hash function is a sequence of bytes that can be of any length. The hash value is computed in an iterative way, where the update method "feeds" the hash function with more input data.

public byte[] digest()

The digest method returns the final digest, which is the hash value computed over the data given through the one or more calls to the update method.

The FileDigest program

The HandshakeDigest program is a Java program that takes a file name as parameter, computes a HandshakeDigest of the (as a SHA-256 hash) of the file's content, and prints the result to the system outpu t. So in a terminal window, you would run the program in the following way:

$ java FileDigest <filename>

where <filename> denotes the name of a file. The program reads the input file as binary data, which means that you need to pay attention to how you deal with the input file in your program; you should the input file as binary data, that is, as a stream of bytes.

### Digest Output

The FileDigest program prints the hash of the file to system output. Since the hash is a binary value, it is not well-suited to be printed directly to a terminal, and therefore the program encodes the hash as text before it is printed.

There are many ways to represent binary values as text. A straight-forward way could be to print the hash as a (very large) integer value, for instance in decimal or hexadecimal notation. But that is not very efficient. If we print an integer value as a hexadecimal string, it means that we use two characters for each byte in the integer. A 4-byte integer, for instance, would be printed as 8 characters (which corresponds to 8 bytes).

Instead, the HandshakeDigest program prints the hash as a Base64-encoded string. Base64 is an encoding scheme that is widely used in applications such as Web and email for encoding binary data. If the file is called "input.txt", the output would look something like this:

$ java FileDigest input.txt   
spqYNYFYnslkBbtblz2xmXIweT8ZrV6imRxwx7yPm0o=

## Assignment

Implement the HandshakeDigest class as a subclass of MessageDigest.

Implement the FileDigest program to compute the hash of a file, as described above.

## Testing

In your KTH GitHub repository, you will find an example data file and a file with the corresponding SHA-256 hash. Run your program and make sure that the result is identical to the content of the file with the hash.

A reference implementation of FileDigest is available here as a jar file: [FileDigest.jar](https://canvas.kth.se/courses/36226/files/6039422?wrap=1)[Download FileDigest.jar](https://canvas.kth.se/courses/36226/files/6039422/download?download_frd=1). Compare the output of your program with the output of the reference implementation.  On the command line, run it like this:

$ java -jar FileDigest.jar <filename>

## Submission

Submit two files: HandshakeDigest.java and FileDigest.java.

* Commit your changes in git and push to your personal repository on KTH GitHub.
  + **Note:** the file should not contain any package declarations. If your IDE uses packages and inserts them for you, you need to remove them.
* Submit the URL of your personal KTH GitHub repository here, in this assignment, to notify us about the submission.

**Important**: We will not automatically check your personal KTH GitHub repository for updates. Whenever you make an update that you want us know about, you have to make a submission here. Submit the URL of your personal KTH GitHub repository.

Task4: Handshake Certificate

Certificates are important in secure communication protocols. They can be used to distribute public keys, and they can be used to authenticate the communicating parties to each other.

In this task, you create, sign and verify certificates. The format of certificates is defined by the ITU-T X.509 standard, and its usage on the Internet is standardised by the PKIX working group in [RFC 5280.Links to an external site.](https://datatracker.ietf.org/doc/html/rfc5280)

You will use certificates in the final project, for the handshake phase of the protocol. During the handshake phase, the client and server authenticate each other, and exchange public keys, through the use of certificates.

## OpenSSL

[Links to an external site.OpenSSL Links to an external site.](https://www.openssl.org/)is an open source implementation of the SSL/TLS protocol. The OpenSSL software also contains application programs for creating and managing certificates, and much more. OpenSSL is available on Linux, Mac and Windows. It normally comes pre-installed in Linux and Mac OS.

These instructions are for the "openssl" command line tool. There may be other tools for SSL/TLS around as well. However, if you want to use some other tool, you are on your own.

### OpenSSL in Windows

For Windows, you may need to download and install OpenSSL yourself. There are several pre-compiled binary versions for Windows. We don't recommended a particular version. Instead, look into the alternatives and pick one! Please feel free to share your experiences on the project discussion forum.

Creating certificates

All certificate operations are performed using "openssl". You can find much information on the web about how to use openssl to create certificates. These instructions give a brief introduction; for more information, search the web!

CA Certificate

In this assignment, you act both as a CA (Certification Authority), and as a user. Start with creating a certificate for your CA. A CA certificate is self-signed, while a certificate for a regular user is signed by a CA.

To create a certificate, give the "req" command to openssl. If you combine it with the "-new" option, openssl will prompt you for the information to include in the certificate. Self-signed certificates are created by giving the "-x509" option for the "req" command. So, a command to create a CA certificate interactively could look something like this:

openssl req -new -x509 ...

As you know, a certificate contains a public key. A self-signed certificate is signed with the corresponding private key. Create a key pair using the "genrsa" command to openssl (not shown here, look it up!). Store the key pair in a file, and tell openssl to read the keys from the file:

openssl req -new -x509 -key <filename> ...

If you instead specify the "-newkey" option together with "-x509", openssl will first create a new key pair for you and then create the certificate:

openssl req -new -x509 -newkey rsa:2048 -keyout privatekey.pem ...

This command creates a self-signed certificate with a 2048-bit RSA key. The key pair is stored in the file "privatekey.pem".

As a CA, you should protect your private key very carefully. So you must not distribute the file with the key pair – keep it to yourself! At the same time, you want to publish your public key so that others can use it to verify certificates you have signed. To publish your public key, you must first extract it from the file with the key pair. Use the "RSA" command for this.

openssl rsa -in privatekey.pem -pubout -out publickkey.pem

### Regular Certificate

A certificate for a regular user, which should be signed by a CA, is created in two steps. The first step is to create a certificate signing request (CSR). A CSR contains the key, the information about the subject that the user would like to have in the certificate, and directives for the CA. A CSR is created using the "req" command to "openssl", in very much the same way as for self-signed certificates. The difference is that you should not use the "-x509" option if you want to create a CSR.

The second step is to ask the CA (which is you...) to sign the certificate. To sign a certificate, use the "x509" command to "openssl" and specify the files with the CSR and the CA certificate as parameters. You may need more parameters as well; you should find this out yourself.

command lines:

(1) first, create self-signed CA (2) then, create CSR (3) use the CA to sign the CSR

openssl req -new -x509 -days 2650 -newkey rsa:2048 -keyout privatekey.pem -out CA.pem

openssl rsa -in privatekey.pem -pubout -out publickey.pem

openssl req -new -newkey rsa:2048 -keyout user\_privatekey.pem -out user.csr

openssl rsa -in user\_privatekey.pem -pubout -out user\_publickey.pem

openssl x509 -req -days 2650 -CA CA.pem -CAkey privatekey.pem -CAserial ca.srl -CAcreateserial -in user.csr -out user.pem

openssl pkcs8 -nocrypt -topk8 -inform PEM -in private-pkcs1.pem -outform DER -out private-pkcs8.der

### Rules and Guidelines for Creating Certificates

* OpenSSL can use different file formats. You should use PEM (Privacy-Enhanced Mail).
* Use proper validity dates for the certificates, both for the CA and the user certificates. The certificates should be valid at least for the duration of this course.
* Make sure to fill in reasonable values for the subject Distinguished Name (DN) in the certificate. In particular, it is mandatory that each certificate has a Common Name (CN) in its subject DN. Your name and email address must be included in the CN. Make the CN for the CA and the CN for the user different, so that you can distinguish them.
  + In the grading, the CN field will be checked against your KTH email address and your full name as it appears in Canvas. So do not use a private email address, or any alternative way of writing your full name.

## Storing Certificates in Files

There are two formats for storing certificates in files, DER and PEM. They are closely related. DER (Distinguished Encoding Rules) is a binary representation of encryption keys and certificates. PEM (which originally stands for Privacy Enhanced Mail) is a textual representation of the same information. An example of a PEM file with a certificate is shown below.

-----BEGIN CERTIFICATE-----  
MIID1jsCAr42CQDT/2gogQ2LSjANBgkqhkiG9w04AQsFd3CBqjELMAkGA1UEBhMC  
U0UxDjAMBgNVBAgMBUtpc3RhMR1wEAYDVQQHDAlTdG9ja2hvbG0xDDAKBgNVBAoM  
...  
e6VINwWxq7FE/zO3ZmKnG83xCxY45drqjBlGjRZezoEPSU+1aR5TKcgYN/7Dg5cn  
xcS3N2kVgQpq475LZFhKLuhiTnkwhD2v9uY=  
-----END CERTIFICATE-----

The first line is a header line that tells that the content is a certificate, and the last line is a footer that marks the end of the certificate. The text between the header and the footer is the DER representation of the certificate, encoded as text using [Base64 encoding Links to an external site.](https://en.wikipedia.org/wiki/Base64).

OpenSSL by default stores X509 certificates in PEM format.

## Certificate Management in Java

To deal with certificates during the handshake, we define a class called HandshakeCertificate.

It has two constructors. The first creates a HandshakeCertificate from an X509 certificate, which is read from an input stream. Hint: the X509Certificate class in Java has a constructor that creates a certificate from a byte stream in PEM or DER format.

HandshakeCertificate(InputStream instream)

The second creates a HandshakeCertificate from an X509 certificate in its encoded form (DER), given as a byte array:

HandshakeCertificate(byte[] certBytes)

Furthermore, HandshakeCertificate has two "getters" to get the X509 certificate in different formats. The first returns the X509 certificate in its encoded form as a byte array:

public byte[] getBytes()

The second returns the X509 certificate as an X509Certificate instance:

public X509Certificate getCertificate()

### Certificate Verification

The verify method cryptographically validates a certificate. It takes another HandshakeCertificate as parameter, namely the certificate for the CA that signed the certificate. The verify method checks the signature, but it does not return anything. Instead, it throws an exception if the verification fails.

public void verify(HandshakeCertificate cacert)

It is mandatory that in your declaration of the verify method, you specify exactly the exceptions that it throws. Using an unspecified exception (such as "throws Exception") is in general considered bad programming. In this case it also hides important information about what the possible results could be of calling this method.

You will find that the X509Certificate class has a corresponding method that you can use directly, so this method can be done easily as a "wrapper" around the X509Certificate method.

### Retrieving Information from a Certificate

In addition to the cryptographic validation, certificate verification involves checking the content of the certificate: validity dates, information about the user ("Subject" in X509 terminology), and more. You do not need to do that here, but you will do it in the final assignment. Therefore, to provide support for such operations, implement a few methods to retrieve various pieces of certificate information:

public String getCN()

public String getEmail()

The getCN method retrieves the subject's Common Name (CN) from the certificate, while getEmail retrieves the subject's email address. The X509Certificate class has no directly corresponding methods. Instead, you need to do some string processing to get this information from the methods that are available in the X509Certificate class.

Assignment

* Create a CA certificate
* Create a user certificate
* Implement the HandshakeCertificate class

To help you get started, there are skeleton files with all the required declarations in the Project resources repository on KTH GitHub.

Tip: study JCA carefully, read online tutorials and look at examples. If you do this right, each class will only be a few lines of codes.

## Testing

Create certificates with openssl and use your implementation of the HandshakeCertificate class to check the certificate. You can also ask a friend to check your certificates.

To help you with the testing, the project resources contains a Java file with a few Junit 5 unit tests.

## Submission

Your submission should contain the following:

1. The CA certificate, in a file called "CA.pem"
2. The user certificate, in a file called "user.pem"
3. Your java program in a file named "HandshakeCertificate.java". The file should not contain any package declarations. If your IDE inserts package declarations for you, you need to remove them.

In order to complete your submission, you need to do two things:

* Commit your changes in git and push your repository to your personal repository on KTH GitHub.
* Submit the URL of your personal KTH GitHub repository here, in this assignment, to notify us about the submission.
  + The URL looks something like this: "https://gits-15.sys.kth.se/IK2206HT22/<user>-HandshakeCertificate.git"

**Note**: We will not automatically check your personal KTH GitHub repository for updates. Whenever you make an update that you want us know about, you have to make a submission here. Submit the URL of your personal KTH GitHub repository, as described above.

# Task 5: Handshake Encryption

In this task, you use the key pair from the previous step to encrypt and decrypt information with RSA. As you can tell by the name, you will use this during the handshake phase in the VPN project.

## The HandShakeCrypto class

We define a class called HandshakeCrypto for the cryptography operations in the handshake. It can be used for encryption and decryption, with either a public key or a private key. The class has two constructors. The first takes a HandshakeCertificate as argument:

public HandshakeCrypto(HandshakeCertificate handshakeCertificate)

When a HandshakeCrypto instance is created in this way, it means that any encryption/decryption operations performed with the instance should be done using the public key from the certificate as key.

The second constructor takes a byte array as argument:

public HandshakeCrypto(byte[] keybytes)

The byte array contains a private key in PKCS#8/DER format (see below). When a HandshakeCrypto instance is created in this way, it means that any encryption/decryption operations performed with the instance should be done using the private key as key.

The class has two methods:

public byte[] encrypt(byte[] plaintext)

The encrypt method takes a plaintext as a byte array, and returns the corresponding ciphertext as a byte array.

public byte[] decrypt(byte[] ciphertext)

The decrypt method takes a ciphertext as a byte array, and returns the corresponding plaintext as a byte array.

Your HandShakeCrypto class should be able to handle certificates and key files with RSA keys of any possible length, that is, all sizes that are defined for RSA.

A note on reading private keys from files

While reading a public key from a certificate file is straight-forward in JCA/JCE, getting the private key is somewhat more involved.

Some background first: there are different ways of representing private keys. OpenSSL supports two common representations, as defined by the standards [PKCS#1 Links to an external site.](https://en.wikipedia.org/wiki/PKCS_1)and [PKCS#8 Links to an external site.](https://en.wikipedia.org/wiki/PKCS_8). Moreover, a file that stores a key can also have different formats. The file can contain the key in binary format (in which case the file format is DER, Distinguished Encoding Rules). It can also be text-encoded, in which case the format is PEM (Privacy-Enhanced Mail). So there are four combinations in total. OpenSSL, as described in these instructions, uses the PKCS#1 standard and PEM file format. Java JCA/JCE, on the other hand, uses PKCS#8 internally and therefore cannot directly read keys in the format generated by OpenSSL. There are basically two ways you can deal with this. (If you come up with a better way, please let us know!)

1. Write the Java code yourself to read key data in PKCS#1 format from a PEM file. It is not very complicated. See for example [https://github.com/mendix/SSLTools/blob/master/src/main/java/com/mendix/ssltools/PrivateKeyReader.java Links to an external site.](https://github.com/mendix/SSLTools/blob/master/src/main/java/com/mendix/ssltools/PrivateKeyReader.java) and [https://stackoverflow.com/questions/11787571/how-to-read-pem-file-to-get-private-and-public-key.Links to an external site.](https://stackoverflow.com/questions/11787571/how-to-read-pem-file-to-get-private-and-public-key)
2. This is probably easier, so this is what we recommend: Convert the OpenSSL key file from PKCS#1/PEM format to PKCS#8/DER, which has better support in JCA/JCE. You still need to process it in Java, but that can be done in one line of code. See for instance [https://stackoverflow.com/questions/20119874/how-to-load-the-private-key-from-a-der-file-into-java-private-key-object Links to an external site.](https://stackoverflow.com/questions/20119874/how-to-load-the-private-key-from-a-der-file-into-java-private-key-object). To convert a PKCS#1/PEM file to a PKCS#8/DER file, use the "pkcs8" command for openssl:

openssl pkcs8 -nocrypt -topk8 -inform PEM -in private-pkcs1.pem  -outform DER -out private-pkcs8.der

## Assignment

Implement the HandshakeCrypto class with the four methods above.

## Testing

The primary test is straightforward: take a piece of text and encrypt it into a ciphertext. Decrypt the cipher text, and check that the result matches. In your personal repository, you can find skeleton code and Junit 5 unit tests.

## Submission

In order to complete your submission, you need to do two things:

* Commit your changes in git and push your repository to your personal repository on KTH GitHub.
* Submit the URL of your personal KTH GitHub repository here, in this assignment, to notify us about the submission.
  + The URL looks something like this: "https://gits-15.sys.kth.se/IK2206HT22/<user>-HandshakeCrypto.git"

**Note**: We will not automatically check your personal KTH GitHub repository for updates. Whenever you make an update that you want us know about, you have to make a submission here. Submit the URL of your personal KTH GitHub repository, as described above.

# NetPipe Project Assignment

Overview

In this assignment, you will implement NetPipe, a network application that provides a basic communication service: it sets up a TCP connection between two hosts, and forwards data between system input/output and the TCP connection. This application is very similar to "netcat", or "nc", a popular application for testing and evaluation. Here, you will implement it in Java, and make it secure.

You can think of NetPipe as an application that sets up a secure tunnel between two computers, so that you can exchange data between them in a safe way. In this way NetPipe can serve as a general-purpose VPN (Virtual Private Network) application that allows you to connect computers across the network in a secure way. For example, in the terminal, system input and output are by default associated to the keyboard and screen, respectively. So if you run NetPipe from the command line in a terminal, you can use it to send data between two terminal windows on different hosts. If you redirect system input and output to files, NetPipe can be used as a simple file transfer program. See the assignment introduction slides for examples of NetPipe usage.

## Secure NetPipe

NetPipe starts with a handshake phase where client and server authenticate each other and establish a session key for encrypting the TCP connection. The handshake phase is based on public-key cryptography, and the client and the server start by exchanging certificates. This has two purposes:

1. **Authentication** The client and server verify the signature of the other party's certificate. As a part of the handshake protocol, each party will encrypt information with its private key. By verifying that this information can be decrypted with the (validated) public key in the certificate, each party can authenticate the other side.
2. **Session key exchange** A session key is generated, and the exchange of the key is protected by encrypting it with public-key encryption.

NetPipe uses a simple PKI with a monopoly model. The client has one CA that it trusts to sign the server's certificate, and vice versa.

## NetPipeClient and NetPipeServer

NetPipe consists of two applications, NetPipeServer for the server and NetPipeClient for the client. Let's look at the server first. This example shows the parameter to NetPipeServer:

$ java NetPipeServer --port=2206 --usercert=server.pem --cacert=ca.pem --key=server-private.der

So it takes four arguments:

* "port" – the TCP port number where the should wait for incoming TCP connections. On this port, the HandShake protocol is carried out (more about this later).
* "usercert" – the name of a file with the server's certificate (in PEM format).
* "cacert" –  the name of a file with the certificate of the CA that (is supposed to have) signed the client's certificate (in PEM format).
* "key" – the name of a file with the server's private key (in DER format).

**Note**: the command above is just an example. Your program should read the certificates and the key from whatever file names the user provides as arguments to the program. So, for example, you cannot assume that the file with the server certificate is called "server.pem". Furthermore, you cannot assume that the files are in the same directory as your program. (If you do, your program will fail the grading tests.)

This example shows the parameter of the NetPipeClient application:

$ java NetPipeClient --host=netpipeserver.kth.se --port=2206 --usercert=client.pem --cacert=ca.pem --key=client-private.der

NetPipeClient takes five arguments:

* "port" – the TCP port number to which NetPipeClient should connect.
* "host"  –the name of the host to which NetPipeClient should connect. It could also be an IP address.
* "usercert" – the name of a file with the client's certificate (in PEM format).
* "cacert" – the name of a file with the certificate of the CA that (is supposed to have) signed the server's certificate (in PEM format).
* "key" – the name of a file with the client's private key (in DER format)..

**Note:**Again, the command above is just an example. Your program should use whatever file names, host names, and port numbers the user provides as arguments to the program. If your program does not do that, it will fail the grading tests.

## The Handshake Protocol

When NetPipeClient connects to the host and port where NetPipeServer is waiting, NetPipeClient and NetPipeServer will perform the Handshake protocol. The Handshake protocol has two main purposes: to authenticate the server and the client, and to negotiate parameters for communication session.

The Handshake protocol is text-based,  which means that all data needs to be encoded as text. To transmit binary data, such as byte arrays, as text, use Base64 encoding.

The handshake protocol consists of sending a message with a set of key-value pairs, where the key and the value are strings. The first message is ClientHello, where the client introduces itself to the server:

|  |  |
| --- | --- |
| ClientHello message | |
| **Parameter** | **Value** |
| "Certificate" | Client's X.509 certificate in PEM format, first byte-encoded and then Base64-encoded. |

The server verifies the certificate, and sends its own certificate in response in a ServerHello message:

|  |  |
| --- | --- |
| ServerHello message | |
| **Parameter** | **Value** |
| "Certificate" | Server's X.509 certificate in PEM format, first byte-encoded and then Base64-encoded. |

The client verifies the certificate, and proceeds by creating the key and IV (Initialisation Vector) for the session. These are sent to the server in a "Session" message.

|  |  |
| --- | --- |
| Session message | |
| **Parameter** | **Value** |
| "SessionKey" | A byte-encoded AES session key encrypted with the server's public key, and then Base 64-encoded. |
| "SessionIV" | A byte-encoded initialisation vector for AES in CTR mode, encrypted with the server's public key, and then Base64-encoded. |

### Handshake Digest

Since most of the negotiation is carried out in plaintext (with the exception of the establishment of the SessionKey and SessionIV), so far it is not sufficiently secure. A man-in-the-middle attack could manipulate the handshake. Furthermore, there is no real authentication of the client yet, so the server cannot be sure that it is indeed communicating with the proper client.

We will address these issues by including digital signatures in the handshake, and as part of the process making sure that the client encrypts some "fresh" material with its private key, in order to ensure the client's authenticity.

### Finished Messages

In a manner similar to SSL/TLS, we add an exchange of finishing messages at the end of the handshake. For this, we introduce two more message types: "ClientFinished" and "ServerFinished".

The finished messages take two parameters: Signature and TimeStamp. Signature is the encrypted digest (HandshakeDigest) of all messages (except the finished message) that have been sent during the handshake by the party is sending the finished message. The digest is encrypted with the private key of the party sending the message. In other words, for NetPipeClient, Signature is the hash of the ClientHello and Session messages, encrypted with the private key of NetPipeClient. For NetPipeServer, Signature is the hash of the ServerHello, encrypted with NetPipeServer's private key.

The TimeStamp parameter is the current time, as a string in the format "2022-12-24 15:00:00" encoded using UTF-8 and then encrypted with the private key of the party sending the message.

|  |  |
| --- | --- |
| ClientFinished message | |
| **Parameter** | **Value** |
| "Signature" | HandshakeDigest of ClientHello and Session messages sent by the client in the handshake, encrypted with the private key of the client, and then converted to text using Base64. |
| "TimeStamp" | Current time, as a string in the format "2022-12-24 15:00:00" encoded in UTF-8, encrypted with the private key of the client, and then converted to text using Base64. |

|  |  |
| --- | --- |
| ServerFinished message | |
| **Parameter** | **Value** |
| "Signature" | Hash of the ServerHello message sent by the server in the handshake, encrypted with the private key of the server, and then converted to text using Base64. |
| "TimeStamp" | Current time, as a string in the format "2022-12-24 15:00:00" encoded in UTF-8, encrypted with the private key of NetPipeServer, and then converted to text using Base64. |

Certificates

When you use Openssl to create certificates, it will prompt you for input for the Subject field in the certificate. Answer with sensible values.

There are additional rules that apply for the Common Name (CN) part of the Subject field in the certificate:

* The CN for the CA certificate should be the string “ca-np.ik2206.kth.se” and the email address should be your email address
* The CN for the server certificate should be the string “server-np.ik2206.kth.se” and the email address should be your email address.
* The CN for the client certificate should be the string “client-np.ik2206.kth.se” and the email address should be your email address.

So, for instance, if you inspect the server certificate with this command (assuming that you store the certificate in a file called "server-certificate.pem"):

$ openssl x509 -in server-certificate.pem -text -noout

The output should contain a line that looks something like the following (together with much other information):

 Subject: C=SE, L=Stockholm, O=KTH, OU=IK2206 Internet Security and Privacy, CN=server-np.ik2206.kth.se/emailAddress=myname@kth.se

## Encrypted Session

As a result of the handshake, the client and the server share a secret key and a initialization vector. After verifying the Finished messages, the client and server switch over to session mode (in both directions!). This means that they encrypt everything they send over the TCP connection, and decrypt everything they receive. The session should use AES encryption in CTR mode, using the key and IV from the handshake (so the IV is the initial value of the CTR mode counter).

## Resources

The main resource for this assignment is what you have done already – the preparatory tasks. You should be able to use it for the security components of the project. In addition, you are provided with a complete port forwarder without security support. The port forwarder consists of the following files:

* **NetPipeClient.java** NetPipeClient without security protection
* **NetPipeServer.java** ForwardServer without security protection
* **HandshakeMessage.java** A class for encoding, decoding and transmitting key-value messages
* **Forwarder.java** A class that "switches" data between input/output streams
* **Arguments.java** A class that does command line parsing (in a rather rudimentary way)

You should be able to use the three last files without modifications. You are free to modify the first two files, and you may add more files.

### KTH GitHub Repository

There is a personal repository created for you on KTH GitHub with skeleton code and test programs. The repository is called "<username>-NetPipe", where "<username>" is your KTH user name.

Clone the repository and use it as a starting point for your work.

### Reference Implementation

When implementing a networking application, it helps a lot to have another working implementation to test against. So as a further help for your testing, you get working implementations of NetPipeClient and NetPipeServer. See [Reference Implementations.](https://canvas.kth.se/courses/36226/pages/reference-implementations)

## Specification of Algorithms

This table specifies the exact details of the various algorithms involved in the assignment.

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Specification** | **Description** |
| Session encryption | AES/CTR/NoPadding Must support 128-bit keys | Symmetric encryption for session. Use SessionKey and SessionCipher classes. |
| Handshake encryption | RSA Must support 1024-bit keys | Encryption for SessionKey and Session IV in "Session" handshake message, and for Finished messages. Use HandshakeCrypto class. |
| Handshake digest | SHA-256 | Hash algorithm for digests Finished message digests. Use HandshakeDigest class. |
| String encoding | Base64 with padding | Used for all encoding of binary data as strings: keys, IVs, certificates, and digests. |

Assignment

The assignment is to take a NetPipe implementation and make it secure by:

* Implementing the handshake protocol as described above, in NetPipeClient and NetPipeServer.
* Validating the certificates – this involves validation of the certificates that the user provides on the command line as arguments to the NetPipeClient/NetPipeServer programs, as well validation of the certificates that NetPipeClient and NetPipeServer exchange over the connection.
* Verifying the integrity of the handshake, and authenticating the two parties, by computing digests of handshake messages and encrypting those with private keys.
* Securing the session between NetPipeClient and ForwardServer by encrypting the communication with symmetric-key encryption, using the key material created during the handshake.

## Requirements

The basic requirement is that the code you submit should conform to the specification in the instructions. In particular:

* NetPipeClient and NetPipeServer should be executed in the way shown in the instructions.
* The message format in the handshake should be **exactly** as shown in the instructions. The Parameter name should be written exactly as shown in the tables – every single character. So it should be exactly "SessionKey", for instance. Not "session key", "sessionKey", etc.
* You should perform proper validation of input parameters to the programs – this includes verifying the CA and client/server certificates that the user provides on the command line.
* You should do proper error handling during the handshake phase – what happens if one party sends illegal arguments, for instance?

### Testing

When implementing a networking application, it helps a lot to have another working implementation to test against. So as a further help for your testing, you get working implementations of NetPipeClient and NetPipeServer with security.

**Make sure to test your implementation against the reference implementation before you submit.** If your submission does not work with the reference implementation, it means that your implementation is not done according to the instructions, and you will not pass.

## Submissions

Your submission should include **all Java source files** (but no class files) that are part of your implementation. You should be using the classes you did during the previous assignments, so in that case you should add your Java files from Task 1 to 5 to your NetPipe repository. The repository should also contain the certificates for the CA, the client and the server, as well as the client's and the server's private keys. Use the following file names:

* **client.pem**The client's certificate in PEM file format.
* **client-private.der**The client's private key in DER file format.
* **server.pem**The server's certificate in PEM file format.
* **server-private.der**The server's private key in DER file format.
* **ca.pem**The CA's certificate in PEM file format.

tO submit, do the following:

* Add your files and commit your changes in git and push to your personal repository on KTH GitHub.
  + **Note:** the file should not contain any package declarations. If your IDE uses packages and inserts them for you, you need to remove them.
* Submit the URL of your personal KTH GitHub repository here, in this assignment, to notify us about the submission.
  + The URL looks something like this: "https://gits-15.sys.kth.se/IK2206HT22/<user>-NetPipe.git"

**Important**: We will not automatically check your personal KTH GitHub repository for updates. Whenever you make an update that you want us know about, you have to make a submission here. Submit the URL of your personal KTH GitHub repository, as described above.

Commands about creating certificates:

Commands used to create certificate and key:

1. generate a keypair for CA console

openssl genrsa -out cakey.pem 2048

2. generate a self-signed certificate for CA

openssl req -x509 -new -nodes -key cakey.pem -days 365 -out ca.pem

3. generate a keypair for server

openssl genrsa -out serverkey.pem 2048

4. generate a CSR for server

openssl req -new -key serverkey.pem -out server.csr

5. generate a certificate for server, signed by CA

openssl x509 -req -CA ca.pem -CAkey cakey.pem -in server.csr -out server.pem -CAcreateserial

6. generate a keypair for client

openssl genrsa -out clientkey.pem 2048

7. generate a CSR for client

openssl req -new -key clientkey.pem -out client.csr

8. generate a certificate for client, signed by CA

openssl x509 -req -CA ca.pem -CAkey cakey.pem -in client.csr -out client.pem -CAcreateserial

9. convert server key to der format

openssl pkcs8 -nocrypt -topk8 -inform PEM -in serverkey.pem -outform DER -out server-private.der

10. convert client key to der format

openssl pkcs8 -nocrypt -topk8 -inform PEM -in clientkey.pem -outform DER -out client-private.der

Testing:

java NetPipeServer --port=2206 --usercert=server.pem --cacert=ca.pem --key=server-private.der

java NetPipeClient --host=netpipeserver.kth.se --port=2206 --usercert=client.pem --cacert=ca.pem --key=client-private.der

cd out/production/wenjian-NetPipe-main/