From LU Verteilte Systeme (184.167)

Lab3: Lab3

General Remarks

- We suggest reading the following materials before you start implementing:
  - Chapter 9 - Security from the book *Distributed Systems: Principles and Paradigms (2nd edition)*
  - **Java Cryptography Architecture (JCA) Reference Guide**: Tutorial about the Java Cryptography Architecture (JCA).
- Group work is **NOT** allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the forum) are allowed but the code has to be written alone.
- Be sure to check the **Hints & Tricky Parts** section! For this assignment we have put lots of helpful code snippets there!

Submission

- You must upload your solution using the **Teaching Tool** before the submission deadline: **11.01.2011, 18:00 CET.** - the deadline is hard! You are responsible for submitting your solution in time. If you do not submit, you won't get any points!
- Do not confuse our lab server (`pasta.dslab.tuwien.ac.at`) with the **Teaching Tool**. The lab server is just for testing purposes. We cannot grade any solutions uploaded there.
- Upload your solution as a **ZIP** file. Please submit the sources of your solution, the build.xml file and the lib directory containing the Bouncy Castle library.
- Your submission must compile and run in our lab environment. Therefore, complete the provided ant file inside our **Project Template**.
- Test your solution extensively in our lab environment. It'll be worth the time.
- Before the submission deadline, you can upload your solution as often as you like. Note that any existing submission will be **replaced** by uploading a new one.
- Please make sure that your upload was successful (i.e., you should be able to download your solution - as the tutors will do during the interview).

Interviews

- After the submission deadline, there will be a mandatory interview (Abgabegespräch). You must register for a time slot to the interviews using the **Teaching Tool**.
- You can do the interview only if you submitted your solution before the deadline!
- The interview will take place in the **DSLab**. During the interview, you will be asked about the solution that you uploaded (i.e., **changes after the deadline will not be taken into account!**). In the interview you need to explain your code, design and architecture in detail.
- Remember that you can do the interview **only once**!

Important: Lab 3 consists of three independent stages. If you want to get all points for this assignment, you will have to implement all of them. If you are satisfied with less, you can leave one or more of them unimplemented.

Differences To Lab 1

- Lab 3 is based on your Lab 1 solution. You will not be penalized again for any errors in your Lab 1 solution, as long as these errors have no impact on Lab 3. You need to fix any errors which somehow hinder Lab 3. For Part 1, it must be possible to log in a user (`!download` and `!list` will also be tested, but require some adaptions anyway). Concerning Part 2, the most simple command in the Client/Proxy communication is `!credits` -- at least this command has to work for testing purposes. Part 3 requires working `!login` and `!list` commands.
- Some arguments for the three applications are no longer specified using command-line arguments, but instead are given using configuration files. Check the **Project Template** section for more details!
- The keys in `user.properties` have changed.

- If you implement stage 1, you may now need to contact mulitple fileservers for the implementation of the `!list` and `!download` commands. Making an upload may now change the usage statistics of multiple fileservers.
- If you implement stage 2, the client application will now need to process and modify the commands before they are sent to the Proxy.
- Because authentication is already included in the establishment of a secure channel (stage 2), the `!login` command now requires only one argument: the username. The password is not needed anymore because, basically, users now identify themselves with their own private key. Likewise, the `user.properties` file no longer needs to be consulted for the user's password.

Description

In this assignment you will learn:
- how to implement a replication mechanism based on Sockets
- cryptographically securing the communication between clients and Proxy
- asymmetric and symmetric cryptographic algorithms
- how to verify message integrity
- using the Java Cryptography Architecture (JCA) API

Overview

Lab 3 is devided into three independent parts. The first part aims at providing an `!upload` command to the client and adds a more sophisticated replication meachanism, whereas Stage 2 and Stage 3 both introduce security. You should refactor and extend your solution for Lab 1 to accommodate the new requirements (please take a look at the **Project Template**).

Stage 1 (5 points): Your solution for Lab 1 provides a way to download files made available through our fileserver network. However, Lab 1 did not specify a way to upload files to the network. Therefore, Stage 1 of this Lab now introduces the `!upload` command. By uploading files, clients can increase their amount of credits in the system. To make things more interesting, a file is not uploaded to every single fileserver -- our fileservers won't be fully replicated any longer. Obvioulsy, we will also need to adapt our implementations of the `!list` and `!download` command to fit this situation. Gifford's scheme shall be used to keep the access to the replicas consistent.

Stage 2 (10 points): This stage secures the communication between the client and the Proxy by implementing a secure channel and mutual authentication using public-key cryptography. In our case, the secure channel will protect both parties against interception and fabrication of messages. Note that the common definition of a secure channel additionally implies a protection against modification. The client communication will remain vulnerable in this regard; however, Stage 3 will show how to address this issue concerning the communication between the proxy and the fileservers.

The first part of setting up our secure channel is to mutually authenticate each party (i.e., every party needs to prove its identity). In this assignment we will authenticate using the well-known challenge-response protocol (which is also discussed in the lecture). To subsequently ensure confidentiality of the messages after the authentication, we will use secret-key cryptography by means of session keys. Note that the session key is shared only between one specific client and the Proxy (and not with other clients). It is generated and exchanged during the authentication phase (i.e., if and only if the authentication is successful, both parties will know how to continue communication securely). This approach ensures that the user and the Proxy are having a confidential communication and that each party is who it claims to be.

Stage 3 (5 points): The last part of this assignment will show you how to verify that a message reaches it receiver unmodified using the JCA. For the sake of simplicity, we will add this feature solely to the otherwise unsecured communication between the Proxy and the fileservers. Here, communication is not protected against interception. By using a Message Authentication Code (MAC) and appending it to each message, the receiver can check whether the message has been modified on the way through the channel.

Installation and Static Registration of the Bouncy Castle Provider

For the second and third part we will use the **Bouncy Castle** library as a provider for the Java Cryptography Extension (JCE) API, which is part of JCA. The Bouncy Castle provider (JDK 1.6 version) is already part of our **Project Template**. Please stick to the provided version as this is the one used in our lab environment.

The provider is configured as part of your Java environment by adding an entry to the `java.security` properties file (found in `$JAVA_HOME/jre/lib/security/java.security`, where `$JAVA_HOME` is the location of your JDK distribution). You will find detailed instructions in the file, but basically it comes down to adding this line (but you may need to move all other providers one level of preference up):

```
security.provider.1 = org.bouncycastle.jce.provider.BouncyCastleProvider
```

Where you actually put the jar file is mostly up to you, but the best place to have it is in `$JAVA_HOME/jre/lib/ext`.

The installation of a custom provider is explained in the **Java Cryptography Architecture (JCA) Reference Guide** in detail.

Note: If you get "java.lang.SecurityException: Unsupported keysize or algorithm parameters" or "java.security.InvalidKeyException: Illegal key size" exception while using the Bouncy Castle library, then check this **hint**.

Project Template

To help you start with the assignment, we made a **template project** which you should use. The project contains an ant build file and four directories: `files`, `keys`, `lib`, and `src`. The `lib` directory contains the Bouncy Castle library which is automatically added to the runtime classpath. The `keys` directory contains all the keys required to test your implementation.

Each private key used in the client-proxy communication is encrypted with same password: 12345. The secret key used in the fileserver network is not password protected at all.

In the `src` directory you will find configuration files named `client.properties`, `fileserver.properties` and `proxy.properties` which are used by the client, fileserver and proxy applications, respectively. In contrast to Lab 1, some information required by the three applications no longer can be read in as command line arguments (this is mainly because we need a better access to those parameters to be able to test your solution in an appropriate way, and because additional information is required by the security stages). Instead, the properties files provide this information now. In these files you only have to adjust the port properties according to the **Lab Port Policy**. All other configuration properties have meaningful default values and you do not need to adapt them. In particular, do not change the names of the properties defined!

Reading of the configuration file in an application is done by using the `Config` class also found in the `src` directory. This class is initialized by giving the name of the `.properties` file it should use to read the configuration from (e.g. `new Config("client")`). The `getString()` and `getInt()` methods will read the configuration property and return its value as the appropriate type (e.g. `config.getInt("proxy.tcp.port")`). You **have to change** your applications to read the values from their respective configuration files.

However, some information are still provided by means of command line arguments:

Fileserver:
- `sharedFolder`: the directory that contains all the files clients can download or have uploaded.
- `tcpPort`: the port to be used for instantiating a ServerSocket (handling the TCP requests from the Proxy)

Client:
- `sharedFolder`: the directory to put downloaded files or to upload files from.

The Proxy no longer expects any command line arguments.

The project also contains a new ant build file (build.xml), in which you only have to adjust the class names and ports. Note that it's absolutely required that we are able to start your programs with these predefined commands!

Stage 1 - Replication and Consistency (5 points)

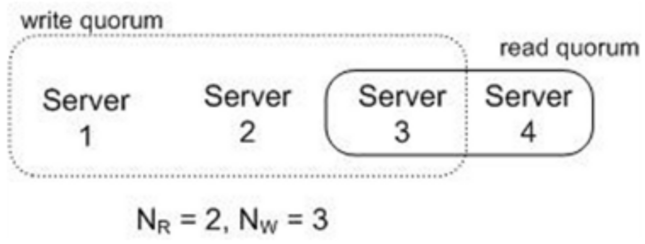
Gifford's scheme shall be used to keep the access to the replicas consistent. To support this, version numbers shall be assigned to the text files known from Lab 1 (exactly one version number for one file). Gifford's scheme works as follows: To read a file that has N replicas a so-called read quorum is assembled, i.e. a subset of N_R fileservers among these N replicas. ~~These N_R may be chosen~~

~~arbitrarily.~~ To modify a file a write quorum of N_W fileservers is built in the same way as the read quorum.

The two values N_R and N_W need to satisfy two constraints:

1. **$N_R + N_W > N$**
2. **$N_W > N/2$**

The following figure gives an example of Gifford's scheme. The first three fileservers have been selected as the write quorum, the last two fileservers have been selected as a read quorum.



If the above constraints are satisfied then at least one fileserver of the read quorum will lead to recent data. It is the Proxy's task to implement the algorithm correctly.

To support the upload of files, the client accepts a new interactive command:

- `!upload <filename>`

  Sends specified file from the private `sharedFolder` to the Proxy.
  E.g.:
  ```
  >: !upload file1.txt
  File successfully uploaded. You now have 600 credits.
  ```

The Proxy will then replicate the file automatically according to the protocol defined above. To this, it requires the name of the file and the actual file content. For we want to foster uploads, our clients get rewarded for uploading: Each uploaded file is worth twice the amount of bytes of the file in credits. For example, if Alice has 500 credits an decides to upload a file with 50 bytes, she will have 600 credits after the upload. Naturally, the Proxy application is the one to decide upon the actual filesize. You can simply use the `String.length()` method for this. Inform the Client about her/his new credit status.

When receiving an upload request, the Proxy asks N_R fileservers for the current version number of the respective file. Choose the N_R fileservers with the lowest usage. The introduction of a versioning mechanism is new to Lab 3. The version of a file depends on the amount of uploads that 'changed' the file. Accordingly, an already existing file has version 0 in the beginning; a file that has just been uploaded and didn't exist before has version 1. Another upload of a file with the same name will increase the version to an amount of 2 and so on. Fileservers do not even check if the file has been changed at all or is completely different from the one that existed before. That is, the used versioning approach is not sophisticated at all and should be straightforward in its implementation.

After consulting N_R fileservers, the Proxy now knows the most recent version of the file (which is the highest version returned, obviously). By adding this version + 1 to the request, it can now initialize the actual upload to N_W fileservers. Again, these fileservers are the N_W fileservers with the lowest usage. After this, it updates the usage statistics of these fileservers (i.e., the filesize is added to the current values) and informs the client of the successful operation.

Fileservers simply need to store the file in their `sharedFilesDir` as you did when downloading files to the client. Moreover, since each fileserver keeps track of the current version of each hosted file, it has to update the version to the value sent by the Proxy. It is enough to keep version information in memory; that is, versions do not need to survive fileserver restarts.

You will also need to adapt your former solution regarding the `!download` and `!list` commands: If the Proxy receives a download request, it asks N_R fileservers (the N_R with the lowest usage) for the current version of this file and obtains it from the one with the highest version. If multiple fileservers provide this version, the fileserver with the lowest usage is chosen. Do not forget to update the usage of the chosen fileserver afterwards. Concerning the `!list` command, contact all fileservers and send a merged list back to the client.

Concerning the implementation of Gifford's scheme, you may assume that no additional fileservers will join the network or old ones disappear after the first client has made an upload. Moreover, you do not need to concern synchronisation of concurrent client requests in this regard.

Stage 2 - Secure Channel (10 points)

The first stage of establishing a secure channel is a mutual authentication. We will authenticate a client and the Proxy using public-key cryptography. This type of authentication is explained in the book *Distributed Systems: Principles and Paradigms (2nd edition)*, page 404, Figure 9-19. However, we also describe the principles in detail below.

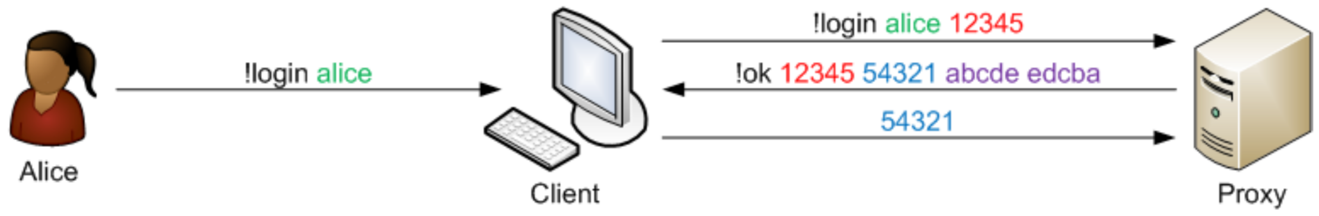
Please note that in this assignment you are not allowed to use `javax.crypto.CipherInputStream` and `javax.crypto.CipherOutpurStream`. Instead you should encrypt and decrypt all messages yourself: Use the `Socket.getInputStream()` and `Socket.getOutputStream()` methods together and decorate these streams with a `java.io.BufferedReader` or `java.io.PrintWriter`, respectively. Note that by encrypting messages, it now becomes possible to send and receive Strings containing new line characters (as you were required to do in Lab 1) without any further ado in a single line of text.

In order to get rid of any special characters which are unsuitable for transmission as text you should **encode your already encrypted messages using Base64 before transmission** (see **code snippets** below). However, do not forget to Base64 decode the messages after receiving, otherwise the decryption of the messages will of course fail.

We highly recommend to hide the security aspects from the rest of your application as much as possible. Note that plain Sockets or encrypted channels have many communalities, e.g., the are both used to send and receive messages. Therefore, it may be a good idea to define a common interface that abstracts the details of the underlying implementation away. You should use the **Decorator pattern** to add further functionalities step by step: For example, you could write a `TCPChannel` that implements your `Channel` interface and provides ways to send and receive `String`s over a Socket. Next, you could write a `Base64Channel` class that also implements your `Channel` interface and encodes or decodes `String`s using Base64 before passing them to the underlying `TCPChannel`. Following this approach may simplify your work in Stage 2 and 3.

Authentication Algorithm

Note: You should implement the authentication algorithm (including the syntax of messages) exactly as described here! Failure to do so may result in losing points. The reason for this is that we will test your assignment using a modified client which relies on the protocol being **exactly** as described. Do yourself and the tutors a favor and implement the protocol as described.



The authentication algorithm consists of sending three messages:

1st Message: The first message is a `!login` request (analogous to the `!login` command from Lab 1). The syntax of the message is: `!login <username> <client-challenge>`. This message is sent by the client and is encrypted using RSA initialized with the Proxy's public key.

- The `username` is the name of the user who wants to authenticate and log in (e.g. alice) and is passed to the Client by the user using the `!login <username>` command.
- The `client-challenge` is a 32 byte random number, which the client generates freshly for each request (see **code snippets** to learn how to generate a secure random number). Encode the challenge separately using Base64 before encrypting the overall message.
- Initialize the RSA cipher with the `"RSA/NONE/OAEPWithSHA256AndMGF1Padding"` algorithm.
- As stated above, do not forget to encode your overall ciphertext using Base64 before sending it to the Proxy.

2nd Message: The second message is sent by the Proxy and is encrypted using RSA initialized with the user's public key. Its syntax is: `!ok <client-challenge> <proxy-challenge> <secret-key> <iv-parameter>`.

- The `client-challenge` is the challenge that client sent in the first message. This proves to the client that the Proxy successfully decrypted the first message (i.e., it proves the Proxy's identity).
- The `proxy-challenge` is also a 32 byte random number generated freshly for each request by the Proxy.
- The last two arguments are our session key. The first part is a random 256 bit secret key and the second is a random 16 byte initialization vector (IV) parameter.
- **Every argument** has to be encoded using Base64 before encrypting the overall message!
- The ciphertext is sent Base64 encoded again.

3rd Message: The third message is just the `proxy-challenge` from the second message. This proves to the Proxy that the client successfully decrypted the second message (i.e., it further proves the client's identity). This message is the first message sent using AES encryption.

- Initialize the AES cipher using the `<secret-key>` and the `<iv-parameter>` from the second message. Details about these parameters are out of scope of this lab - you will learn about them in a Cryptography lecture.
- Use the `"AES/CTR/NoPadding"` algorithm for the AES cipher.
- Again, encrypt and encode the message before sending it.

The final end product of the authentication is an AES-encrypted secure channel between the client and the Proxy, which is used to encrypt all future communication. You MAY NOT send any message unencrypted (between a client and the Proxy) in this assignment. You MAY NOT send any messages besides the first two authentication messages (as described above) using the RSA encryption (i.e., the RSA encryption is strictly for the authentication part).

To generate a random 32 byte numbers to be used for challenges and random 16 bytes numbers to be used for IV parameter, you should use the `java.security.SecureRandom` class and its `nextBytes()` method as shown in the **Hints & Tricky Parts** section. Base64 encodinging them is required because this method could return bytes which are unsuited to be inserted in a text message. Same holds true for random secret keys in the AES algorithm (see the **Hints & Tricky Parts** section on how to generate them). That is: Always encode your challenges, IV parameters and secret keys separately in your message using Base64. This message is then encrypted and gets Base64-encoded again before sending it.

Client Application Behavior

When a client application is started, it doesn't know which user will try to log in. Therefore the client application will need to process the `!login` request before it is sent to the Proxy to find out which user is trying to log in and read his private key (used for decrypting the `!ok` message). Make sure a private key for this user does exist, otherwise, print an exception. The processing of the `!login` command also includes appending the `client-challenge`.

There are two new configuration properties in the `client.properties` file, which the client application will need for the authentication phase:

- the `keys.dir` property denotes the directory where to look for the user's private key (named `<username>.pem`),
- and the `proxy.key` property defines the file from where to read Proxy's public key.

Proxy Application Behavior

The Proxy application should read its private key during the startup time. The user's public keys are read when the Proxy receives a log in request. The user is said to be online when the authentication phase is successfully completed.

There are also two configuration properties in the `proxy.properties` file, which the Proxy will need for the authentication phase:

- the `keys.dir` property denotes a directory where to look for user's public keys (named `<username>.pub.pem`),
- and the `key` property telling where to read the Proxy's private key.

Stage 3 - Message Integrity (5 points)

Note: You should implement the syntax for private messages exactly as described here, for the same reasons as discussed above. Furthermote, make sure that the UDP messages your fileservers send to the Proxy actually adhere to the `!alive <tcpPort>` format (this has to do with the way we will test this stage during your interview).

In this part of the assignment we will add an integrity check for the TCP messages exhanged between the Proxy and the Fileservers. However, the communication won't get encrypted - the implementation will only make sure that a third party cannot tamper with a message unnoticed. To this, it relies on Message Authentication Codes (MACs).

Whenever the Proxy sends an TCP request to a fileserver and whenever a fileserver responds, the application needs to append a HMAC (a hash MAC). To generate such a HMAC you should use SHA256 hashing (`"HmacSHA256"`) initialized with a secret key shared between the Proxy and all fileservers in the network. See the **Hints & Tricky Parts** on how to read in the shared secret key or create and initialize HMACs. After the HMAC is generated, it should be encoded using Base64. Add the HMAC to the original message by making it the first argument and Base64 encode the resulting message before sending it.

To verify the integrity of the message, the receiver generates a new HMAC of the received plaintext to compare it with the received one. In case of a mismatch, the behaviour of a fileserver should be as follows: The respective message is printed to the standard output and the Proxy is informed about the tampering, using the same channel the message was received. When the Proxy receives this report or notices a message from a fileserver itself was changed, the Proxy repeats the respective operation. Like a fileserver, whenever it receives unverified messages, the Proxy has to print their content to the console.

Note that the described behaviour now requires the fileservers to respond to every request that arrives from the Proxy. Nevertheless, stick to the desgin of Lab1 where the TCP connection was closed after a single request/response cycle, even though an unverified message has been received.

Lab Port Policy

Since it is not possible to open `ServerSocket` on ports where other services are already listening for requests, we have to make sure each students uses its own port range.
So if you are testing your solution in the lab environment (i.e., on the lab server) you have to obey the following rule: you may only use ports between **10.000 + dslabXXX * 10** and **10.000 + (dslabXXX + 1) * 10 - 1**. So if your account is dslab250 you may use the ports between 12500 and 12509 inclusive. Note that you can use the same port number for TCP and UDP services (e.g., it is possible to use TCP port 12500 and UDP port 12500 at the same time).

Regular expressions

We provide some regular expressions you can use to **verifiy that the messages you exchange between the three applications are well-formed**. This is important because we will test your program against own code. We recommend to use Java assertions for this. The provided build file enables them automatically.

```
final String B64 = "a-zA-Z0-9/+";

// stage II
// Note that the verified messages still need to be encrypted (using RSA or AES, respectively)
and encoded using Base64!!!

// login request send from the Client to the Proxy
String firstMessage = ...
assert firstMessage.matches("!login \\w+ ["+B64+"]{43}=") : "1st message";
// the Proxy's response to the client
String secondMessage = ...
assert secondMessage.matches("!ok ["+B64+"]{43}= ["+B64+"]{43}= ["+B64+"]{43}= ["+B64+"]{22}==")
: "2nd message";
// the last message send by the Client
String thirdMessage = ...
assert thirdMessage.matches("["+B64+"]{43}=") : "3rd message";

// stage III

// UDP message that is send from the fileserver to the Proxy in a recurring manner
String udpMessage = ...
assert message.matches("!alive 1[0-9]{4}");

// messages beeing exchanged between Proxy and fileservers before the final Base64 encoding
```

```
String hashedMessage = ...
~~assert message.matches("["+B64+"]{43}= [\\s^\\s]+");~~
assert message.matches("["+B64+"]{43}= [\\s[^\\s]]+");
```

Hints & Tricky Parts

"java.lang.SecurityException: Unsupported keysize or algorithm parameters" or "java.security.InvalidKeyException: Illegal key size"

You will need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from **Java SE Download** page (last download link). The installation explanation can be found in README file, but basically you will just have to copy `local_policy.jar` and `US_export_policy.jar` into `$JAVA_HOME/jre/lib/security/` directory.

The decrypted messages are in gibberish and no exception is thrown.

Possible reasons:

1. You forgot to decode the decrypted message back from Base64 format.
2. Wrong value was used for initialization vector (IV) parameter when initializing the AES cipher.
3. You didn't use `Cipher.DECRYPT_MODE` when initializing the AES cipher

java.security.NoSuchProviderException: No such provider: BC

This exception is thrown when the Bouncy Castle library is not properly installed. Please recheck that you correctly did all the steps from the **Installation and Static Registration of the Bouncy Castle Provider** section and actually use the so configured JDK.

java.security.NoSuchAlgorithmException: No such algorithm: "..."

The most probable cause is that you misspelled the name of the algorithm. Please check that name of the algorithm is same as mentioned in this assignment.
This exception is also thrown when the Bouncy Castle provider is not properly installed. If the name of the algorithm is correct, than you should recheck that you correctly did all the steps from the **Installation and Static Registration of the Bouncy Castle Provider** section.

java.security.InvalidKeyException: no IV set when one expected

You forgot to initialize the AES cipher using the initialization vector (IV) parameter. The IV parameter is mandatory for the `"AES/CTR/NoPadding"` algorithm. See **code snippets** on how to correctly initialize the AES cipher.

Helpful Code Snippets

All code snippets are just examples and omit all exception handling for clarity. This does not mean that you should not do exception handling in your lab solution!

- **How to encode into and decode from Base64 format?**
- **How to read a PEM formatted RSA private key?**
- **How to read a PEM formatted RSA public key?**
- **How to generate a secure random number?**
- **How to generate an AES secret key?**
- **How to initialize a cipher?**
- **How to read the shared secret key?**
- **How to create a hash MAC?**
- **How to verify a hash MAC?**

How to encode into and decode from Base64 format?

```
import org.bouncycastle.util.encoders.Base64;
```

```
[...]

// encode into Base64 format
byte[] encryptedMessage = ...
byte[] base64Message = Base64.encode(encryptedMessage);

// decode from Base64 format
encryptedMessage = Base64.decode(base64Message);
```

How to read a PEM formatted RSA private key?

```
import java.security.KeyPair;
import java.security.PrivateKey;
import org.bouncycastle.openssl.PEMReader;
import org.bouncycastle.openssl.PasswordFinder;

[...]

String pathToPrivateKey = ...
PEMReader in = new PEMReader(new FileReader(pathToPrivateKey), new PasswordFinder() {
          @Override
          public char[] getPassword() {
               // reads the password from standard input for decrypting the private key
               System.out.println("Enter pass phrase:");
               return new BufferedReader(new InputStreamReader(System.in)).readLine();
          }
     });
KeyPair keyPair = (KeyPair) in.readObject();
PrivateKey privateKey = keyPair.getPrivate();
```

How to read a PEM formatted RSA public key?

```
import java.security.PublicKey;
import org.bouncycastle.openssl.PEMReader;

[...]

String pathToPublicKey = ...
PEMReader in = new PEMReader(new FileReader(pathToPublicKey));
PublicKey publicKey = (PublicKey) in.readObject();
```

How to generate a secure random number?

```
import java.security.SecureRandom;

[...]

// generates a 32 byte secure random number
SecureRandom secureRandom = new SecureRandom();
final byte[] number = new byte[32];
secureRandom.nextBytes(number);
```

How to generate an AES secret key?

```
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

[...]

KeyGenerator generator = KeyGenerator.getInstance("AES");
// KEYSIZE is in bits
generator.init(KEYSIZE);
```

```
SecretKey key = generator.generateKey();
```

How to initialize a cipher?

```
import javax.crypto.Cipher;

[...]

// make sure to use the right ALGORITHM for what you want to do
// (see text)
Cipher crypt = Cipher.getInstance(ALGORITHM);
// MODE is the encryption/decryption mode
// KEY is either a private, public or secret key
// IV is an init vector, needed for AES
crypt.init(MODE, KEY [,IV]);
```

How to read the shared secret key?

```
import java.io.FileInputStream;
import java.security.Key;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.util.encoders.Hex;

[...]

byte[] keyBytes = new byte[1024];
String pathToSecretKey = ...
FileInputStream fis = new FileInputStream(pathToSecretKey);
fis.read(keyBytes);
fis.close();
byte[] input = Hex.decode(keyBytes);
// make sure to use the right ALGORITHM for what you want to do
// (see text)
Key key = new SecretKeySpec(input,ALGORITHM);
```

How to create a hash MAC?

```
import java.security.Key;
import javax.crypto.Mac;

[...]

Key secretKey = ...
// make sure to use the right ALGORITHM for what you want to do
// (see text)
Mac hMac = Mac.getInstance(ALGORITHM);
hMac.init(secretKey);
// MESSAGE is the message to sign in bytes
hMac.update(MESSAGE);
byte[] hash = hMac.doFinal();
```

How to verify a hash MAC??

```
import java.security.MessageDigest;
import javax.crypto.Mac;

[...]

// computedHash is the HMAC of the received plaintext
byte[] computedHash = hMac.doFinal();
// receivedHash is the HMAC that was sent by the communication partner
byte[] receivedHash = ...
```

```
boolean validHash = MessageDigest.isEqual(computedHash,receivedHash);
```

Further Reading Suggestions

- **APIs**
  - Java SE 6: **Cipher**, **SecureRandom**, and **Signature** classes
  - Bouncy Castle: **PEMReader** and **Base64**
- **Books**
  - **Handbook of Applied Cryptography** (free version)
