

Designing a Secure Client-Server System Master of Science Thesis in the Programme Software Engineering & Technology

FREDRIK ANDERSSON

Department of Computer Science and Engineering CHALMERS UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Göteborg, Sweden, September 2009

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Designing a Secure Client-Server System

Fredrik Andersson

© Fredrik Andersson, September 2009.

Examiner: Sven-Arne Andreasson

Department of Computer Science and Engineering Chalmers University of Technology SE-412 96 Göteborg Sweden Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering Göteborg, Sweden September 2009

Abstract

This report describes and discusses the design of a client-server system from a security point of view. The main topics are authentication and data security which can be divided into secure transfer and secure storage.

Authentication is the act where the server and the user prove their knowledge of a shared secret to each other. The shared secret can be of three different types, something you know, something you have or something you are. When talking about a software system a combination of something you know and something you have, for example a password and a hardware token, is the best choice for a system where high security is important. To be able to authenticate without revealing the secret to the other party or any external party an algorithm that uses techniques from public key cryptography and have a similar design as a Diffie-Hellman Key Exchange is used.

Data security is dependent on mainly two properties, confidentiality and integrity, and if both of them can be guaranteed the data is considered secure. Confidentiality is provided by symmetric key encryption and integrity is provided by either a message digest or a MAC.

Contents

Α	bstract		3	
C	ontents		4	
1	Intr	oduction	5	
	1.1	Methodology	5	
	1.2	Delimitations	6	
	1.3	Overview	6	
2	Background		7	
	2.1	Cryptography	7	
	2.2	Authentication	10	
	2.3	Data security	15	
3	Des	ign description	17	
	3.1	Network module	17	
	3.2	Storage module	24	
	3.3	Cryptographic framework for the client	27	
	3.4	Cryptographic components	29	
4	Con	clusion	32	
5	Bibl	iography	33	
Α	Appendix A: Design document for network and storage of the IAM system			

1 Introduction

This report will cover the design of the security critical parts of a system called Intellectual Asset Manager (IAM) developed by CIP Professional Services (CIP PS). The IAM system is a client-server solution that will be used to manage intellectual properties which means that the security of the system is very important since some data stored in it may be corporate secrets.

CIP PS is a consultant firm that is helping their clients to create and facilitate knowledge-based business and innovation. They are also helping them to manage their intellectual assets like patents, relational networks and such.

The IAM system is developed both as an internal tool to be used by the employees at CIP PS to make their work easier as it is going to be able to better keep track of different resources than what they are doing now. It will also be a compliment to the services CIP PS offer to their clients to give them a system that they can use to manage their intellectual properties in an easier way.

The general architecture of the system is a service oriented architecture where the server provides services to the client which users can use if they have the right permission. The parts of the architecture that is interesting for this report is the connection between server and client and the storage part on the server since it is at these two points in the system where data is most vulnerable to exposure. The main subject of this report is to describe a design for these two parts that makes it very hard for an intruder to exploit these vulnerabilities to gain access to secret information.

The server is implemented as a standalone server using Java as implementation language while the client is a Flash based web application which means it is implemented using the ActionScript language. The connection between server and client is handled by a TCP/IP connection instead of using a HTTP connection as usual in a web application.

1.1 Methodology

A study of existing works in the different areas of security that will be covered by the design will be performed. The different areas that need to be covered in this study are authentication, secure transfer, secure storage and some general cryptography to be able to understand the other areas.

To be able to guarantee security, standards is going to be used for as many of the security critical parts as possible since standards generally are well used and tested in a lot of different circumstances. This means that there is less chance of them being broken compared to algorithms and systems that have not been challenged as much about their security as a well used standard would have. Another advantage of using standards is that they can often be used for free which may not be true in other cases.

The standards are chosen to fit as good as possible with the IAM system, following that a design should be developed that embeds these standards into modules that can be connected with the other parts of the IAM system. Important things to think about here is that the design must utilize standards in a way that keeps their supposed level of security and different standards must be used to provide a totally secure solution.

1.2 Delimitations

The most important thing here is that it is assumed that the system is running in a clean environment to be secure. The security of the system is designed specifically to protect where external security cannot help, as during data transfer over an insecure network, authentication and if an adversary gets hold on some data in storage. It cannot guarantee security if the client or server is infected by malware.

Another delimitation is that three-part authentication with a trusted third part is not considered for this system since there is no reason for a user to distrust a server that can guarantee knowledge of a shared secret. A trusted third party is usually used when no such guarantees can be made, like when setting up the secure connection before the user proves its identity.

A limitation that influences the design is the languages that are chosen to implement the system. The design is using as much of the implementation languages own API's as possible to make the system easier and less time consuming to implement. Java has most of the things that are needed to implement a secure system but ActionScript does not have anything at all. That means that time must be spent to develop these parts for the client.

1.3 Overview

This report has two main sections, the first gives a background (section 2) of subjects that is used later in the report, and these subjects are cryptography (section 2.1), authentication (section 2.2) and data security (section 2.3). The second describes the design (section 3) of the different parts of the system that involves any security (section 3.1 and 3.2), and also a cryptographic framework for the client (section 3.3) and motivations for the chosen cryptographic components (section 3.4). The last section of the report (section 4) is used for conclusions.

2 Background

The security of the IAM system can be separated into two different areas and in this chapter the background for these subjects are described. The subjects are authentication and data security, but there is also a part with more general information about cryptography since that is the foundation for both of these subjects.

2.1 Cryptography

This section provides some basic knowledge about the different classes of cryptographic algorithms used by the system to provide secure data transfer and storage. Since the focus of this report is more the security properties of different algorithms and not so much the actual algorithms this section does not contain anything advanced about cryptography. Instead the descriptions are about when and for what different classes of algorithms are used.

2.1.1 Symmetric key cryptography

Symmetric key cryptography is based upon all parts knowing the shared secret which the keys for encryption and decryption is derived from. Often the two keys are actually equal to each other. There are two types of methods for symmetric key cryptography, block ciphers which takes a specified number of bits as input to the algorithm and produces an output of the same length and stream ciphers that encrypts one bit at a time [1]. In the IAM system only block ciphers are used so that will be the focus of this section.

When there is more data that is going to be encrypted with a block cipher than fits within a block the data has to be divided into blocks that are encrypted one by one. If the blocks are encrypted without any relation to each other it is called that Electronic Code Book mode is used. This is however not a very good way to use a block cipher if a lot of similar data is encrypted since an input block always produces the same output block which means that if a lot of input blocks are the same it could be possible to determine the plaintext from the ciphertext without knowing the key [1]. To solve this problem a more advanced mode of operation should be used where the output depends on more than the output of the block cipher, for example each block could depend on something from the previous block or each block could be exclusive-or'ed with a counter to create diversity between input blocks that are equal to each other. Some common modes of operation also supports encryption of data that cannot fill a full block but when using one that does not support this, padding must be used to fill up the last block. An important feature of a padding scheme is that the fill up data must be recognizable and easy to remove again after decryption [2].

Compared with public key cryptography that is described below symmetric key cryptography is very efficient so it is clearly the winner if a lot of data is going to be encrypted. The downside is that a shared secret needs to be distributed to both parts by some other means before it can be used to encrypt any data [1].

2.1.2 Public key cryptography

In difference to symmetric key cryptography there is no need for a shared secret between the parts when they are using public key cryptography. Instead each party creates a pair of keys, one public for encryption and a private for decryption. The public key can then be distributed to those who want to send encrypted data to the owner of the corresponding private key. Even though

the problem of sharing secret keys are eliminated by using public key cryptography there is still a problem with confirming that a public key belongs to the entity that it is supposed to belong to [1]. To do this some kind of authentication of keys is needed; an example of this is the use of a trusted third party that can confirm the origin of a public key [3].

Public key cryptography is usually based on hard mathematical problems which make it infeasible to compute the plaintext from the ciphertext or the private key from the public key if sufficiently large numbers are used. Examples of such problems are the problem of factoring large integers and the problem of calculating discrete logarithms. Both of these have algorithms for solving them but if used with numbers that are large enough the time it will take to solve them with the computational power available today will exceed the age of the universe [1].

Public key cryptography is mostly used to encrypt small amounts of data since it have a rather large computational cost compared to symmetric key cryptography. For larger amounts of data it is often better to use the latter since it is so much faster but public key cryptography is then used to solve the problem with distribution or agreement on a shared secret. It can also be used to digitally sign data with the private key so that everyone with the public key can confirm its origin.

2.1.3 Cryptographic hash functions and Message Authentication Codes

A cryptographic hash function is used to calculate a fixed length message digest from a message of arbitrary length. To be called a cryptographic hash function it must also satisfy certain properties [4]:

- The function shall be a one-way function which means that it is computational infeasible to compute a message from the message digest.
- The function shall be collision resistant which means that it is computational infeasible to find two messages that results in the same message digest.

Another property that a cryptographic hash function should satisfy is that it should be very fast [1].

A Message Authentication Code (MAC) is similar to a message digest since it is also calculated by a one-way function and that it is a very low probability of finding two messages that produces the same MAC. The difference is that except for the message, functions that calculates a MAC also takes a key as input which results in that different MAC's can be calculated from the same message depending on the key [5].

A MAC is used to check that a message has not changed since the MAC was calculated (its integrity) and to check the origin of the message (its authenticity). This can be done since both the message and the key must be correct to calculate a correct MAC but it is not possible to determine which of these checks that failed [5].

Even though a message digest also can be used to check the integrity of a message [4] it is more used to bring messages down to a workable size to increase the speed of other algorithms and it can also be used to prove the knowledge of some secret data without actually revealing the secret since it is a one-way function [1].

2.1.4 Random number generation

Random numbers are important in cryptography to reach a sufficient level of security when generating keys but it is generally hard to achieve true random sequences because of lack of unpredictable sources. There is an important difference between pseudo random number generators and cryptographically secure random number generators. While both provide random numbers with good statistical properties a high measure of unpredictability is also needed for a cryptographically secure random number generator [6].

The usual way to generate random numbers is to acquire a seed, which should be a random sequence, which is then used as a parameter to a function that can generate a new random number each time it is called. For a cryptographically secure random number generator there are requirements on both the seed and the function generating the random numbers [6].

The seed both need to be long enough to be able to have a sufficient number of possible values, for example a 128-bit seed has 2^128 values which is needed for an AES-key. A shorter seed would make it easier for an adversary to guess the key since when a seed has been chosen random numbers will be generated from that seed in a predetermined order, which means that if a 64-bit seed is used to create 128-bit key there is still only 2^64 values that the key can have. Another thing that is important about the seed is that it needs to have near to the same probability of all values to not make it considerably easier for an adversary since it has a higher chance of success in guessing the value by trying the values with higher probability first [6].

The level of randomness (or entropy) in a random sequence can be measured in *bits of information* and *min-entropy*:

bits of information
$$= \sum -p_i * \log_2(p_i)$$
 (1)

$$min-entropy = -\log(max(p_i))$$
 (2)

Where i go from 1 to the total number of values and p is the probability of choosing a specific value. For a good seed both of these measurements should be the same as the number of bits in the seed, if they are not the seed needs to consists of more bits than the level of security requires.

2.1.4.1 Source of entropy

To create a good seed it is required to have a good source of entropy. Examples of such sources are anything that creates random noise that can be digitalized and read by the computer that is going to use it, which means that random noise from hardware is the easiest to use [7]. The best source of entropy would produce bits where each bit has the same probability to be 1 or 0, but sources where the probability is different can also be used but more data needs to be produced for the same number of bits of information.

Since accessing this kind of noise is more on the level of the operating system than application level it is good that both Windows and different UNIX systems have ways of gathering random data that can be accessed by applications. For UNIX systems it can be reached in the special file dev/random [6] and for Windows through the function called CryptGenRandom which is included in the Windows Crypto API [8]. These are usually the best sources that can be read from locally,

but if that is not possible other sources must be used. Such sources could be system properties that are somewhat random or user generated entropy [6].

System properties that can be used as a source for entropy can be anything that has some randomness to it; an example is the system clock since it is random when the call to get the current time happens. But it is important to only use the part that can be considered random for this kind of values, which in this would be the two least significant bytes if the system time is measured in milliseconds. This is because it may be possible to predict values higher than that for anyone that knows the implementation of the system [6].

Sources that can be used as user generated entropy are for example timing of key strokes and mouse movements. The problem with this type of entropy is that it takes time to gather which causes problems if it is going to be used right after the application has been started. Also key strokes is often buffered which means that they lose some of their entropy, though it can be solved by not using every stroke the gathering time will be even longer [6].

2.1.4.2 Pseudo Random Number Generators

The function that is generating random numbers from the seed also has a property that needs to be satisfied for it to be called cryptographically secure. That is that if a generated number is known it shall be infeasible to calculate the following sequence without knowing the internal state of the generator. If a generator has this property the level of security depends on the seed and that the algorithm for calculation of the pseudo-random numbers is not reducing the security level of the seed [6]. This happens for example if a seed with 256 bits of information is used with a hash-function with an output of 160 bits.

2.2 Authentication

When talking about authentication in a client-server environment it refers to the process of confirming the identity of a user of the client and often also confirming that the client is talking to the correct server [9]. Since the client and server probably are going to communicate after the authentication process it is also used to agree upon one or more ephemeral keys that can be used to encrypt data that is sent between them (see 2.3). This chapter will describe different types of methods to authenticate user and some terminology that is used throughout this report.

2.2.1 Types of authentication methods

There are mainly three types of secrets that can be used as shared secrets between two or more parts for authentication. These are something you know, something you have and something you are [3]. Something you know is usually some kind of memorable password but can also be an answer to a question that only the user knows the answer to [10]. Something you have can be anything that the user is able to carry with him but is usually some hardware token that contains some secret. The token can either be passive and only stores the secret [11] or be active and use the secret to produce one-time passwords [10]. Something you are can be a fingerprint or data from a retinal scan [10]. Lately a fourth type of secret called *somebody you know* has been discussed. The idée behind this is to make use of a user's social network for authentication [10].

Something you know is clearly the type of secret that is used the most since passwords are used on most web sites with any kind of login service [12]. Often a weak authentication scheme like sending the password as plaintext is used. This is insecure but often accepted when used on

channels that are hard to eavesdrop [3]. Another type of scheme is the challenge-response scheme where the secret is never sent between the parts. Instead one part sends a random challenge that is used to calculate a response using the secret so that it is not possible to learn the secret by only eavesdropping. However the use of this scheme requires that there are enough possible values for the secret so that it is infeasible to do a brute force search for the secret [3].

Below positive and negative properties about the three different types of secrets are presented. Requirements to make each type secure are also presented along with specific threats against that kind of secret.

2.2.1.1 Authentication using memorable passwords

The use of memorable passwords (also called weak passwords) is the most common way to authenticate a user of a computer system [12]. The reason for this is that memorable passwords are easy to use for both users and those who make these systems since no extra equipment is needed to store or access the secret that is used to identify the user [12]. Unfortunately memorable passwords are also very short and often common words or names which make them vulnerable to password guessing attacks [12].

A password guessing attack is exactly what the name says, an adversary trying to guess the password of a user to compromise its identity and get access to the system through that user's account. There are two different kinds of password guessing attacks, online attack where the adversary is trying to make legal authentication attempts with guessed passwords and offline attack where the adversary is performing an exhaustive search against some publicly available data, for example data that has been acquired through eavesdropping on a legitimate authentication session performed by a user [13]. It is the last of these two that is the biggest threat against an authentication scheme using memorable passwords since the search space for these weak passwords are small compared to cryptographically secure keys [13]. To protect against offline password guessing attacks it must be infeasible to derive any information about the password from the data that is passed between the participants of the authentication procedure [14]. This can be done by masking the password by combining it with another temporary secret value in a way that makes it hard to confirm the password without knowing something more than the adversary cannot know [12]. Online guessing is not as big a threat since it is possible to limit the number of guesses that an adversary can make by not allowing more than a given number of guesses before taking measures to prevent any more attempts. As long as the system can detect the user trying to authenticate it should be no problem to prevent anyone to guess the password this way [15].

Another weakness with memorable passwords is that they are vulnerable to malware such as key loggers since they must be entered to the system using a keyboard [9] and they are also vulnerable to someone looking when the password is entered. This is not as sophisticated as malware but work as well and with some equipment it may not even be necessary for the watcher to be close [16]. These are typical attacks which the authentication scheme itself cannot provide protection against.

A limitation with memorable passwords is the fact that they have to be memorized, both that they have to be small as discussed above and because they can be forgotten, and since the

number of passwords for a lot of people is more than a few [17] users may start writing them down which will make it harder to keep them secret. Also if a password is forgotten there will be a need of a way to recover the password or get a new one which needs to be as secure as the authentication scheme to not be a liability since an authentication system isn't more secure than its weakest component [10].

2.2.1.2 Authentication using hardware tokens

Hardware token is a common name for a small hardware device that is used to store a user's secret, often used in conjunction with a memorable password (see 2.2.3.4) [18]. Since a hardware token does not have the problem of forgetting things as the human memory does it can be used to store cryptographically secure secrets that are infeasible to guess because of the large set it is chosen from. That it can't be guessed makes it possible to use faster computation techniques when working with this kind of secret compared to the relatively slow public key techniques used when using memorable passwords to authenticate a user [11]. This is an advantage since they are going to be used on small hardware tokens with limited computing power. Examples of these computation techniques are cryptographic hash functions and bit operations instead of arithmetic operations that are used by public key techniques [11].

A common type of authentication using hardware token is that the token is used to produce one-time passwords which either changes after each login, after a set time interval or is depending on some challenge from the server. The one-time password is computed from a secret that both server and client know and possibly other parameters like a memorable password and public random numbers [18]. A one-time password makes it more difficult to gain unauthorized access because a password that changes all the time cannot be guessed like static passwords. It also provides protection against replay attacks since the shared secret is never transferred over the network [19]. A small weakness that some of this type of token has is that the one-time password is displayed on the token for the user to type into the application which makes it possible for keyloggers to record the passwords [20] [21]. It is not as big threat as for memorable passwords but a one-time password can still be useful for an adversary. It can in some circumstances use the password to authenticate itself as the user ones if it manages to intercept the authentication attempt from the user [19]. This means that the same steps as for a memorable password need to be taken to hide it from an eavesdropper to prevent all possible access.

Another type uses a permanent strong password to identify the bearer of the device it is stored on [11]. This is generally not as secure as using one-time passwords, especially if the token has no computational power since it will reveal the secret every time it is used by the device that is reading from the token. This means that it can be vulnerable to attacks from malware or fake reading devices [16].

Even though the secret that is stored in the hardware token is considered a stronger secret than a memorable password the hardware token itself is not considered as strong because a lost hardware token can cause a high amount of damage if the wrong person gets hold of it while a forgotten password only causes problems to the actual user. This is the main reason that hardware tokens are usually used together with a memorable password [10].

2.2.1.3 Authentication using biometrics

A third method of authentication is using biometrics which is a collection of methods using secrets derived from *something you are*, that is an individual's genetic traits or behavioral characteristics [22]. Some examples of biometrics used for authentication are palm/fingerprint, iris, voice and palm vein pattern.

Since biometrics is of the type *something you are* it can in difference to other authentication methods also be used for identification. The differences are that for identification a biometric sample is compared against all posts in a database to find a match that tells the identity of the owner of the sample while for authentication a user is claiming an identity and the sample is only matched against that identity to confirm or deny that the user is who it is claiming to be [23]. However the focus here will be on the use of biometrics for authentication.

Biometrics present some advantages against other authentication methods, the most prominent one is that in a system that uses biometrics for authentication a user should be able to authenticate itself without any doubt since the biometric data connected to that user is unique to the individual user and can only be entered into the system if the user is physically at the authentication device [24]. Still this characteristic of biometrics also presents some difficulties, especially when using it for remote authentication.

The risk of spoofing attacks is even greater when using biometrics for remote authentication than it is for other authentication methods since if an adversary can succeed in compromising some user's biometric data that the user uses for authentication with a remote service it will be hard to revoke the identity theft. This is because that an individual only has one set of biometrics [24]. Another problem is that some of the capture devices may also be susceptible to spoofing, for example by using gelatin fingerprint, high-quality image of an iris or voice recording [22]. Although it may not be an easy task to perform these spoofs, since the adversary in all cases need something from the user and it can be hard to not make the user suspicious when taking it this is a security risk. Because of these risks biometrics is usually used in combination with some other type of secret to make it more secure (see 2.2.3.4) [24].

Other issues are that storing biometric samples on a server raises a need for encrypting the samples so they cannot be used if they are stolen. The reason for the need of encryption instead of the use of one-way functions as is used for other authentication methods is that a capture of biometrics is never exactly the same which gives a problem with one-way functions since they are supposed to make it impossible to relate two closely related values. The need for encryption raises the need for key management which often poses some difficulties [24].

To solve these problems client side matching can be used instead, but it requires that the client can be trusted which need extra cryptographical support to accomplish. There are also ways to use a biometric sample to generate keys that can then be used to authenticate with the server. Challenges here are to be able to generate the same key every time and to preserve that the secret still is *something you are* instead of *something you have* [24].

The advantages with biometrics are mainly that it makes the life of the user much easier, it cannot be forgotten or lost as passwords and hardware tokens and it is something you always carry with you [22]. Also it is not easily stolen or copied since the user uses unique features of its

body and work is done to prevent the kind of spoofing attacks with copied "body parts" mentioned above [24].

Some disadvantages is that some users may think techniques used with biometrics are intruding on their privacy and that biometrics is a somewhat costly solution both because of that the actual capture devices are expensive and that most techniques are protected by patents [23].

2.2.1.4 Multi-factor authentication

Some secrets are not strong enough to protect an identity by itself so it needs additional secret(s) to be secure. It is important that this other secret is of another type than the first one since secrets of the same type has the same weaknesses. To use multiple types of secrets for authentication is called multi-factor authentication (or sometimes two-factor authentication when two secrets are used).

Multi-factor authentication is often used when secrets of the types *something you have* or *something you are* is the primary secret as mentioned in the chapters about hardware tokens and biometrics. The reason for this is not their cryptographic properties but the fact that they are physical things that can be stolen or copied in some cases. Because of this they are often protected by another secret, usually a password, so that they can't be used by the wrong person. Secrets of the type *something you know* is not exactly a subject of this problem since a forgotten password cannot be used by anyone else, although passwords can also be stolen by other means than breaking the authentication scheme [10].

When using biometrics or hardware tokens it is a good thing to use multi-factor authentication because of the extra security provided by also using a short password comes at an acceptably cheap cost even though it takes away the advantage that users does not have to keep anything in their memory [24]. Passwords are harder to complement in this way since that method is often chosen because it is cheap and simple which means that the extra security provided by an extra factor may not be an option because it will most likely change these properties [23].

2.2.2 Terminology

This part contains information about different terminology that concerns authentication that is used throughout this report when authentication is discussed.

2.2.2.1 Mutual authentication

When both the client is authenticated to the server and the server have proved it is the server that the client wanted to connect to, mutual authentication has been achieved [25]. This is used to prevent server spoofing attacks which is especially important when doing remote connections in completely unreliable networks like the Internet but can also be necessary in more secure environment to protect from possible inside attacks [13]. To achieve mutual authentication there are a few different ways depending on different situations, one is to use a trusted third party that can confirm the identity of the server to the client [12]. Another way is that the client already has a message digest (see 2.1.3) of the server's public key to identify the server as the rightful owner of the matching private key [13]. A third way is that both client and server have to prove that they know a shared secret which will identify them to each other [14].

2.2.2.2 Zero knowledge proofs

A zero knowledge proof is a way to show knowledge of a secret without revealing anything about the secret itself. It is often used in authentication schemes since it does not give any information to a potential adversary while proving knowledge of the secret to the other part and since the other part also knows the secret it can confirm the correctness of the proof by doing the same computations [26].

2.2.2.3 Perfect forward secrecy

This is a property that authentication schemes has if it is impossible for an adversary to obtain old session keys if it has managed to learn a user's secret and it is not possible to learn the secret by doing a brute-force attack on a session key [12].

2.2.2.4 Server spoofing attack

A server spoofing attack is when an adversary is pretending to be the server that the user is trying to connect to. This makes it possible for the adversary to obtain user data that the user thinks it is sending to the server; instead the adversary can use them to perform a legitimate login attempt to the server. The best way to protect against this kind of attack is to use mutual authentication since then it is not possible to trick the client into believing it is talking to the server when it is actually talking to the adversary [11].

2.2.2.5 Replay attacks

Here the adversary's approach is to record messages from communication between server and client and resends messages or parts of messages at a later time [13]. The goal with this is to be able to impersonate the user or server or to obtain some secret data that can be used for other attacks [12].

2.3 Data security

There are two properties that are important to satisfy for data to be secure. Those are:

- Confidentiality, which means that data is resistant against anyone being able to find out the content of it.
- Integrity, which means that data is resistant against tampering.

The confidentiality property is usually fulfilled by using symmetric key encryption (see 2.1.1) to encrypt the data because it is a lot faster than public key encryption that would be the other choice for this. Integrity is reached by using either a MAC (see 2.1.3) or a message digest (see 2.1.3) to identify that the data is the same after decryption as it was before encryption [27].

The difference between using MAC or message digest is that a message digest needs to be encrypted since it will be the same for identical data. This is also true for a MAC computed with the same key as the encryption (and initialization vector if that is used) but that does not help an adversary after a connection is terminated if a new key is used for every new session.

2.3.1 Data transfer

When sending secure data between two or more participants it is also important that no data get lost. To avoid data loss the sender needs to be able to resend messages that are found with errors and the receiver needs to tell the sender that the message has been received. The reason for the last one is that since the whole message is encrypted when it is sent, any change to the

encrypted message will make the decryption of the message unreadable which means that the receiver cannot ask for a specific message to be resent. The solution to this is that the sender keeps sending a message at a given interval until told by the receiver to stop. Usually it only has to send the message once [28]. For this to be possible every message should have a message id which can either be a number that is increased by one for each message or be some random number that has a very small chance of being repeated during the same session. The message id is also good for preventing replay attacks if the receiver is keeping track of message id's of received messages.

It is very hard to break a secure connection that uses confidentiality and integrity as described above anywhere between the participants, when it has been established with a session key that is only known by them. But the data transferred between them may still be vulnerable if any of the participants is infected by malware and since an intruder usually uses the easiest way this would probably be the approach to get access to data instead of trying to break the cipher that messages are encrypted with [29].

2.3.2 Secure storage

The most secure way to remotely store data from a user point of view is to let the data be encrypted on the client before it is sent to the server and then never decrypted until the client have received it again after a user request. This is called end-to-end security [27] and is very secure because no trust is placed on the server and security is only dependant on the keys the client used to encrypt the data. However in a system where a user is not the single owner of the data this may be harder to accomplish since keys may be needed to be distributed between multiple users which greatly reduce the security of the data since an adversary now has a lot more entry points. Taking it one step further to when the data is owned by the system and different users have access to different data. That makes end-to-end security impossible to achieve. Instead the data needs to be secured in two phases; during transfer between client and server and during storage on the server. Anytime data passes between these phases it is insecure since it needs to be re-encrypted with another key. In a system like this it is important that external threats that does not attack the system directly, like malware, is considered.

An important thing to avoid loss of data during storage is redundancy, data should be stored in more than one place to avoid loss if one set of data is corrupted or destroyed. This can either be done using multiple servers, which can also be used to increase security by only storing a part of the data on each server [30], or two sets of the data can be stored on a single server.

Another important aspect of secure storage is how the keys are managed [27]. In difference to the ephemeral keys that are used for secure transfer the keys that are used for secure storage can be used up to two years and they need to stay secret all the time [28]. And if those keys are needed to decrypt data after their active period they need to stay secret even longer.

3 Design description

This section presents the design of the network and storage modules. The modules are designed to fit well with the IAM system but also to be enough independent that they can be used in other systems. They are both designed mainly from a security point of view since they are going to be used in a system where confidentiality is very important but they are also designed to be able to run in a multi-threaded environment in a safe way. This is important to be able to handle a lot of users at the same time.

Except for the network and storage modules a framework for cryptography is presented. This is needed for the client since the ActionScript library does not contain any cryptographic API which is needed for the network module. Also descriptions about the different cryptographic components needed for both network and storage modules can be found in section 3.4.

3.1 Network module

One of the main thoughts for the design of the network module is that the server and client should be as similar as possible. The only differences are because of their different roles and the differences of the languages they are implemented with. The reason for making the designs of them alike is both to make them easier to implement and to make it easier to use both parts of the module.

For both client and server the module consists of three main classes that have the same roles in both versions (see figure 1 and figure 2). Those are ConnectionManager, Session and Authenticator. ConnectionManager is used as an interface to the network module and also handles incoming and established connections for the server. Session handles a single connection and Authenticator is used during the authentication process.

The Session class is a link between the Socket, connected to the other part in the connection, and the ConnectionManager or the Authenticator during the authentication phase. It defines how data is sent over the Socket which at the lowest level is a stream of bytes representing a message preceded by an integer that tells how many bytes the message consists of. The reason for using byte streams instead of sending objects is because Java and ActionScript are not compatible for that. Another way it could be done is to send strings encoded in UTF (Unicode Transformation Format) which is supported by both Java and ActionScript and also takes care of the length of each message which makes it very easy to use. The reason why it is not used here is that the classes used for cryptography in Java operates on byte arrays and when a byte array has been encrypted it is not possible to convert it back to a string and send it like that instead because there is not a representation in UTF for all bytes (or sequence of bytes). This gives the result that when converting from byte array to string and then back to byte array again, the two byte arrays would not be equal which would ruin almost every message sent.

A problem with using byte streams is that there is no protection against data corruption, e.g. if someone adds a byte in the middle of a message it will corrupt that message and all messages coming after that. This is solved by clearing the input buffer when such an error is detected, and detection is made either by that the integer preceding the message is to large or negative or that it takes to long time to receive a full message, e.g. if there is less bytes in the buffer than what is

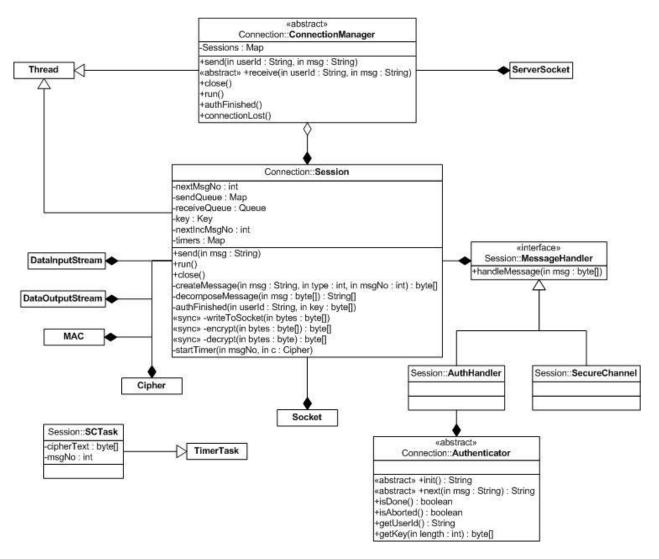


Figure 1: Class diagram over the Java network module

told by the integer. Corrupted data that slips by these checks will be ignored on a higher level once the integrity check has been made.

Message handling on a higher level is done by a message handler that is represented as an inner class in Java and a method in ActionScript. The Session class has two different message handlers, one for handling messages when in the authentication phase and one that is used when the secure channel is established. The first of these is quite simple and is only forwarding messages to the Authenticator and sends the answers to the other part. When the authentication is done it also sets up the secure connection (see 2.3).

The confidentiality property of the secure connection is provided by the AES block cipher (see 3.4.1) and the integrity property is provided by the SHA-256 cryptographic hash function (see 3.4.2). The reason for using a message digest instead of using a MAC is because it provides better structure to the design when creating messages to have the bytes that provide the integrity check to be encrypted with the rest of the message. Since a MAC does not need encryption it would be unnecessary overhead to use that while a message digest have to be encrypted

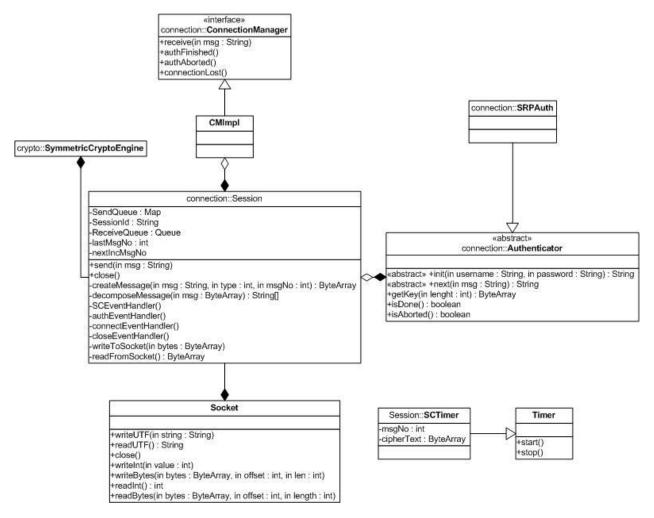


Figure 2: Class diagram over the ActionScript client network module

anyway. Another reason is that the hash function is needed for authentication on the client while MAC would not be needed anywhere else so by choosing to use message digests means that less amount of code needs to be written.

Most of the differences between the server and the client that is not because of their difference in functionality is because the difference in multithreading between Java and ActionScript. When using sockets in java multithreading is a necessity to be able to receive data at any time without blocking the rest of the application but ActionScript does not have any multithreading at all and is relying on events instead. Each time an event is triggered an event handler is executed, in this design a message handler is an event handler for the event that is triggered when there is some data in the input buffer of the socket. Since ActionScript is single threaded there is no need for any thread safety but in Java it is necessary to consider all code that may be executed by two threads at once. If such code has effects that may lead to a bad state or behavior it must be made mutual exclusive so that only one thread can execute it at once. An example of this is when sending data since it is bad if two messages get entangled with each other.

A weakness that the Session class has is that it is vulnerable to an adversary that continues to send random bytes to a certain socket since it can effectively block Session from being able to receive any legitimate data. This can be done since each time any error is found it will throw the whole input buffer away and there seem to be no good solution for this problem. However this

will only affect the Session that is the target for this attack and will not be able to put down the whole server. Also this does not accomplish very much for the adversary because it will not give it any access to secret data.

The network module is not designed solely for use in the IAM system but it has a rather specific role, which is that of a secure channel with authentication though it should not be very complicated to extend it to perform other roles as well, e.g. an unprotected connection. Also when used for its original purpose there are some options that can be made about what cipher and authenticator that is used to make it possible to adapt the module if necessary.

3.1.1 Authentication

The authentication is as mentioned above performed by an implementation of the abstract Authenticator class but is still using the Session to send messages. The design intends to make it possible to use any authentication scheme that can be used to authenticate two parts to each other over a network connection. Although the design has been made to be able to use any authentication scheme this section will present a recommendation for this system.

3.1.1.1 Type of authentication method

In section 2.2.1 three different types of authentication methods where described and also that combinations of these methods can be used to improve security. This section will discuss which of these methods that is best suited for the IAM system.

All three types of authentication methods have their strengths and weaknesses which make them suitable for different kinds of secure systems and this doesn't have that much to do with the level of security that can be achieved with any of them but is instead more about practical matters. This is because for all types there are authentication schemes that provide such security that it is infeasible for an adversary to perform a successful attack against the scheme that steals a user's identity or impersonates the server. But there are still other ways for an adversary to compromise a user account without breaking the protocol since one of the biggest potential security flaws for most authentication schemes is the ability of the users to keep their secrets secret.

Hardware tokens can be lost or stolen while biometrics can be copied in some cases and passwords can be forgotten by the user. The difference here is that if an adversary manages to steal a hardware token or copy a biometric feature that is used for authentication it is a potential security risk while a forgotten password is only an inconvenience for the user. This is one of the reasons that both hardware tokens and biometrics are seldom used by themselves but combined with a short password that will keep the adversary busy if the other secret is compromised. It will also give some time to set the compromised secret as invalid if the theft is discovered.

So as long as a password is only kept in the user's memory and not stored in any physical way by the user it is the best type of secret since it cannot be stolen like the other two. The security can also be enhanced by using it in a multi-factor scheme together with one or both of the other two but that extra security comes at the cost of extra hardware that may be expensive for both hardware tokens and biometrics.

The greatest risk for a password to be exposed is when the user is entering it into the system since it is the only time it should be out in the open if handled correctly. At this moment it is

vulnerable both to someone observing the user typing it and to key loggers that can log the password and send it to the adversary. There are similar risks with hardware tokens and biometrics but to a much smaller degree since it involves switching out hardware in the biometrics case and for some cases with hardware tokens, or only gaining access to one-time passwords that can only be used for a short time that is used by many hardware tokens.

The recommended choice of authentication method for the IAM system is multi-factor authentication that uses both password and hardware token. The reason for this is that even though an authentication scheme that only uses a password is secure on its own and that it is a much cheaper solution than any of the alternatives it can never account for human errors on its own. That gives the choice between either a combination between password and hardware token or password and biometrics where the hardware token is the clear winner in this type of system. Biometrics are unnecessary expensive for use in this system since a reader would be needed for every single computer that may be used as a client. It is also more difficult to implement since the client application then needs to read data from a hardware device while a hardware token can be used to calculate a one-time password that then is typed in by the user. Except for protecting against human errors using hardware tokens also protects against malware since a one-time password is often not much use after being logged by a key logger.

3.1.1.2 Choice of authentication scheme

Although that the recommendation is to use an authentication method with more than one factor a request for the system was to use only a password for authenticating users. This means that an authentication scheme that can provide secure authentication with passwords as the only secret is needed but since it is recommended to use multi-factor authentication a scheme where it is possible to also include another factor is the best choice. One such authentication scheme is APKAS-AMP that is described in the IEEE std 1363.2-2008 standard [31] which is a standard for password-based public key cryptography and includes about ten schemes for authentication and key agreement. Except for this standard there are only a few on this subject. One of them is RFC2945 which describes the Secure Remote Password (SRP) protocol which is also specified in IEEE 1363.2 with only small modifications. Another standard is PKCS #5: Password-Based Cryptography Standard from RSA Laboratories but it was never really considered because it was discovered to late in the process and the decision to use APKAS-AMP was already made.

Like all of the schemes from IEEE 1363.2 APKAS-AMP provides mutual zero knowledge proof which means that both server and client prove that they know the password without revealing it to the other party during the authentication process. And also that both client and server gets authenticated by the other party to avoid server spoofing. Other security properties that apply to all schemes of IEEE 1363.2 are [31]:

- Successful use of scheme to agree on a shared key is limited to parties that know the password or the password-verification data that corresponds to it.
- Knowledge of the password-entangled public keys and any other public keys exchanged between participants does not allow a third party to verify guesses for the password, or to determine the shared keys.
- A participant that does not know the password gets the opportunity to verify at most one (or at most a very small number) of guesses for it for each interaction with the other.

The second property takes care of the greatest threat to password based authentication, offline guessing attacks, since if the adversary cannot verify guesses in an exhaustive search it is not possible to find out the correct password this way. Online password guessing is taken care of by the third property, if it is only possible to guess one password at a time it is easy for the server or user to detect an attack and can then lock the account to prevent further guesses.

Another property that is important for the protection of the user's passwords is that the authentication scheme belongs to the asymmetric trust model which means that only a password verifier, not the password, is stored on the server. A password verifier is a value that is derived from the password but it is computationally infeasible to retrieve the password from the password verifier. A problem with this property is that it is not possible to combine the password with another authentication factor which means that one-time passwords cannot be used unless the authentication scheme has another input parameter. This is what makes APKAS-AMP the ideal choice for the IAM system since it is the only scheme in IEEE 1363.2 that has such a parameter.

One negative thing about APKAS-AMP is that it has stricter requirements on its domain parameters than most other schemes which means that more computation is needed to generate domain parameters and some extra checks is needed in the authentication procedure. On the other hand is APKAS-AMP one of the least complex schemes of 1363.2 which should make up for this.

3.1.1.3 Description of APKAS-AMP

APKAS-AMP is like all authentication schemes in IEEE 1363.2 based on public key cryptography (see 2.1.2) and users identify themselves to the server using a short memorable password. The basic idée is to agree upon a shared secret without exposing the password. To accomplish this both parties generate a private key and then they use it to create a public key that they send to the other party. At least one party must also include the password when creating the public key; this is called a password-entangled public key. The shared secret can then be calculated from the public key of the other part (or both public keys) and the own private key. These calculations may be very different between different schemes though.

In APKAS-AMP the client first calculates its public key (w_c) from its private key (s_c) and then sends the public key to the server (3). The server then calculates its public key (w_s) from the password verifier $(v=g^u)$, the client's public key and its own private key (s_s) before sending the public key to the client (4). When this is done the server can calculate the shared secret from the client's public key and its own private key (5) while the client uses the server's public key and its own private key to calculate the shared secret (6).

$$w_c = g^{s_c} \tag{3}$$

$$w_{S} = \left(w_{C}^{i} * v\right)^{S_{S}} \tag{4}$$

$$z = (w_c * g)^{s_s} \tag{5}$$

$$z = w_s^{((s_c+1)/(s_c*i+u))}$$
 (6)

The reason for that it is hard for an adversary to be able to calculate the session key or the passwords from intercepted public keys is that all calculations are made in a finite field. To be able to do this it is necessary to calculate a discrete logarithm which is a hard problem for large numbers [1]. For security reasons it is important that a sufficiently large finite field is used, in this case the field is defined by a prime q that has a length of 1024 bits. This means that all calculations made within this finite field are done modulo q. It is also important that the generator q is of order q where q is a prime that has a length of 160 bits. That means that q mod q = 1 must be satisfied or else this scheme will not work properly [31].

Since only the server does a commitment of the password in this scheme it is very important that the server does not use any keys derived from the shared secret before it has confirmed that the client knows the same secret. If the server uses the keys before it has received a commitment of the password from the client it may be possible for an adversary that masquerades as the client to perform an offline-guessing attack to find the password. This type of scheme is called a unilateral commitment scheme while a scheme where both parts commit themselves with the password is called a bilateral commitment scheme [31].

It is not necessary for the client to confirm the key of the server since it did not do a commitment it is not possible for the server to do any guesses of the password. Still it is a good choice to let the client perform key confirmation of the server anyway since it is the only good way to confirm that the server did know the password verifier. Otherwise this may not be discovered before an encrypted message is sent from the server to the client since the client would not be able to decrypt it in a correct way if the server used a false password verifier to create its key. This would be impractical since there is no way to know if the "corrupted" message really was corrupted somewhere between the server or the client or if the server is using the wrong key which means that the connection needs to be kept open since it impossible to know what caused the problem.

One important property that APKAS-AMP has is perfect forward secrecy (see 2.2.2.3). Since the shared secret does not include the password (5) it is not possible to find out the password from a leaked session key and it is also not possible to retrieve old session keys by using the password. Due to the design of the IAM system only the first of these properties are of any use here because if an adversary would get access to a user's password it will be able see all information and history available to that user anyway and will not need an old session key to access the information contained in old encrypted messages. This design however increases the importance of the impossibility to retrieve the password from a session key since a lost password sets the secrecy of all information at risk.

APKAS-AMP can be implemented in two different settings; either the discrete logarithm setting or the elliptic curve setting. The implementation for the IAM system will use the discrete logarithm setting because of better support in Java since the only thing that is needed is the possibility to use big integers, in this case up to the value of 2^{1024} .

3.1.1.4 Storage of user information

When storing user information, there are some measures that can be taken to improve the security if the user file/database is stolen. As mentioned above an authentication scheme that uses a password verifier on the server instead of a clear password is preferable since it will make it a lot harder for an adversary that has access to the user file. The reason for this is that

password verifiers of possible passwords must be computed to be able to compare them to the verifiers in the file. To improve the security even more it can be good to use a slow function to compute the password verifiers to make it take longer for an adversary performing a brute-force attack on the user file [31]. Another thing is to use individual salt, a random number that may be public that is combined with the password using some one-way function, for each user since that will make it impossible to search for more than one password at a time. If individual salt is not used an adversary can do a search for a large range of passwords and then test them on all users hoping that it will match at least one. This is not possible when using individual salt [31].

3.1.1.5 Password recovery

It is inevitable that users sometimes will forget their password which means that some way to recover a lost password must be available. It is very important that a password recovery service is at least as secure as the authentication procedure because if it is not, it will weaken the whole system since it would be easier for an adversary to recover a password through the password recovery procedure and then authenticate in the normal way [10].

In this system there is no automatic password recovery that the user can use by itself because of the possible weakness. Instead the user has to contact an administrator and identify itself either by recognition or some id to get a new password. Because of how the IAM system stores passwords it is not possible to retrieve the old one.

3.2 Storage module

The design of the storage module is dependent on how the stored data is used in the IAM system so it is necessary to understand how this is handled in the system to understand the decisions for the design of the storage module.

All data is stored in a single file that uses XML for structure and when a user connects to the server the server sends all data that it has stored, or if the user has restricted access only the parts that it is allowed to access will be sent, back to the client. Then as long as the client is connected it has the full state of the data locally and any modifications it makes will be sent to the server for storage but it will also be broadcasted to all other clients that are currently connected. In this way all clients will always have the same state of the data as the server.

The most important property of the storage module except its ability to store data in a secure way is that it is able to run correctly in a multithreaded environment since each user connected to the IAM system is running its own thread. This affects any operation that modifies the data since if the data is being modified by two or more threads at the same time it probably becomes corrupted or in a state that does not include the changes from all threads. It is also necessary to avoid reading of data as a write operation is performed by another thread because it can cause incomplete data to be presented to the user that made the read operation.

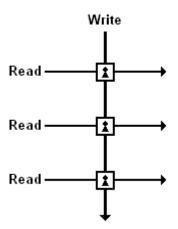


Figure 3: A simple model about how the lock used in storage module works. Read operations can run simultaneously but only one write operation can run at the same time.

The solution to this is to use a lock that allows parallel read operations but all write operations are exclusive (see figure 3). Unfortunately this will not do much for performance for this system because of the way data is stored and the fact clients only need to read from the storage once for each session but it still seems beneficial to use this kind of read-write lock instead of a mutual exclusive lock since it makes it possible to use this design in another setting where read operations are more common.

Considering that the whole data file must be read every time it is going to be modified it would be a lot of reading and writing to the file, and following that a lot of encryption and decryption if only a few users are doing changes to the system. If a lot of users are using the system this could be a serious performance issue. The solution is to not modify the file for every modification a user does, but instead save the changes in memory until it is time to write them to the file. The file is then updated by a set interval which should be long enough so that no performance issues may arise.

The design of the storage module (see figure 4) provides security by using symmetric key cryptography for confidentiality protection and message authentication code for integrity protection. It also uses a set of two data files with exactly the same content to prevent loss of a lot of data if an error would occur when writing to the file. It is also able to recover from such an error as long as at least one of the files is not corrupt.

The security of the storage module depends on the cipher and the mode of operation used for encryption (see 3.4.1), the MAC algorithm used for integrity checks (see 3.4.3) and the ability to keep the keys secret. Since the first two should use algorithms that are infeasible to break in a reasonable amount of time with the computational capacity available the possible weak point is how keys are managed.

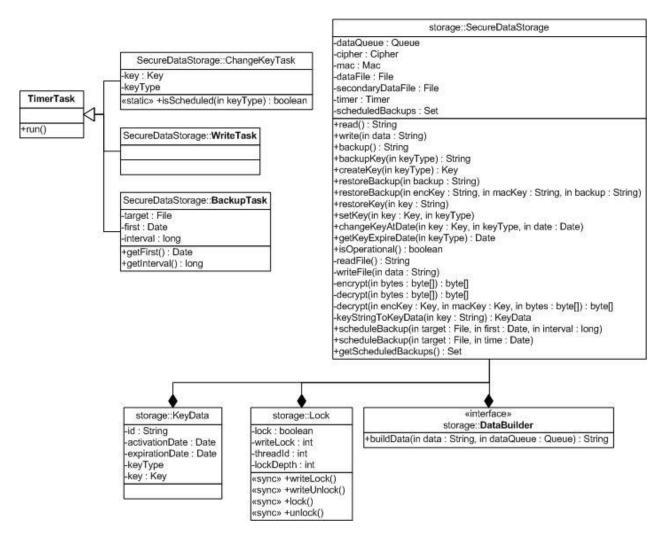


Figure 4: Class diagram showing the design of the storage module.

3.2.1 Key management

The storage module consistently has two different keys that are active if it is in operational state; one key is used for encryption and decryption of data while the other is used for calculating MAC's. If there is no active key for either of these the storage module will not be operational and calls to some methods may result in a bad state. There can also be two keys, one for each active key, that are scheduled to become active at a specified time to cover for smooth transitions between keys when they need to be changed.

All four of these keys are stored in the memory of the server that is running the application which will keep them physically secure until they are being deactivated. This is considered secure because there is no easy way to recover the keys from the memory [28]. Keys that have been deactivated should be overwritten so they cannot be compromised later and used for recovering data from old backups. Since the active period of a key used for data encryption or integrity protection of the same data should be at most 2 years key material that is stored in memory may leave traces even after it has been overwritten and it could be a good thing to move the key after being in the same memory location for some time [28].

The reason for changing keys are that the more a key is used the more probable it is that it can be compromised because it is easier to obtain a key from encrypted data if you have a lot of data

than if you only have a small amount. This does not mean however that old keys should be completely destroyed when they are deactivated, they should be kept safe until they are not needed to decrypt old data anymore [32]. In this case it means that keys should not be destroyed until backups that are encrypted with them are no longer needed.

To save keys for storage outside the storage module there is a method that can be used to backup keys as a string that can be saved to a file. An important thing about these backups is that they are not automatically encrypted so if they are going to be stored on a disk that is not physically secure they must be encrypted first and the key used for that must then be stored somewhere where it is physically secure. That a key is physically secure means that it is stored in such a way that it is hard to get access to it without permission, for example it is stored on an encrypted disk that is encrypted by a physically secure key or it is stored on a disk that is kept in a safe that is hard to break into [28].

Except for the sake of keeping keys to use for restoring old backups it is very important to have backups of the active keys since if one of those keys gets lost because of a system crash or something similar there must be a backup to restore the key or all data that was encrypted with that key will be lost. So for recovery purposes it is wise to have a backup of the key stored somewhere else than in the actual system [28]. It is also possible to let the system recover from this kind of failure by itself but since this is by far the easiest way of doing it and since the system is not expected to fail a lot it reasonable to assume that this recovery method will work very well.

Good key management should essentially protect against keys becoming compromised but if that happen it is important to minimize the damage. All data that is encrypted by a key that is suspected or confirmed compromised must immediately be re-encrypted with a new key since it is no longer secure. It is also important to assess the damage this may have caused to determine if any actions need to be taken [28].

In the case of the IAM system it would be severe if the encryption key were compromised since it would let the adversary use it to read secret corporate information. It cannot be used to modify information though since then he would reveal himself since the integrity check will fail. On the other hand the key that is used for calculating MAC's does not pose as large threat since unless he breaks the MAC algorithm it is not possible to get any information from it. Though it is possible for the adversary to replace the stored data and the system wouldn't notice since it is possible to create a valid MAC with the compromised key. If both keys were compromised the adversary can do anything possible with the data file without the system noticing.

It would be possible to get greater security that only lets the adversary see small bits of information when he manage to compromise a key by spreading the information over multiple keys. But this would probably also require multiple servers and that was not in the scope of this design but could be considered if the storage module of the system is going to be extended.

3.3 Cryptographic framework for the client

Since there is no cryptographic API in ActionScript the cryptographic algorithms needed for authentication and data transfer must be included in the security design. One alternative here could be to just implement the needed algorithms as a single class each and make them more or less only usable in the context of the IAM system. This alternative would however not be

extendible and the reusability of the algorithms may be very limited. Also this way of doing things may also make it harder to change cryptographic components if they need to be changed because of the algorithm being broken or some other reason. A better alternative is to create a framework that can be used as a base for implementing all of the most common types of cryptographic components and also makes it easy to change between components of the same type.

The cryptographic framework for the client is designed to be both reusable and extendible but also to fit well with the design of the network module for the client (see figure 5). The main part of the framework consists of the interface CryptoEngine and its subclasses. It includes most of the classes that is used in encryption/decryption operations for the two main directions of cryptography, symmetric key cryptography and public key cryptography, and also classes for calculations of MAC's, hash values and random number generation.

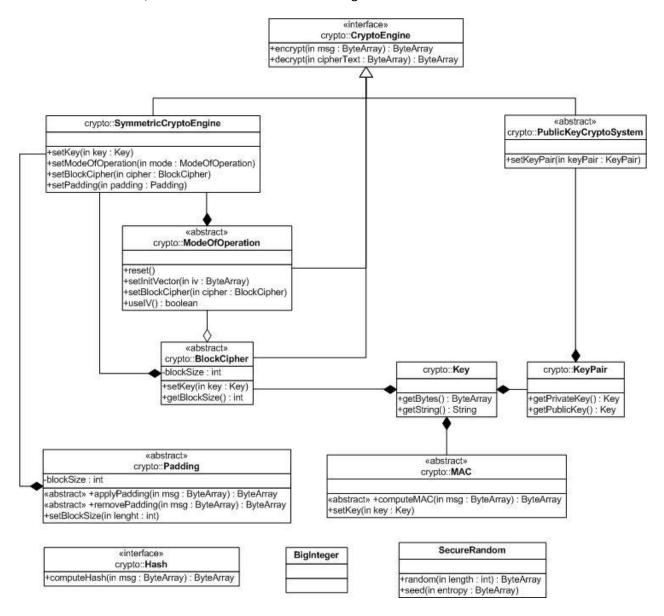


Figure 5: Class diagram showing the design of the cryptographic framework for ActionScript

The design of the symmetric key cryptography part is made very dynamic so that any of the different parts can be switched out easily. This means that the SymmetricKeyCryptoEngine class can be used in the same way regardless of which block cipher, mode of operation and padding scheme that is used. This way it is very easy to extend and reuse different parts, for example if a new block cipher is implemented it is not necessary to implement new modes of operation and padding schemes to use with this block cipher. However there is one intended limitation because of an adaption to work better with the network module; it is that if the mode of operation uses an initialization vector (IV) it will be added to the beginning of the resulting byte array. In the case of the network module for the IAM system this adaption will make it easier to change between different modes of operation since no changes will be needed there based on if the mode uses IV or not. In other applications this may not be optimal, for example if the IV is going to be encrypted for better security, but in most cases this should not be a problem. Compared to the Java Cryptography Architecture (JCA) [33] it is an improvement when creating the kind of solutions where the mode of operation can vary between modes that do or do not use an IV. In Java there is no way to get this information from the Cipher object that is used for encryption/decryption, it is only possible to get the name of the mode of operation that is used but also this is unnecessary complicated.

The framework is made to be easy and intuitive to use in difference to JCA which more seems to be made for possible extension on a larger scale and requires some reading before you are able to use it. JCA uses engine classes that are interfaces to specific type of cryptographic components to generate a component specified by a string by the user [33]. This can be useful since there will only be a small set of classes but on the other hand the user needs to know the strings used for generation of specific components and in some cases also which components that fit together. In the framework presented here the user has direct access to the actual classes that implements the components which should make it more intuitive to use. The framework is also easy to extend but it does not have as good extendibility as JCA that has a design where it is possible to separate between different providers and also let the choice of provider of a component be either visible or invisible to the user. This makes it possible for a user of JCA to have different implementations of the same algorithm usable at once. While that is also possible in this framework by using packages with different names (or similar) it can become messy and is not as seamless as JCA. This framework is much better suited as a small scale cryptographic framework which fit this situation perfectly.

3.4 Cryptographic components

This chapter discusses which cryptographic components that are used and motivates why they were chosen. Also random numbers is discussed in this chapter since it is needed for the authentication scheme.

3.4.1 Cipher and mode of operation

As mentioned before the IAM system uses symmetric key cryptography for encryption of data both for storage and transfer between server and client because it gives much better efficiency than public key cryptography. Thou since performance are not a very big issue the only requirement on the block cipher is that it must provide security for a foreseeable future.

The choice of block cipher for this system is the AES (Advanced Encryption Standard) cipher partly because it is available in the Java API which enables easy implementation on the server. More importantly the Rijndael algorithm, the algorithm behind AES, was selected from a number of strong ciphers because of speed and security strength [1] but in truth any of the five finalists of the competition to become the new standard could have been used for this system.

To avoid the problem with block ciphers that is mentioned in section 2.1.2 the Cipher Block Chaining (CBC) mode of operation will be used. As the name claims this mode chains the blocks together by exclusive-or'ing the plaintext blocks with the previous ciphertext block which will prevent equal plaintext blocks from producing equal ciphertext blocks. The first plaintext block is exclusive-or'ed with a random block called initialization vector (IV) to prevent the same message to produce the same ciphertext. The IV should be different for each new message to prevent security risks [2].

The reason for choosing CBC as the mode of operation for the IAM system is that it is simple so it is easy to implement and also that it is included in the Java API. Still there is a problem with CBC when there is a bit error in the plaintext or an error during the calculation or storage of the previous ciphertext. Because the chaining such an error will propagate through the rest of the message. Another drawback with CBC against some other more advanced modes of operation is that it requires a whole block of data before any data can be encrypted [1]. In this design however, those problems are not of big consequences since bit errors are more a concern on a hardware level than when the algorithm is implemented in software. The issue with the necessity of using full blocks is not a problem both since the whole messages will be available at once and to use smaller block sizes would decrease the efficiency since more encryption/decryption operations would be needed to be executed by the cipher.

Since CBC must have full blocks as input padding is necessary to fill up the last block. For this the PKCS5 padding, the padding used in the standard "PKCS5: Password-based encryption", is used [34]. The reason for using PKCS5 padding is again that it is possible to use it with AES and CBC in Java, other than that it is not important how the padding scheme works because the only thing that is required by the padding for AES with CBC mode is that it is easy to remove again after the message has been decrypted.

3.4.2 Cryptographic hash function

A cryptographic hash function is used for the integrity check of the secure connection (see 2.3) and is also used by the authentication scheme and the random number generator used for the client. Since Java already has a few different hash functions in its cryptographic API one of them will be used to avoid unnecessary implementation work. That one is SHA-256 and the reason for using it is mostly because it is not broken and the others (SHA-384 and SHA-512) that are not have higher security than necessary for what it will be used for. Actually those three that is considered broken, MD2 and MD5 [35], or partly broken, SHA-1 [36], would not introduce any flaws because it is their collision resistant property that is broken while the authentication scheme depends on that a hash function is a one-way function and the secure channel only needs it to be highly unlikely to produce the same message digest from two different messages. The reason to not use any of them is that it is not good practice to use anything that is considered broken even if it does not have any bad effects in this application, and by using a

hash-function that has a higher level of security the application can possibly be considered secure for a longer time.

3.4.3 Message Authentication Code

The storage module uses a MAC for the integrity check of the stored data. The MAC algorithm that is used is the HMAC algorithm. This is an algorithm that uses a cryptographic hash function to provide its one-way property and in this case that hash function is SHA-256. The reason for using this algorithm is that it is the only one that is provided by Java and also that it is very effective because of the use of hash functions. An example of another algorithm is CBC-MAC which uses a block cipher in a modified CBC mode to create a MAC.

3.4.4 Random numbers

While Java already have a cryptographically secure random number generator [37] that uses entropy source from the operating system [38] ActionScript lacks this which means that it must be implemented for the client. This causes some problems especially with entropy generation since ActionScript has limitations that make it hard to access an entropy source that is managed by the operating system and it has access to relatively few system properties that is not enough to give a suitable level of security.

One solution is to let the user generate the needed entropy but since it will be needed when authenticating to the server there is no time for generating entropy while working which would be the ideal way to do it. This means that if an input device is going to be used the user must be prompted to use the device on purpose of generating entropy and one way to do that is if the user moves the mouse back and forth horizontally. Then entropy can be gathered by taking the position of the mouse pointer at a certain interval or at the turning points. To get a better result the screen can also be divided into segments of 256 pixels since then it will not matter if the user only moves the mouse over a small part of the screen. The downside of that is that it will take more time to gather the needed entropy but since this is very important for the security of the application it is a necessary sacrifice.

A problem with this solution is that it is a quite tedious procedure for the user even if it only takes a few seconds to perform which would make a solution without user interaction better. But because of the importance of security for the IAM system it is not an option to sacrifice security against usability so a solution must at least be as good as this one. The best solution would have been if it were possible to obtain entropy from the local operating system even if it is necessary to use another language than ActionScript and save it to a file which then could be opened by the client application. But since it has to be able to run in a browser the choices are very few and none of them can do this.

The pseudo random number generator algorithm that is going to be used is the Hash_DRBG algorithm which is one of a few recommendations in the publication NIST SP 800-90 [7]. The reason for using this is that these recommendations are free to use and that they are recommended by NIST should mean that they have been thoroughly tested and analyzed since they are used by the U.S. Government. Of the algorithms in this publication Hash_DRBG is the one that is best suited for the client since it is both simple to implement and the hash function component that is needed is already going to be implemented for other uses by the authentication scheme.

4 Conclusion

This report describes and discusses a secure client-server solution that provides secure authentication and secure transfer through a network module and secure storage through a storage module. The network module is designed to work with almost any system that needs secure authentication and transfer while the storage module is more limited to systems that use a similar storage model as the IAM system. For guaranteed security all cryptographic algorithms that are used within these modules are standardized which gives a higher chance of them being thoroughly tested and analyzed.

There are two issues with the design presented in this report. The first is that the network module is vulnerable against a denial of service attack that can effectively block a connection between a client and the server. But it cannot be used to acquire any secret data from the system which makes it somewhat tolerable. The second is that there is no really good way to obtain secure random numbers when using ActionScript. The proposed method of using mouse movements is negative to the usability of the client application since the user will be prompted to move the mouse to generate a random seed every time the application is loaded. This is something that may make a user frustrated, especially if the user doesn't understand the reason for this extra work.

The Problem with ActionScript is mainly because it is not at the moment made to handle this kind of secure applications that uses TCP/IP sockets for communication. Maybe it would be a better solution to use SHTTP for secure communication instead, but then it would be necessary to use TLS (Transport Layer Security) to provide secure transfer which also includes the use of a trusted third party which was to be avoided in this system. Also the server would have to be completely redesigned to run on an application server instead of being a standalone server application.

5 Bibliography

- [1] Trappe, W., & Washington, L. C. (2002). *Introduction to cryptography with coding theory*. New Jersey: Prentice Hall.
- [2] National Institute of Standards and Technology. (2001). *NIST Special Publication 800-38A*. Gaithersburg: National Institute of Standards and Technology.
- [3] Gentry, C., Mackenzie, P., & Ramzan, Z. (2005). Password Authenticated Key Exchange Using Hidden Smooth Subgroups. *The 12th ACM conference on Computer and communications security* (pp. 299-309). Alexandria, Virginia, USA: ACM.
- [4] National Institute of Standards and Technology. (2002). *Federal Information Processing Standards Publication 180-2*. Gaithersburg: National Institute of Standards and Technology.
- [5] Bellare, M., Canetti, R., & Krawczyk, H. (1996, Spring). Message Authentication using Hash Functions The HMAC Construction. *CryptoBytes*, *2* (1), pp. 12-15.
- [6] Schiller, J., & Crocker, S. (2005). *RFC4086: Randomness Requirements for Security.* The Internet Society.
- [7] Natianal Institute of Standards and Technology. (2007). *NIST Special Publication 800-90*. Gaithersburg: Natianal Institute of Standards and Technology.
- [8] Microsoft. (2009). *CryptGenRandom Function (Windows)*. Retrieved June 24, 2009, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/aa379942(VS.85).aspx
- [9] Sabzevar, A. P., & Stavrou, A. (2008). Universal Multi-Factor Authentication Using Graphical Passwords. *IEEE International Conference on Signal Image Technology and Internet Based Systems* (pp. 625-632). Bali, Indonesia: IEEE.
- [10] Brainard, J., Juels, A., Rivest, R. L., Szydlo, M., & Yung, M. (2006). Fourth-Factor Authentication: Somebody You Know. *The 13th ACM conference on Computer and communications security* (pp. 168-178). Alexandria, Virginia, USA: ACM.
- [11] Chang, Y.-F., & Chang, C.-C. (2004). A Secure and Efficient Strong-Password Authentication Protocol. *SIGOPS Operating Systems Review*, 79-90.
- [13] Halevi, S., & Krawczyk, H. (1999). Public-Key Cryptography and Password Protocols. *Transactions on Information and System Security*, 230-268.
- [14] Bresson, E., Chevassut, O., & Pointcheval, D. (2003). Security Proofs for an Efficient Password-Based Key Exchange. *The 10th ACM conference on Computer and communications security* (pp. 241-250). Washington, DC, USA: ACM.
- [15] Lin, C.-L., Sun, H.-M., & Hwang, T. (2000). Three-party Encrypted Key Exchange: Attacks and A Solution. *SIGOPS Operating Systems Review*, 12-20.

- [16] Sasamoto, H., Christin, N., & Hayashi, E. (2008). Undercover: Authentication Usable in Front of Prying Eyes. *The twenty-sixth annual SIGCHI conference on Human factors in computing systems* (pp. 183-192). Florence, Italy: ACM.
- [17] Brasee, K., Makki, S. K., & Zeadally, S. (2008). A Novel Distributed Authentication Framework for Single Sign-On Services. *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing* (pp. 52-58). Taichung, Taiwan: IEEE.
- [18] Vipin, M., Sarad, A., & Sankar, K. (2008). A Multi Way Tree for Token Based Authentication. *International Conference on Computer Science and Software Engineering* (pp. 1011-1014). Wuhan, China: IEEE.
- [19] Haller, N. (1995). RFC1760: The S/KEY One-Time Password System. The Internet Society.
- [20] RSA Security. (2009). RSA SecurID 800 Authenticator Product Description.
- [21] VeriSign. (2009). *VeriSign Identity Protection Center About VeriSign Identity Protection*. Retrieved June 24, 2009, from VeriSign: https://idprotect.verisign.com/learnmoretoken.v
- [22] Boatwright, M., & Luo, X. (2007). What do we know about biometrics authentication? *4th annual conference on Information security curriculum development*. Kennesaw, Georgia: ACM.
- [23] Bala, D. (2008). Biometrics and Information Security. *The 5th annual conference on Information security curriculum development* (pp. 64-66). Kennesaw, Georgia: ACM.
- [24] Bhargav-Spantzel, A., Squicciarini, A., & Bertino, E. (2006). Privacy Preserving Multi-Factor Authentication with Biometrics. *The second ACM workshop on Digital identity management* (pp. 63-71). Alexandria, Virgina, USA: ACM.
- [25] Abdalla, M., Bresson, E., Chevassut, O., Möller, B., & Pointcheval, D. (2006). Provably Secure Password-Based Authentication in TLS. *ACM Symposium on Information, computer and communications security* (pp. 35-45). Taipei, Taiwan: ACM.
- [26] Goldreich, O., Micali, S., & Wigderson, A. (1991). Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of ACM*, 38 (3), 690-728.
- [27] Kher, V., & Kim, Y. (2005). Securing Distributed Storage: Challenges, Techniques, And Systems. *StorageSS* (pp. 9-25). Fairfax, Virginia, USA: ACM.
- [28] National Institute of Standards and Technology. (2006). *NIST Special Publication 800-57-Part1*. Gaithersburg: National Institute of Standards and Technology.
- [29] Gasmi, Y., Sadeghi, A.-R., Stewin, P., Unger, M., & Asokan, N. (2007). Beyond Secure Channels. *Proceedings of the 2007 ACM workshop on Scalable trusted computing* (pp. 30-40). Alexandria, Virginia, USA: ACM.
- [30] Subbiah, A., & Blough, D. M. (2005). An Approach for Fault Tolerant and Secure Data Storage in Collaborative Work Environments. *StorageSS* (pp. 84-93). Fairfax, Virginia, USA: ACM.

- [31] Institute of Electrical and Electronics Engineers. (2008). 1363.2-2008: Standard Specification for Password-Based Public-Key Cryptographic Techniques. New York: Institute of Electrical and Electronics Engineers.
- [32] Fumy, W., & Landrock, P. (June 1993). Principles of Key Management. *IEEE Journal on Selected Areas in Communications*, 785-793.
- [33] Sun Microsystems. (2006). *Java Cryptography Architecture*. Retrieved June 9, 2009, from Sun Microsystems:
- http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html
- [34] Kaliski, B. (2000). *RFC2898: PKCS #5: Password-Based Cryptography Specification Version 2.0.* The Internet Society.
- [35] Wang, X., & Yu, H. (2005). How to Break MD5 and Other Hash Functions. *Eurocrypt 2005* (pp. 19-35). Jinan, China: Springer-Verlag.
- [36] Yin, Y. L., Wang, X., & Yu, H. (2005). Finding Collisions in the Full SHA-1. *25th Annual International Cryptology Conference* (pp. 17-36). Santa Barbara, California, USA: Springer Berlin / Heidelberg.
- [37] Sun Microsystems. (2008). *SecureRandom (Java Platform SE 6)*. Retrieved April 2, 2008, from Sun Microsystems: http://java.sun.com/javase/6/docs/api/java/security/SecureRandom.html
- [38] Sun Microsystems. (2003). *JDK 1.4.2 Security-related APIs & Developer Guides*. Retrieved April 2, 2009, from Sun Microsystems: http://java.sun.com/j2se/1.4.2/docs/guide/security/
- [39] NIST. (2008, November 3). *General Information*. Retrieved April 2, 2009, from National Institute of Standards and Technology: http://www.nist.gov/public_affairs/general2.htm

Appendix A

Design document for network and storage of the IAM system

1 Introduction

This document describes the design of a secure connection with authentication for a client-server system and the design of a secure storage module. Both of these modules are design to satisfy the requirements of CIP Professional Services Intellectual Asset Manager System (working name) but they are independent modules that should fit any system with similar requirements.

Except for these two modules a design for a cryptographic library is provided since that is not provided by the default API of ActionScript. The design however is only a shell and only those algorithms that are necessary for implementation of this system is described here but the design is made to support implementation of a full cryptographic library.

This document is divided into eight sections where section 2-5 describes the design of the different modules, section 6-7 describes the format of the messages sent between client and server and data backups and the last section has descriptions and pseudo-code for the cryptographic algorithms that is used in this system.

1.1 General guidelines

This topic describes general implementation guidelines that are used throughout this document.

- Methods beginning with get/set are usually just getters and setters and should if not further described be implemented as such.
- Methods beginning with "is" return the Boolean value of the attribute with the same name.
- Constructors with no implementation details are supposed to only initialize attributes that is represented by its parameters, and nothing else.
- Italic style is used to show that a word is a name of an attribute or parameter.
- Conversion between byte arrays and strings shall be done using the UTF-8 character set except if the byte array consists of encrypted data. Encrypted data should instead be converted into a string of hexadecimal numbers or something similar.

2 Client network module

This is the part of the client that handles the connection to the server and is responsible for authentication and that the data transfer between client and server is secure. This is a small module, consisting of only two primary classes, Session and Authenticator, and an interface called ConnectionManager (see figure 1). Session is the class that handles receiving and sending data to the Socket and is responsible for everything connected to that. Authenticator is an abstract class responsible for the authentication procedure and is used for implementing different authentication schemes. The ConnectionManager interface must be implemented by the class that is supposed to communicate with the network module; it defines methods that a Session object uses to communicate with the rest of the application.

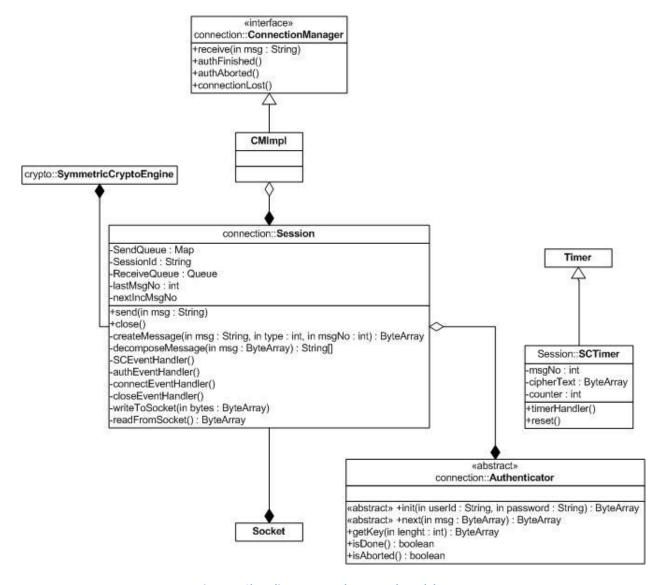


Figure 1: Class diagram over the network module

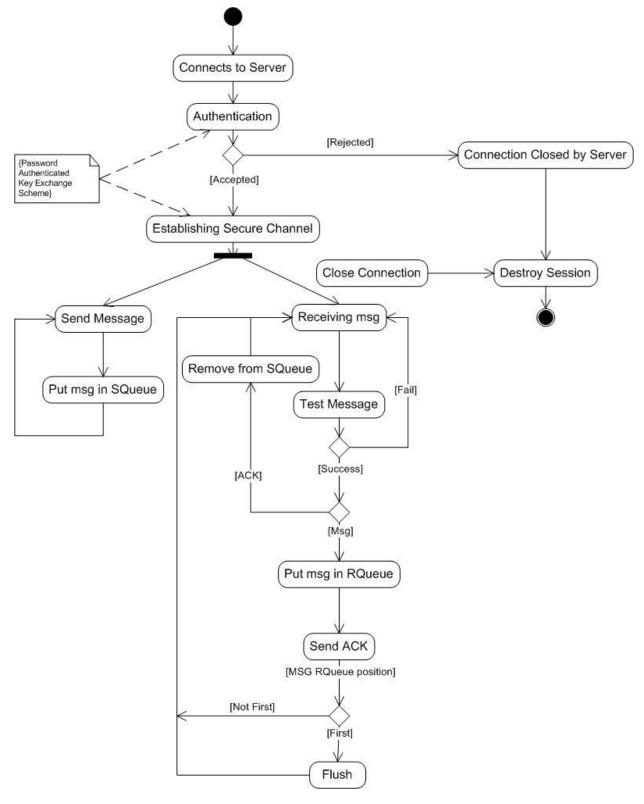


Figure 2: Activity diagram over the client network module.

The different steps that is performed when connecting and maintaining a connection is shown in an activity diagram in figure 2. The connection procedure consists of two activities, authentication and establish secure channel. The authentication activity is actually many steps but they are not described here since they differ from different authentication schemes, more about that in the

description of the Authenticator class (see section 2.2) and in the security chapter of this document (see section 8.1).

The rest of the diagram shows what is happening when either a message is sent to or received from the server. These two flows shows how two queues are used to manage that messages are received in the right order on application level and that messages are saved to be able to be resent if no ACK-message is returned. One thing that is not shown in this diagram is that there is a timer for each sent message that will resend the message if no ACK-message is received for that message before the timer reaches a set limit.

2.1 Session

This is the class that handles the connection to the server. It runs in two different states, first when created until authentication is done it will run in authentication state and pass all received messages to the authenticator that is associated with it. When that is done, if the authentication was accepted it will switch to secure channel state where it can exchange messages with the server without confidentiality and integrity risks.

2.1.1 Constructor

The constructor of session should:

- Set authEventHandler() as the event handler for socketData event of the Socket.
- Set closeEventHandler() as the event handler for close event of the Socket.
- Set connectEventHandler() as the event handler for connect event of the Socket.
- Initialize the Authenticator and if a message was produced send it to the server using the writeToSocket method.

2.1.2 public send:void

This method is used to send messages to the server that the client is connected to using a Socket that is associated with this Session. Using this method ensures that the message is sent over a secure channel that protects its confidentiality and integrity. To accomplish this, the message is encrypted (see section 8.2) to protect its confidentiality and a message digest (see section 8.3) is also sent with the message to ensure its integrity.

2.1.2.1 Parameters

msg – The message that is going to be sent to the server.

2.1.2.2 Preconditions

• The authentication procedure must be finished.

2.1.2.3 Postconditions

- The message has been encrypted and sent to the server.
- A timer that will resend the message after a set amount of time unless stopped has been created and started.

2.1.2.4 Sequence diagram

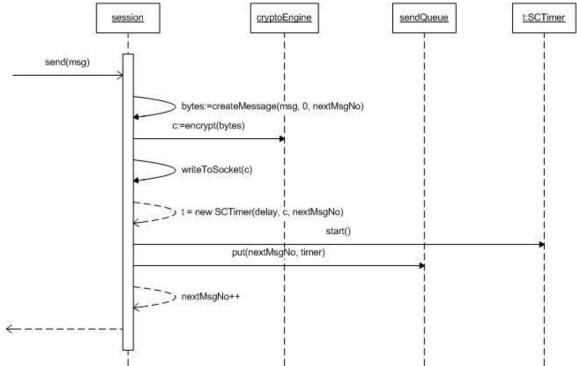


Figure 3: Sequence diagram over send(msg:string):void

2.1.3 public close:void

This method is used to close the connection to the server by closing the Socket that is associated with this Session.

2.1.3.1 Parameters

no parameters

2.1.3.2 Preconditions

no preconditions

2.1.3.3 Postconditions

• The Socket that is associated with this Session has been closed

2.1.3.4 Sequence diagram

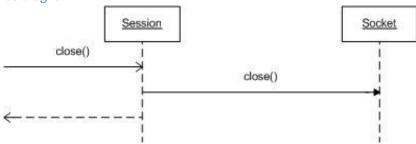


Figure 4: Sequence diagram over close():void.

2.1.4 private createMessage:ByteArray

This method is used to create a message using the format (see section 6) that is used for messages that are going to be transferred over the secure channel. This method is also part of setting up that secure channel since it calculates the message digest (see section 8.3) that is used to protect the integrity of the message.

2.1.4.1 Parameters

msg – The message that is going to be bundled into a "secure message". An application level message.

type – This is the type of the message where 0 = MSG and 1 = ACK.

msgNo – The message number this message is going to have.

2.1.4.2 Preconditions

• type must be of value 0 or 1.

2.1.4.3 Postconditions

A ByteArray has been returned that corresponds to the message described in section 6.

2.1.4.4 Sequence diagram

no sequence diagram

2.1.5 private decomposeMessage:String []

This method is used to decompose messages created with the createMessage method described in 2.1.3.

2.1.5.1 Parameters

bytes – The bytes that the message consists of.

2.1.5.2 Preconditions

no preconditions

2.1.5.3 Postconditions

• A string array consisting of three strings has been returned, where the first is the message number, the second is the message type and the third is the message.

2.1.5.4 Sequence diagram

no sequence diagram

2.1.6 private SCEventHandler:void

This method is used to handle messages that arrive to the Socket associated with this Session after the authentication has been finished. It is supposed to be called when a socketData event occurs which means that there is some data in the Socket buffer.

2.1.6.1 Parameters

no parameters

2.1.6.2 Preconditions

• A socketData event has occurred.

2.1.6.3 Postconditions

- If a null message has been received nothing has happened.
- If the message digest of the message was correct one of the following happens
 - If the message received was of type MSG it was put in the received-queue and if it was
 first in this queue it has been forwarded to the ConnectionManager that is associated
 with this Session. Being first in the queue is the same as having the same message

- number as the attribute nextIncMsgNo. Also if any other message(s) gets first in the queue after this one has been forwarded they has been forwarded too.
- o If the message received was of type ACK the timer with the same message number has been stopped and removed from the send queue.

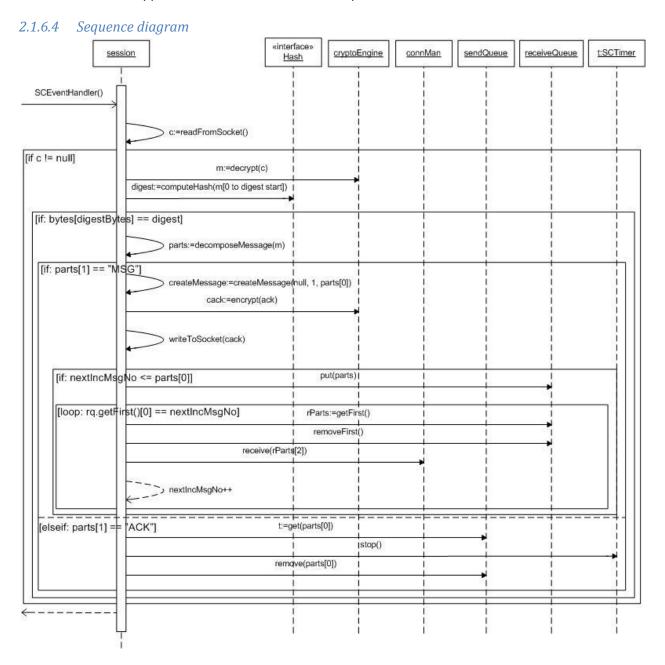


Figure 5: Sequence diagram over SCHandler():void

2.1.7 private AuthEventHandler:void

This method is used to handle messages during the authentication phase of the Session. Like in 2.1.5 it is called when a SocketData event occurs and it usually only forwards messages to the Authenticator that is associated with this Session and then sends the return messages from the Authenticator to the server. It also checks if the authentication is done or aborted and takes appropriate steps if any of these are true.

2.1.7.1 Parameters

no parameters

2.1.7.2 Preconditions

A socketData event has occured

2.1.7.3 Postconditions

- If the message received was a null message nothing has happened
- Otherwise
 - o The message received has been handled by the Authenticator
 - o If a return message was produced it has been sent to the server.
 - If the authentication was aborted this has been forwarded to the ConnectionManager associated to this Session.
 - If the authentication was done this has been forwarded to the ConnectionManager associated to this Session and the shared key for the secure channel has been set and the EventHandler for SocketData events has been changed to SCEventHandler.

2.1.7.4 Sequence diagram

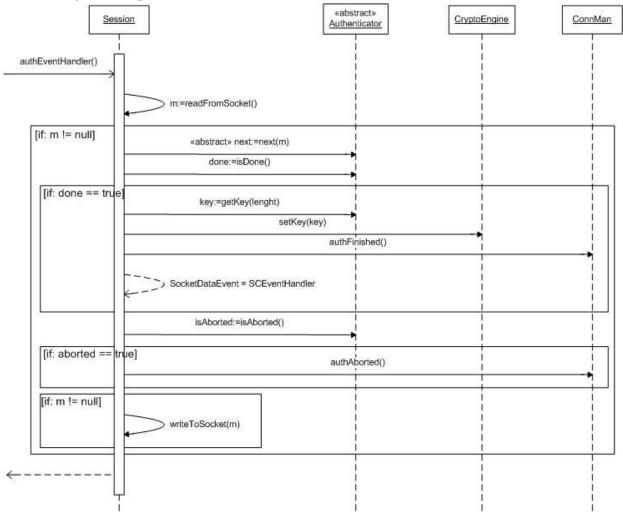


Figure 6: Sequence diagram over authEventHandler():void

2.1.8 private connectEventHandler:void

This method is the event handler that is called when the Socket associated with this Session has finished connecting to the server. It is used to initiate the authentication procedure.

2.1.8.1 Parameters

no parameters

2.1.8.2 Preconditions

• A connect event has occurred.

2.1.8.3 Postconditions

- The authentication procedure has been initiated.
- If a message was produced during initialization it has been sent to the server.

2.1.8.4 Sequence diagram

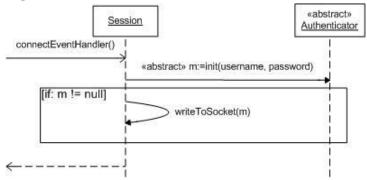


Figure 7: Sequence diagram over connectEventHandler():void

2.1.9 private closeEventHandler:void

This is the event handler that is used when the Socket associated with this Session is being closed from the server.

2.1.9.1 Parameters

no parameters

2.1.9.2 Preconditions

A close event has occurred.

2.1.9.3 Postconditions

• The ConnectionManager associated with this Session has been informed that the connection to the server has been lost.

2.1.9.4 Sequence diagram

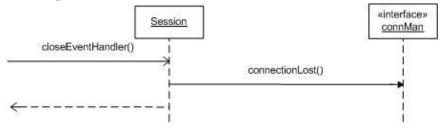


Figure 8: Sequence diagram over closeEventHandler():void

2.1.10 private writeToSocket:void

This method is used to write to the Socket associated with this Session. When writing to the socket the actual bytes that are going to be sent to server is preceded by an int that tells the number of bytes in this message. This is because the server needs to know how many bytes it is supposed to read before a message is at end.

2.1.10.1 Parameters

bytes – The bytes that is going to be written.

2.1.10.2 Preconditions

• The ByteArray bytes is not allowed to be null.

2.1.10.3 Postconditions

• An int with value of the length of *bytes* has been written to the Socket and *bytes* has been written to the Socket after the int.

2.1.10.4 Sequence diagram

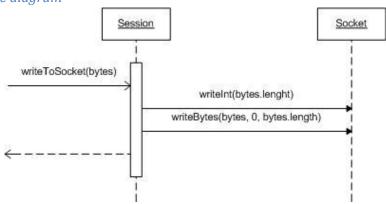


Figure 9: Sequence diagram over writeToSocket(bytes:ByteArray):void

2.1.11 readFromSocket:ByteArray

This method is used to read data from the Socket associated with this Session. Data is read by first reading an int from the Socket and then reading a number of bytes according to the value of this int. It also has measures to prevent two attacks (see figure 10):

- By sending only 1 3 bytes further messages will be corrupted since it will cause the int that is
 read first to consist of these bytes and then the first of the 4 bytes the correct int consists of.
 When this happen the whole buffer must be emptied before messages can be received in a
 correct way again.
- If the int received first is bigger than the length of the following message it will cause the read operation to wait and cause a lock until more data is received. To prevent this, a timer aborts the reading operation after a set amount of time and if such a timeout occurred the read bytes will be disposed off.

2.1.11.1 Parameters

no parameters

2.1.11.2 Preconditions

There must be some data to read from the Socket associated with this Session.

2.1.11.3 Postconditions

- Either a ByteArray including a number of bytes corresponding to a message has been returned. The number of bytes in the ByteArray is equal to the value of an int that was read from the Socket before any bytes was read.
- Or a null value has been returned
 - This happen if the int read from the socket to determine how many bytes to read was bigger than a maximum value or if the reading took longer than a maximum time a reading is allowed to take.

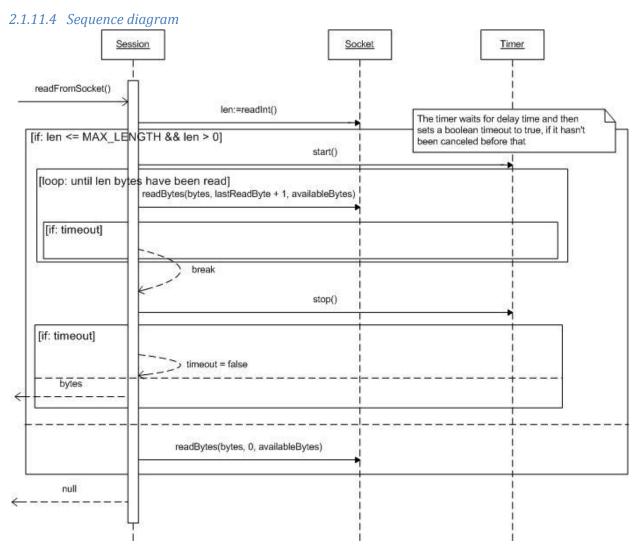


Figure 10: Sequence diagram over readFromSocket():ByteArray

2.2 <abstract> Authenticator

This is an abstract class that provides a base for implementing one side of an authentication scheme (see section 8.1). The design is based upon that the authentication procedure can be in different states, where the state is updated when a message from the other part is received. Also for each state update the Authenticator can (but not necessarily) send a message to the other part. The first state is reached by invoking the *init* method, all other state updates is performed by the *next* method. The *next* method can be used until *isDone* or *isAborted* returns true and when *isDone* returns true, key material that can be used to construct a shared key shall be available.

2.2.1 <abstract> public init:ByteArray

This method is used to initialize the authentication procedure by performing the first step of that procedure.

2.2.1.1 Parameters

username – the username that the user want to identify itself with. *password* – the password associated with the username.

2.2.1.2 Preconditions

- The authentication procedure must not already be initiated.
- The authentication must not already be done.

2.2.1.3 Postconditions

- The first state of the authentication procedure has been reached.
- A message to be sent to the other part or null has been returned.

2.2.1.4 Sequence diagram

no sequence diagram

2.2.2 <abstract> public next:ByteArray

This method is used to advance the authentication procedure to the next state.

2.2.2.1 Parameters

msg – A message from the other part

2.2.2.2 Preconditions

• msg must not be null.

2.2.2.3 Postconditions

- If isDone and isAborted was false the state was updated to the next state and a message to the other part or null has been returned.
- if isDone or isAborted was true, null has been returned.
- if the final state was reached, isDone has been set to true.
- if the authentication has failed, is Aborted has been set to true.

2.2.2.4 Sequence diagram

no sequence diagram

2.2.3 public getKey:ByteArray

This method is used to get key of a certain length.

2.2.3.1 Parameters

length – the length in bytes of the key that should be returned.

2.2.3.2 Preconditions

• isDone is true.

2.2.3.3 Postconditions

• A ByteArray containing *length* bytes has been returned. These bytes have been derived from the key material using the deriveKey function described in section 8.1.1.

2.2.3.4 Sequence diagram

no sequence diagram

2.3 <interface> ConnectionManager

This interface must be implemented by the class that is going to be communicating with the Session object. The methods of this interface are methods that are called by the Session object on different circumstances and should not be used by any other source. The implementations of these methods depends on the application so this is mainly a description of when they are called by Session

2.3.1 public receive:void

This method is called when a message has been received from the server.

2.3.1.1 Parameters

msq – the message that has been received.

2.3.1.2 Preconditions

• A message has been received from the server.

2.3.1.3 Postconditions

postconditions are application dependant.

2.3.1.4 Sequence diagram

no sequence diagram.

2.3.2 public authFinished:void

This method is called when the authentication procedure has been finished.

2.3.2.1 Parameters

no parameters

2.3.2.2 Preconditions

• The authentication procedure has been finished.

2.3.2.3 Postconditions

postconditions are application dependant.

2.3.2.4 Sequence diagram

no sequence diagram.

2.3.3 public authAborted:void

This method is called when the authentication procedure has been aborted

2.3.3.1 Parameters

no parameters

2.3.3.2 Preconditions

• The authentication procedure has been aborted

2.3.3.3 Postconditions

postconditions are application dependant.

2.3.3.4 Sequence diagram

no sequence diagram.

2.3.4 public connectionLost:void

This method is called when the connection is lost

2.3.4.1 Parameters

no parameters

2.3.4.2 Preconditions

• The connection to the server has been lost

2.3.4.3 Postconditions

postconditions are application dependant.

2.3.4.4 Sequence diagram

no sequence diagram.

2.4 SCTimer

This class is a subclass of Timer (a class from the ActionScript library) that is used to resend messages when an ACK messages hasn't been received fast enough. The method timerHandler must be set as the event handler for the timerComplete event.

2.4.1 timerHandler:void

This is an event handler that is used when the timer has reached its specified delay. It resends the message and then restarts itself.

2.4.1.1 Parameters

no parameters

2.4.1.2 Preconditions

• The delay of this timer has been reached.

2.4.1.3 Postconditions

- If *counter* was less than the max number of resends the message associated with this timer has been resent.
- If counter was less than the max number of resends this timer has been restarted.
- If *counter* was equal or greater than the max number of resends the connection has been closed to avoid further errors or tampering.
- counter has been incremented by one.

2.4.1.4 Sequence diagram

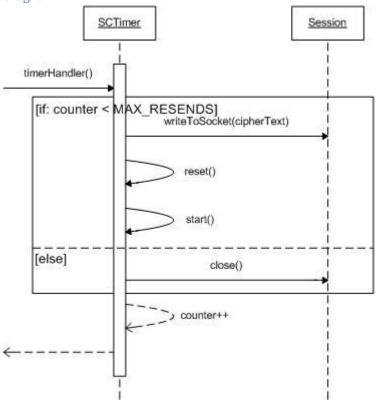


Figure 11: Sequence diagram over timerHandler():void

3 Client cryptographic module

This is mainly a framework for a number of different cryptographic techniques without any specific implementations provided. Example of a cipher and other cryptographical techniques that are needed for the network module is presented in the security section (see section 8) of this document.

The main thought behind this module is that it should be usable not only in this application but in any case where cryptography is needed and easy to extend with more cryptographic schemes. The design is also done in a way that makes it possible to switch out different parts where this can be made possible. An example of this is that when using symmetric key cryptography it is possible to choose which block cipher, mode of operation and padding to use.

The design of this cryptography framework is shown in figure 12 and is mainly focused around the two different encryption models; *symmetric key cryptography* and *public key cryptosystem*, but also support other cryptographic functionality like *hash-functions* and *message authentication code*. It also includes a class for generation of cryptographically secure random bytes and class for representing big integers that is necessary for public key cryptography.

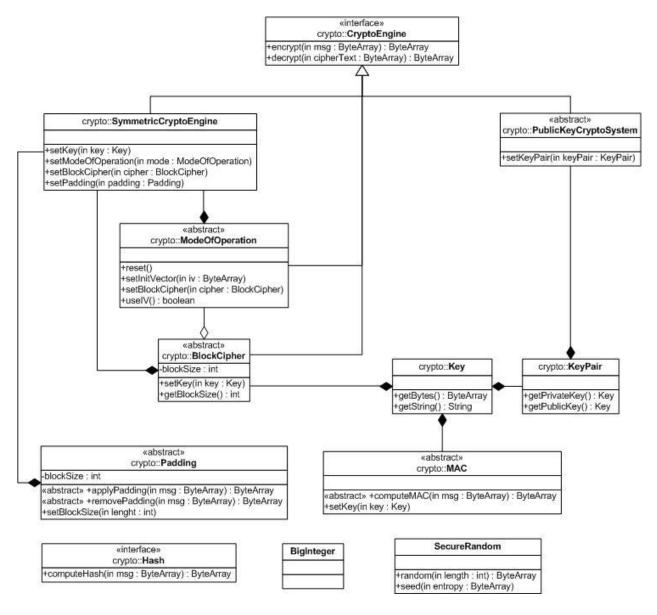


Figure 12: Class diagram showing the design of the client crypto module

3.1 <interface> CryptoEngine

This is a common interface for all classes that perform encryption and decryption.

3.1.1 public encrypt:ByteArray

This method is used to perform an encryption operation.

3.1.1.1 Parameters

msg – The message as a ByteArray that is going to be encrypted.

3.1.1.2 Preconditions

• msq is not null.

3.1.1.3 Postconditions

• A ByteArray containing an encrypted version of *msq* has been returned.

3.1.1.4 Sequence diagram

no sequence diagram

3.1.2 public decrypt:ByteArray

This method is used to perform a decryption operation.

3.1.2.1 Parameters

c – The cipher text that is going to be decrypted

3.1.2.2 Preconditions

• c is not null.

3.1.2.3 Postconditions

• A ByteArray containing a decrypted version of *c* has been returned.

3.1.2.4 Sequence diagram

no sequence diagram

3.2 SymmetricCryptoEngine

This is a subclass of CryptoEngine that is used for symmetric key cryptography though only with the use of block ciphers. Its responsibilities are to hold a suit of a ModeOfOperation object, a BlockCipher object and a Padding object and using those to encrypt or decrypt a ByteArray of variable length. This is done by dividing the input ByteArray in blocks of a size specified by the BlockCipher that is used and feed them one by one to the ModeOfOperation object and then put the result blocks together again. If the last block is shorter than the necessary block size it is padded using the Padding object.

3.2.1 public encrypt:ByteArray

see 3.1.1

3.2.1.1 Parameters

see 3.1.1.1

3.2.1.2 Preconditions

• see 3.1.1.2

3.2.1.3 Postconditions

- If ModeOfOperation, BlockCipher or Padding is not set null has been returned otherwise see 3.1.1.3
- The returned ByteArray the same length as msg + the length of the padding.

3.2.1.4 Sequence diagram

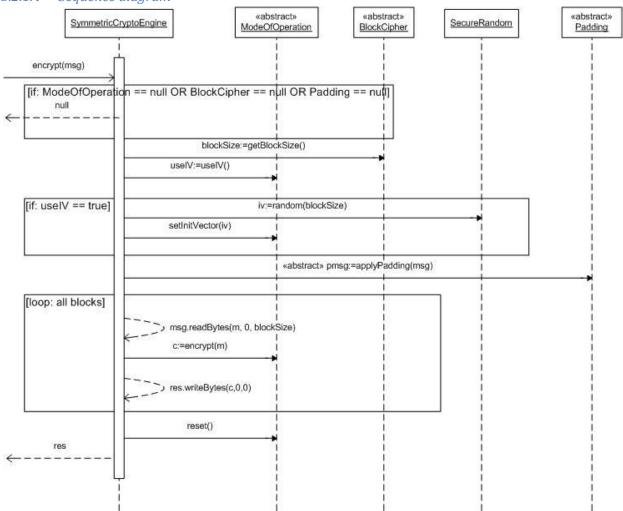


Figure 13: Sequence diagram over encrypt(msg:ByteArray):ByteArray

3.2.2 public decrypt:ByteArray

see 3.1.2

3.2.2.1 Parameters

see 3.1.2.1

3.2.2.2 Preconditions

see 3.1.2.2

3.2.2.3 Postconditions

- If ModeOfOperation, BlockCipher or Padding is not set null has been returned otherwise see
- The returned ByteArray the same length as c the length of the padding.



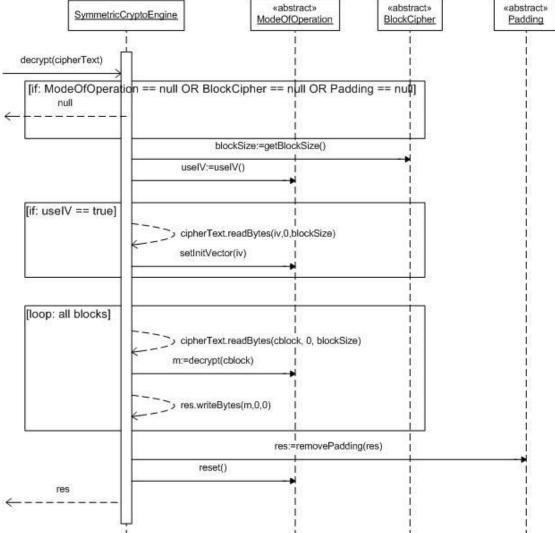


Figure 14: Sequence diagram over decrypt(cipherText:ByteArray):ByteArray

3.2.3 public setKey:void

Sets the key of the BlockCipher that is associated with this SymmetricCryptoEngine

3.2.3.1 Parameters

key – The key that is going to be used by the BlockCipher

3.2.3.2 Preconditions

no preconditions

3.2.3.3 Postconditions

If BlockCipher is set key has been associated with it.

3.2.3.4 Sequence diagram

no sequence diagram

3.2.4 public setModeOfOperation:void

Sets the mode of operation of this SymmetricCryptoEngine.

3.2.4.1 Parameters

mode – The ModeOfOperation that is going to be associated with this SymmetricCryptoEngine.

3.2.4.2 Preconditions

no preconditions

3.2.4.3 Postconditions

- mode has been set as the ModeOfOperation associated with this SymmetricCryptoEngine.
- If there is a BlockCipher associated with this SymmetricCryptoEngine it has also been associated with *mode*.

3.2.4.4 Sequence diagram

no sequence diagram

3.2.5 public setBlockCipher:void

Sets the block cipher of this SymmetricCryptoEngine.

3.2.5.1 Parameters

cipher – The BlockCipher that is going to be associated with this SymmetricCryptoEngine.

3.2.5.2 Preconditions

no preconditions

3.2.5.3 Postconditions

- cipher has been set as the BlockCipher associated with this SymmetricCryptoEngine.
- If there is a ModeOfOperation associated with this SymmetricCryptoEngine *cipher* has also been associated with it.
- if there is a Padding associated with this SymmetricCryptoEngine the blockSize of *cipher* has been set as the blockSize of Padding

3.2.5.4 Sequence diagram

no sequence diagram

3.3 <abstract> ModeOfOperation

This sub class of CryptoEngine is a base for implementing different modes of operation used with block ciphers. More about modes of operation and description for the chosen implementation can be found in sections 8.2 and 8.2.2.

3.3.1 <abstract> public encrypt:ByteArray

see 3.1.1

3.3.1.1 Parameters

see 3.1.1.1

3.3.1.2 Preconditions

- msg must be of the same length as the block size of the BlockCipher associated to this ModeOfOperation.
- A BlockCipher must be associated with this ModeOfOperation

3.3.1.3 Postconditions

- see 3.1.1.3
- The returned ByteArray has the same length as msq.

3.3.1.4 Sequence diagram

no sequence diagram

3.3.2 <abstract> public decrypt:ByteArray

see 3.1.2

3.3.2.1 Parameters

see 3.1.2.1

3.3.2.2 Preconditions

- *c* must be of the same length as the block size of the BlockCipher associated to this ModeOfOperation.
- A BlockCipher must be associated with this ModeOfOperation

3.3.2.3 Postconditions

- see 3.1.2.3
- The returned ByteArray has the same length as c.

3.3.3 <abstract> public reset:void

This method is used to reset the state of this ModeOfOperation and is used with modes that store a state.

3.3.3.1 Parameters

no parameters

3.3.3.2 Preconditions

no preconditions

3.3.3.3 Postconditions

• The state of this ModeOfOperation has been set to its initial state.

3.3.3.4 Sequence diagram

no sequence diagram

3.3.4 public setInitVector:void

This method is used to set the initialization vector that is needed by many modes of operation.

3.3.4.1 Parameters

iv – The initialization vector represented by a ByteArray.

3.3.4.2 Preconditions

• *iv* must be the same length as the block size of the BlockCipher associated with this ModeOfOperation.

3.3.4.3 Postconditions

• The initialization vector has been set to iv.

3.3.4.4 Sequence diagram

no sequence diagram

3.3.5 public useIV:boolean

This method tells weather this ModeOfOperation uses an initialization vector or not.

3.3.5.1 Parameters

no parameters

3.3.5.2 Preconditions

no preconditions

3.3.5.3 Postconditions

• Has returned true if this ModeOfOperation uses initialization vector, false otherwise.

3.3.5.4 Sequence diagram

no sequence diagram

3.4 <abstract> BlockCipher

This is a subclass of CryptoEngine which is used as a base class for implementations of different block ciphers. Block ciphers are used to encrypt or decrypt blocks of bytes of a specified length and it is subclasses of this class that do the actual encryption and decryption work when using the SymmetricCryptoEngine.

3.4.1 <abstract> public encrypt:ByteArray

see 3.1.1

3.4.1.1 Parameters

see 3.1.1.1

3.4.1.2 Preconditions

• *msg* must be of the same length as the block size.

3.4.1.3 Postconditions

- see 3.1.1.3
- The returned ByteArray has the same length as msg.

3.4.1.4 Sequence diagram

no sequence diagram

3.4.2 <abstract> public decrypt:ByteArray

see 3.1.2

3.4.2.1 Parameters

see 3.1.2.1

3.4.2.2 Preconditions

• c must be of the same length as the block size.

3.4.2.3 Postconditions

- see 3.1.2.3
- The returned ByteArray has the same length as c.

3.4.2.4 Sequence diagram

no sequence diagram

3.5 <abstract> PublicKeyCryptoSystem

This is a subclass to CryptoEngine that is used as a base class to implement different public key crypto systems.

3.5.1 <abstract> public encrypt:ByteArray

see 3.1.1

3.5.1.1 Parameters

see 3.1.1.1

3.5.1.2 Preconditions

• see 3.1.1.2

3.5.1.3 Postconditions

• if no KeyPair is associated with this PublicKeyCryptoSystem null has been returned otherwise see 3.1.1.3

3.5.1.4 Sequence diagram

no sequence diagram

3.5.2 <abstract> public decrypt:ByteArray

see 3.1.2

3.5.2.1 Parameters

see 3.1.2.1

3.5.2.2 Preconditions

• see 3.1.2.2

3.5.2.3 Postconditions

• if no KeyPair is associated with this PublicKeyCryptoSystem null has been returned otherwise see 3.1.2.3

3.5.2.4 Sequence diagram

no sequence diagram

3.6 <abstract> Padding

This is a base class for implementing different padding schemes that is used to fill up remaining bytes of the last block in a message when using a block cipher. A description of a padding scheme can be found in section 8.2.3.

3.6.1 <abstract> applyPadding:ByteArray

This method is used to apply padding to a msg.

3.6.1.1 Parameters

msg – The message that is going to be padded.

3.6.1.2 Preconditions

- blockSize must be set.
- msg must not be null

3.6.1.3 Postconditions

• A ByteArray, which length is a multiple of *blockSize* AND where the first bytes is the same as in *msg* and the rest are filled according to the padding scheme used, has been returned.

3.6.1.4 Sequence diagram

no sequence diagram

3.6.2 <abstract> removePadding:ByteArray

This method is used to remove padding from a msg.

3.6.2.1 Parameters

msg – The message that padding is going to be removed from.

3.6.2.2 Preconditions

- blockSize must be set.
- msg must not be null
- Length of msg must be a multiple of blockSize

3.6.2.3 Postconditions

- If padding were correctly applied to *msg a* ByteArray which has a length between *msg.length* (*blocksize* + 1) and *msg.length* has been returned. The bytes of the returned ByteArray was part of the original message and the removed bytes was added padding.
- Else an exception has been thrown.

3.6.2.4 Sequence diagram

no sequence diagram

3.7 <abstract> MAC

This is a base class for implementing different ways to compute a Message Authentication Code. More about MAC's can be found in chapter section 8.4.

3.7.1 <abstract> public computeMAC:ByteArray

This method is used to compute a MAC of a given message.

3.7.1.1 Parameters

msg – The message that a MAC is going to be computed for.

3.7.1.2 Preconditions

- msg must not be null.
- A Key must be associated with this MAC.

3.7.1.3 Postconditions

• A ByteArray containing the MAC of msg has been returned.

3.7.1.4 Sequence diagram

no sequence diagram

3.8 <interface> Hash

This is an interface for classes implementing cryptographic hash-functions. More about hash-functions and pseudo code for one hash-function can be found in chapter section 8.3.

3.8.1 public computeHash:ByteArray

This method is used to compute a hash value of specific message.

3.8.1.1 Parameters

msg – The message that the hash value is going to be computed for

3.8.1.2 Preconditions

• msq must not be null.

3.8.1.3 Postconditions

• A ByteArray containing the hash value.

3.8.1.4 Sequence diagram

no sequence diagram

3.9 SecureRandom

This class is used to provide cryptographically secure random numbers, or more specific random strings of bytes. Before an object of this class is used to generate random numbers it should be seeded from a good source of entropy. More about this and pseudo code for one method of generating pseudo random numbers can be found in chapter section 8.6.

3.9.1 public randomBytes:ByteArray

This method is used to generate a specific number of random bytes. This method should never be used if the SecureRandom object has not been properly seeded since the entropy of the data it produces in such a case is questionable.

3.9.1.1 Parameters

length – The number of bytes that are going to be generated

3.9.1.2 Preconditions

no preconditions

3.9.1.3 Postconditions

• A ByteArray containing *length* pseudo random bytes has been returned.

3.9.1.4 Sequence diagram

no sequence diagram

3.9.2 public seed:void

This method is used to seed the random number generator. More about how a good seed can be created from an entropy source in chapter section 8.6.1 and 8.6.2.

3.9.2.1 Parameters

entropy – A number of random bytes that this SecureRandom is seeded with.

3.9.2.2 Preconditions

no preconditions

3.9.2.3 Postconditions

This SecureRandom has been seeded and is ready for proper use.

3.9.2.4 Sequence diagram

no sequence diagram

3.10 BigInteger

This is a class that is used to represent integers that does not fit in a 32-bit integer. Such big integers are needed for public key cryptography that uses a mathematical problem that is hard to solve for very large numbers. This class also has the usual mathematical operations that are used by public key cryptography.

For use with the authentication scheme presented in section 8.1 it needs to be able to perform addition, multiplication, modulo, power and multiplicative inverse modulo x. For efficiency it also need to be able to calculate power and modulo with the same method since it is not practical to use with big numbers otherwise. It is also necessary with a method that can test an integer if it is a prime.

Except for these methods it should be possible to create a BigInteger from a ByteArray and also convert a BigInteger into a ByteArray. The way this is done must be compatible with the class BigInteger in the Java library.

It is also possible that other operations are needed to compute the operations above.

3.10.1 public add

Adds this BigInteger with x and returns the sum as a new BigInteger. Can be implemented using schoolbook addition.

3.10.1.1 Parameters

x – The second term.

3.10.2 public multiply

Multiplies this BigInteger with *x* and returns the product as a new BigInteger. Can be implemented using schoolbook multiplication but there are more efficient methods, as schönhage-strassen.

3.10.2.1 Parameters

x – The second factor.

3.10.3 public mod

Computes this BigInteger modulo *x* and returns the result as a new BigInteger. This is the same as remainder of this divided by x except that if the remainder is negative, x should be added to it to give a positive modulo. Normal integer schoolbook division can be used for this.

3.10.3.1 Parameters

x – The modulo that this is going to be reduced to.

3.10.4 public pow

Computes this exp and returns the result as a new BigInteger.

3.10.4.1 Parameters

exp – The exponent. It should not be possible for this value to be bigger than a normal 32-bit int because of memory issues.

3.10.5 public modPow

Computes this exp modulo mod and returns the result as a new BigInteger.

3.10.5.1 Parameters

exp – The exponent.

mod – The modulo.

3.10.6 public modInverse

Calculates the multiplicative inverse of this modulo x and returns the result as a new BigInteger. The inverse this $^{-1}$ is satisfies this * this $^{-1}$ mod x.

3.10.6.1 Parameters

x – The modulo of this inverse.

3.10.7 public isProbablyPrime

Tests if this is a prime and returns true if the probability of that is high and false otherwise.

4 Server network module

The design of the network module (figure 15) of the server is similar to the design of the network module of the client. The differences is that the ConnectionManager is made for receiving incoming connections as well as keeping track of multiple Sessions and only the receive method is application dependent where in the client all methods of the ConnectionManager. Also there are differences because the server and the client is implemented in different languages, for example since it is not possible to pass methods as method parameters in Java the two different methods that takes care of incoming messages must be contained in classes instead. This is done with two internal classes to Session, AuthHandler and SecureChannel, that both implement an internal interface, MessageHandler. Also the class SCTask that is used for resending of messages that has gone over time is an internal class to Session.

The goal of this design has been to make the server and the client as similar as possible since that will make it easier to implement and understand both sides of the network modules. This also reflects on the activity diagram (figure 16) that is describing the workings of the server that is almost exactly the same. The only differences are that it shows that the server can connect to multiple clients and a user failing to authenticate a number of times in a row should have its account locked.

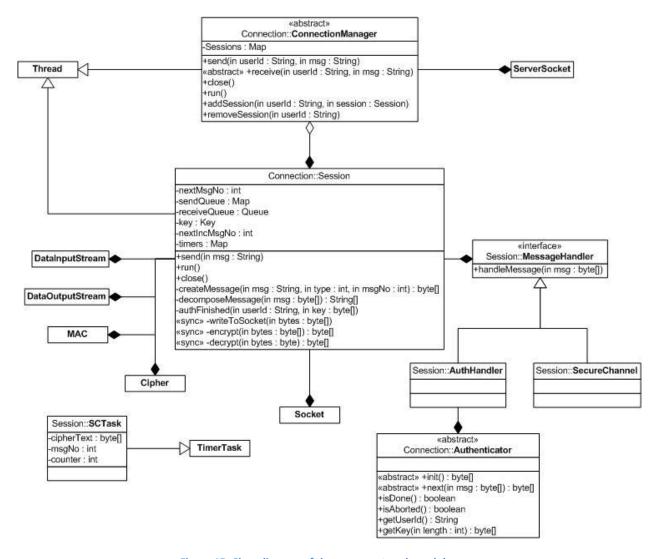


Figure 15: Class diagram of the server network module

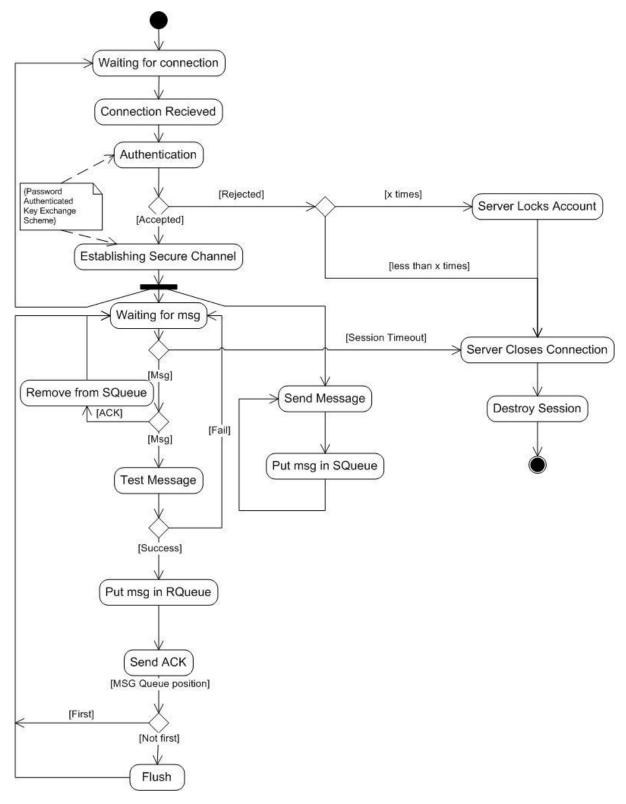


Figure 16: Activity diagram of the server network module

4.1 <abstract> ConnectionManager

This class has multiple purposes, it acts as the interface between the different Sessions that are established and other modules of the application and also handles new incoming connections. ConnectionManager is a subclass to thread to be able to use a separate thread for waiting for new incoming connections.

4.1.1 public send:void

This method is used by the application to send a message to a specified user. It is actually only forwarding the message to the Session that is associated with the userId specified by the sending thread.

4.1.1.1 **Parameters**

userId – The userId that is associated with the Session that is going to be used to send this message. msg – The message that is going to be sent.

4.1.1.2 Preconditions

msg must not be null.

4.1.1.3 Postconditions

• If there was a Session associated with *userId*, *msq* has been sent using this Session.

4.1.1.4 Sequence diagram

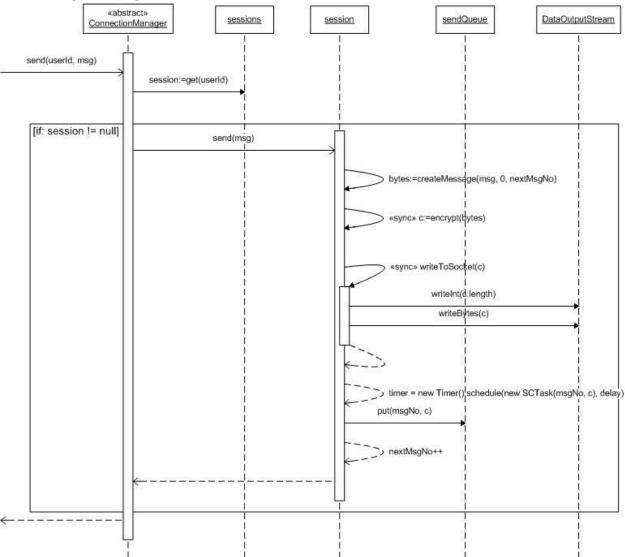


Figure 17: Sequence diagram over send(userId:String, msg:String):void

4.1.2 <abstract> public receive:void

This method is called by a session when it has received a message from its associated Socket. It should be implemented to do what the application should do when a message has been received.

4.1.2.1 Parameters

userId – The id of the user that is associated with the Session that made the call to this method. *msg* – The message that has been received.

4.1.2.2 Preconditions

- msq must not be null.
- userId must be previously registered to be associated with a specific Session.

4.1.2.3 Postconditions

postconditions depends on the application

4.1.2.4 Sequence diagram

no sequence diagram

4.1.3 public close:void

This method is used to close the connections to all clients and to close the thread that is waiting for new connections. This method should be used when the server is shutdown.

4.1.3.1 Parameters

no parameters

4.1.3.2 Preconditions

no preconditions

4.1.3.3 Postconditions

- The attribute *closed* has been set to true.
- All client connections have been closed, by invoking the *close* method of the Session's associated with this ConnectionManager.
- The ServerSocket associated with this ConnectionManager has been closed.

4.1.3.4 Sequence diagram

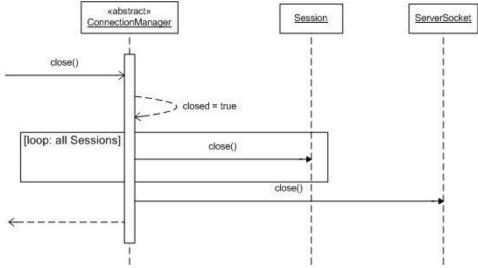


Figure 18: Sequence diagram over cloes():void

4.1.4 public run:void

This method, which is inherited from Thread, is used to receive new incoming connections. It consists mainly of a loop that can only be stopped by setting the attribute *closed* to false which uses a ServerSocket to accept new connections. For each connection that is accepted a new Session is created.

4.1.4.1 Parameters

no parameters

4.1.4.2 Preconditions

no preconditions

4.1.4.3 Postconditions

• for each round of the loop a Session has been created and started.

4.1.4.4 Sequence diagram

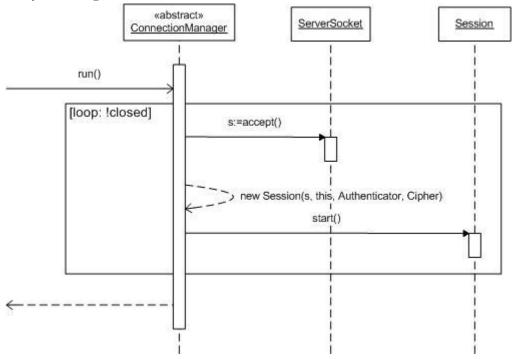


Figure 19: Sequence diagram over run():void

4.1.5 public addSession:void

This method is used to add a Session to the collection of Session's that the ConnectionManager maintains. It is used by the Session class when the authentication phase has been finished. This method also makes sure that a user only can be connected through one Session at once, if a user associated with a Session trying to add itself when the user already is registered to be associated with another Session; the new Session will be closed.

4.1.5.1 Parameters

userId – The user id that is associated to the Session trying to add itself. *session* – The Session that is going to be added.

4.1.5.2 Preconditions

session must have finished authentication.

4.1.5.3 Postconditions

- If sessions was not containing any post with userId as key session has been added with userId as key.
- If the attribute sessions was containing a post with userId as key session has been closed.

4.1.5.4 Sequence diagram

no sequence diagram

4.1.6 public removeSession:void

This method is used to remove Session's from those that are associated with this ConnectionManager. It is called by a Session when it closes.

4.1.6.1 Parameters

userId – The user id associated with the Session that wants to remove itself.

4.1.6.2 Preconditions

no preconditions

4.1.6.3 Postconditions

• If sessions was containing a post with userId as key that post has been removed.

4.1.6.4 Sequence diagram

no sequence diagram

4.2 Session

This is the class that handles the connection to a client. It runs in two different states, first when created until authentication is done it will run in authentication state and pass all received messages to the authenticator that is associated with it. When that is done, if the authentication was accepted it will switch to secure channel state where it can exchange messages with the server without confidentiality and integrity risks.

4.2.1 Constructor

The constructor of Session should:

• Initialize the Authenticator and if a message was produced send it to the server using the writeToSocket method.

4.2.2 public send:void

This method is used to send messages to a client that the Socket that is associated with this Session is connected to. Using this method ensures that the message is sent over a secure channel that protects its confidentiality and integrity. To accomplish this, the message is encrypted (see section 8.2) to protect its confidentiality and a message digest (see section 8.3) is also sent with the message to ensure its integrity.

4.2.2.1 Parameters

msg – The message that is going to be sent.

4.2.2.2 Preconditions

• The authentication procedure must be finished.

4.2.2.3 Postconditions

- The message has been encrypted and sent.
- A timer that will resend the message after a set amount of time unless stopped has been created and started.

4.2.2.4 Sequence diagram

see section 4.1.1.4

4.2.3 public run:void

This method is inherited from Thread and is for listening on the Socket (DataInputStream) for incoming messages from the client. It uses a loop to continue to run until the attribute *closed* has

been set to false. For each round of the loop a message is received, more about how messages is received is described in the description of the client method readFromSocket found in section 2.1.10 of this document.

4.2.3.1 Parameters

no parameters

4.2.3.2 Preconditions

no preconditions

4.2.3.3 Postconditions

- For each round of the loop
 - o If a correct message was received it has been forwarded to the MessageHandler.
 - Otherwise the buffer has been cleared.

4.2.3.4 Sequence diagram

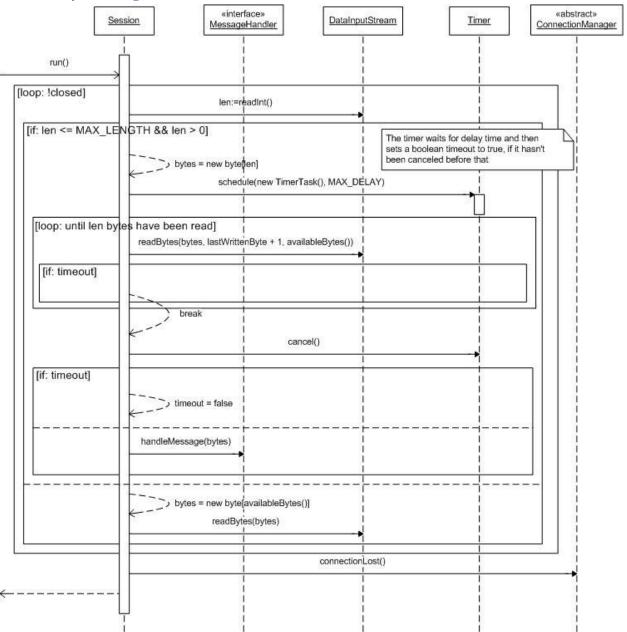


Figure 20: Sequence diagram over run():void

4.2.4 public close:void

This method is used to close the Session. If the client closes the connection this method shall be called automatically.

4.2.4.1 Parameters

no parameters

4.2.4.2 Preconditions

no preconditions

4.2.4.3 Postconditions

- The attribute *closed* has been set to true.
- The Socket has been closed.

4.2.4.4 Sequence diagram

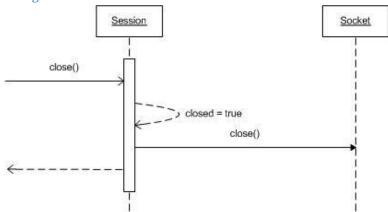


Figure 21: Sequence diagram over close():void

4.2.5 private createMessage:byte []

see section 2.1.3

4.2.6 private decomposeMessage:String []

see section 2.1.4

4.2.7 private authFinished:void

This method is called by the AuthHandler when the authentication is finished.

4.2.7.1 Parameters

userId – The user id that the user of the client has identified itself with.

key – The key that has been derived from the authentication procedure.

4.2.7.2 Preconditions

- The authentication must be finished, which means that the isDone method of the Authenticator must be true.
- key must be a valid key of appropriate length.
- userId must not be null.

4.2.7.3 Postconditions

- The Thread.name associated with this Session has been set to userId.
- The key associated with this Session has been set to *key*, and doing this the key has been converted from a byte array to a SecretKeySpec that is of appropriate type for the Cipher used in this instance of Session
- This Session has been added to the ConnectionManager that is associated to it.

4.2.7.4 Sequence diagram

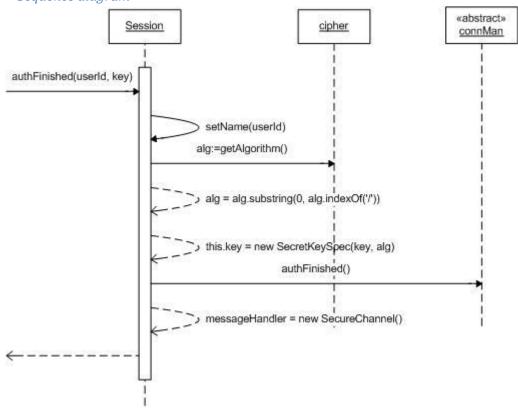


Figure 22: Sequence diagram over authFinished():void

4.2.8 <sync> private writeToSocket:void

This method is used to write messages to the DataOutputStream (Socket) associated with the Session. This method is synchronized so that it will not be possible that different threads in the server application accidently send messages that are mixed together.

4.2.8.1 Parameters

bytes – An array of bytes that is going to be sent the client.

4.2.8.2 Preconditions

• bytes must not be null.

4.2.8.3 Postconditions

- An int containing the length of bytes has been written to the DataOutputStream.
- After that all bytes in bytes has been written to the DataOutputStream.

4.2.8.4 Sequence diagram

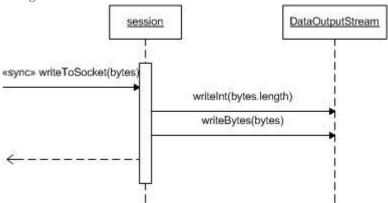


Figure 23: Sequence diagram over writeToSocket(bytes:byte[]):void

4.2.9 <sync> private encrypt:byte []

This method is used to encrypt a message using the Cipher that is associated with this Session. It is synchronized since the Cipher must be set up before every encryption or decryption operation which means it must be done synchronously in a multithreaded application.

4.2.9.1 Parameters

bytes – The bytes that is going to be encrypted.

4.2.9.2 Preconditions

- A Key must be associated with this Session.
- bytes must not be null.

4.2.9.3 Postconditions

- The Cipher has been initialized for encryption with the Key associated with Session
- A byte array containing initialization vector and cipher text has been returned.

4.2.9.4 Sequence diagram

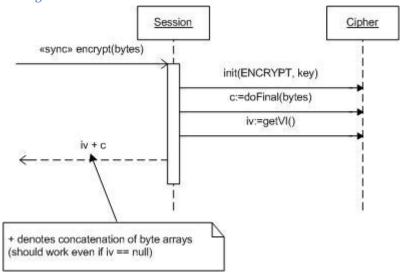


Figure 24: Sequence diagram over encrypt(bytes:byte[]):byte[]

4.2.10 <sync> private decrypt:byte []

This method is used to decrypt a message using the Cipher that is associated with this Session. It is synchronized since the Cipher must be set up before every encryption or decryption operation which means it must be done synchronously in a multithreaded application.

Note that this method doesn't work with Cipher's that does not use initialization vectors, for example when mode of operation is set to ECB (Electronic Code Book) mode.

4.2.10.1 Parameters

bytes – The bytes that is going to be decrypted

4.2.10.2 Preconditions

- A Key must be associated with this Session.
- bytes must not be null.

4.2.10.3 Postconditions

- The Cipher has been initialized for decryption with the Key associated with Session and the first block of *bytes* as initialization vector.
- A byte array containing plain text has been returned.

4.2.10.4 Sequence diagram

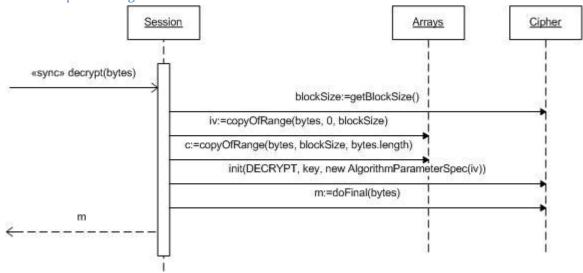


Figure 25: Sequence diagram over decrypt(bytes:byte[]):byte[]

4.3 SCTask

This class is a subclass to TimerTask and an internal class of Session. It is used to resend messages when a Timer it is associated to have waited a set delay. If the message has been resent too many times it will shut down the connection.

4.3.1 public run:void

This method is only executed if the Timer that this SCTask is associated to has waited a set delay. When executed this message resends the message that is associated to it and then starts a new Timer.

4.3.1.1 Parameters

no parameters

4.3.1.2 Preconditions

This SCTask must be associated with a Timer that is calling this method

4.3.1.3 Postconditions

- If counter was less than the max number of resends
 - The cipher text associated with this SCTask has been written to the Socket associated with the Session that this SCTask is associated to.
 - o The timer has been restarted with this SCTask associated to it.
- If *counter* was equal or more than the max number of resends the Session that this SCTask is associated to has been closed.
- counter has been incremented by one.

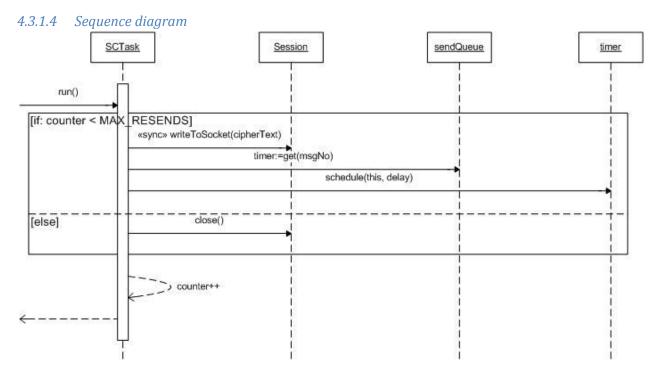


Figure 26: Sequence diagram over run():void

4.4 <interface> MessageHandler

This interface is internal to Session and is used to define the classes that handles incoming messages in different states of the Session.

4.4.1 public handleMessage:void

This method is used to handle a message, it is called every time the Session has successfully read a message from the DataInputStream (Socket) and handles the message depending on which implementation that is currently used.

4.4.1.1 Parameters

msg – The message that is going to be handled.

4.4.1.2 Preconditions

• *msg* must not be null.

4.4.1.3 Postconditions

implementation dependant

4.4.1.4 Sequence diagram

implementation dependant

4.5 AuthHandler

This class is a subclass to MessageHandler and an internal class to Session. It is used to handle messages during the authentication procedure.

4.5.1 handleMessage:void

This method is used to handle messages during the authentication phase of the Session. It usually only forwards messages to the Authenticator that is associated with this AuthHandler and then sends the return messages from the Authenticator to the client. It also checks if the authentication is done or aborted and takes appropriate steps if any of these are true.

4.5.1.1 Parameters

see 4.4.1.1

4.5.1.2 Preconditions

see 4.4.1.2

4.5.1.3 Postconditions

- The message received has been handled by the Authenticator
- If a return message was produced it has been sent to the client.
- If the authentication was aborted the Session that this AuthHandler is associated to has been closed
- If the authentication was done the authFinished method of the Session that this AuthHandler is associated to has been called.

4.5.1.4 Sequence diagram

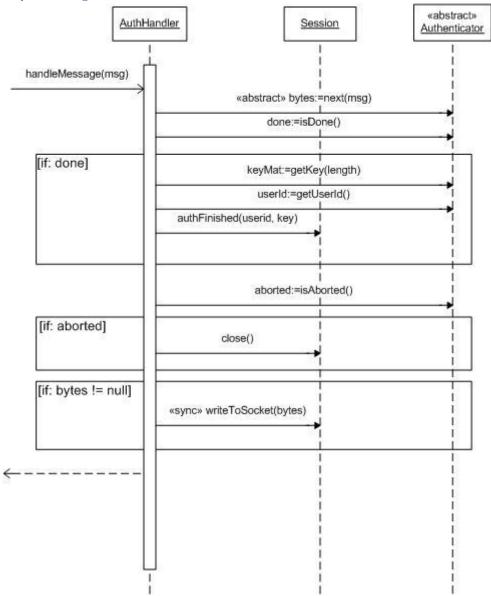


Figure 27: Sequence diagram over handleMessage(msg:byte[]):void

4.6 SecureChannel

This class is a subclass to MessageHandler and an internal class to Session. It is used to handle messages when the Session is in secure channel state.

4.6.1 public handle Message:void

This method is used to handle messages that are received after the authentication has been finished.

4.6.1.1 Parameters

see 4.4.1.1

4.6.1.2 Preconditions

see 4.4.1.2

4.6.1.3 Postconditions

- If the message digest of the message was correct:
 - o If the message received was of type MSG it was put in the received-queue and if it was first in this queue it has been forwarded to the ConnectionManager that is associated with the Session that this SecureChannel is associated to. Being first in the queue is the same as having the same message number as the attribute nextIncMsgNo. Also if any other message(s) gets first in the queue after this one has been forwarded they are forwarded too.
 - o If the message received was of type ACK the timer with the same message number has been stopped and removed from the send queue.

4.6.1.4 Sequence diagram

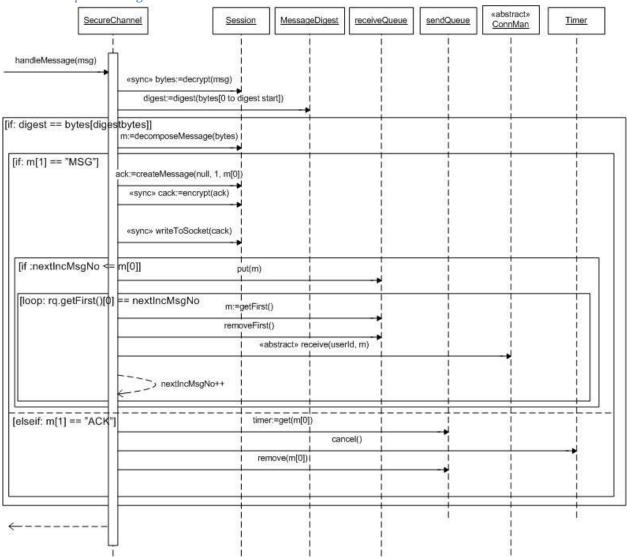


Figure 28: Sequence diagram over handleMessage(msg:byte[]):void

4.7 <abstract> Authenticator

see 2.2

4.7.1 <abstract> public init:Byte[]

see 2.2.1

4.7.1.1 Parameters no parameters4.7.1.2 Preconditions

4.7.1.3 Postconditions

see 2.2.1.3

see 2.2.1.2

4.7.1.4 Sequence diagram

no sequence diagram

4.7.2 <abstract> public next:Byte[]

see 2.2.2

4.7.2.1 Parameters

see 2.2.2.1

4.7.2.2 Preconditions

see 2.2.2.2

4.7.2.3 Postconditions

see 2.2.2.3

4.7.2.4 Sequence diagram

no sequence diagram

4.7.3 public getKey:byte []

see 2.2.3

4.7.3.1 Parameters

see 2.2.3.1

4.7.3.2 Preconditions

see 2.2.3.2

4.7.3.3 Postconditions

see 2.2.3.3

4.7.3.4 Sequence diagram

no sequence diagram

5 Data storage module

This module is used for secure data storage and is designed especially for this application but is also possible to use in applications that save their data in a similar way. Since clients are being sent all data that they are authorized to use it is not necessary to have any visible structure in the saved file which is a positive thing since even such structure can give an adversary that manages to steal the file some information about its contents. The problem with this is that encrypting and decrypting a lot of data requires much computational power but it can be solved by limiting the number of read and write operations. The number of read operations is naturally low in this application since data is only read when a user has finished authentication and the number of write operations is limited by introducing a queue that stores data that is going to be written until it can be written to the file. The data in the queue will then be encrypted and written to the file at a set interval that will be long enough to not make the server encumbered with too many write operations to slow it down.

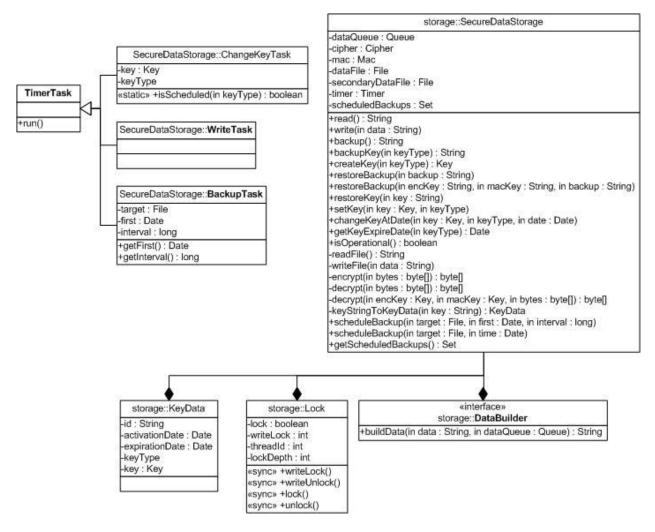


Figure 29: Class diagram over the data storage module.

The data storage module consists of one class, SecureDataStorage, that is the interface to the module and a few helper classes, see figure 29. The main functionality of the module is to read and write to an encrypted data file but there are also a number of administrative operations like changing keys, making backups on both data and keys and a number of other things that are described separately below.

The module is not dependent on the format of the data since this format can be specified by the user of the module by implementing the DataBuilder interface.

5.1 SecureDataStorage

This class is the center of the data storage module since it performs most of the tasks of the module by itself and is also the interface to other modules. It has two methods that is used for reading and writing to the SecureDataStorage but also administrative methods that is used for making backups of the internal state of the object and also for restoring it using these backups.

The internal state of the class consists of a pair of keys, one for encryption/decryption and one for computing MAC's, and a queue that stores data that is yet to be written to the data file. While the data file is not a part of the internal state of the SecureDataStorage object it is part of the state of data storage model since it contains all the data that is securely stored on the disk. The confidentiality of the data in the file is protected by encryption using the AES block cipher (see section 8.2) and its integrity is protected by a HMAC using SHA-256 (see section 8.4). Both cipher text and MAC is stored in the data file.

To increase the robustness of the system the data file is actually two identical files. If one file gets corrupted it is still possible to recover without using a backup file and it can be done without interruption of the service. Mostly read operations will only read from one file while write operations have to update both files every time. For the rest of this section these files will be mentioned as a single file.

All backups created by SecureDataStorage is represented by a string in the application but should be saved to some file. How backups of keys should be handled can be read about more in section 8.5 and the layout of the backups can be found in section 7.

5.1.1 public read:String

This method is used to read data from the SecureDataStorage. It will read all data in the file and *queue* and put it together using the associated DataBuilder before returning it all as a string. NOTE1: no actual reading should be allowed from either file or queue if some other thread is writing at the same time. NOTE2: data from the file need to be decrypted before it can be put together with the data from the queue.

5.1.1.1 Parameters

no parameters

5.1.1.2 Preconditions

- An active key of type encryption must be set and it must not have expired.
- An active key of type MAC must be set and it must not have expired.

5.1.1.3 Postconditions

- If integrity check on the file passed
 - All data contained in the data file and queue has been returned as a string.
- Else
 - An IntegrityFailException has been thrown.

5.1.1.4 Sequence diagram

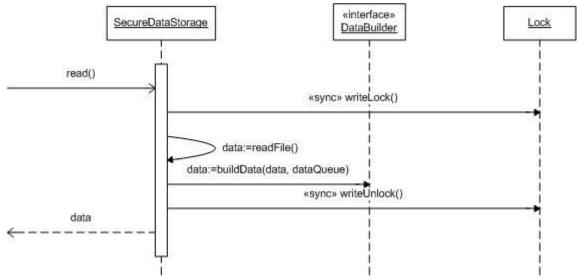


Figure 30: Sequence diagram over read():string

5.1.2 public write:void

This method is used to write data to the SecureDataStorage. This method will not actually write anything to the data file but only add it to *queue* to wait for the next time data is written to the file. NOTE: no writing should be done unless no other write or read operations is active.

5.1.2.1 Parameters

data – The data that is going to be written to the SecureDataStorage.

5.1.2.2 Preconditions

- An active key of type encryption must be set and it must not have expired.
- An active key of type MAC must be set and it must not have expired.

5.1.2.3 Postconditions

• data has been added to queue.

5.1.2.4 Sequence diagram

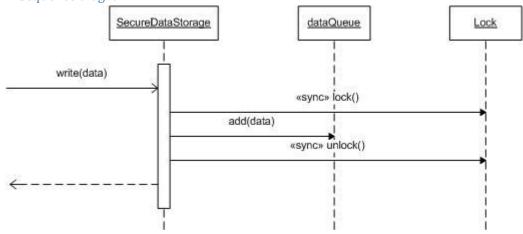


Figure 31: Sequence diagram over write(data:string):void

5.1.3 public backup:String

This method is used to backup the data that is stored in the SecureDataStorage. This includes both the data file and the queue. NOTE: no read operations should be performed during a write operation.

5.1.3.1 Parameters

no parameters

5.1.3.2 Precondition

- An active key of type encryption must be set and it must not have expired.
- An active key of type MAC must be set and it must not have expired.

5.1.3.3 Postconditions

• The data in the data file and the queue has been returned as a string that is preceded by a header which includes identifiers for the keys that this backup is protected by and the date this backup was made. The format of this header can be found in section 7.

5.1.3.4 Sequence diagram

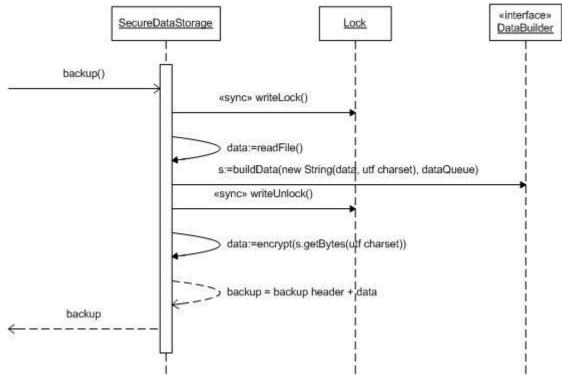


Figure 32: Sequence diagram over backup():string

5.1.4 public backupKey:String

This method is used to backup the keys that are used to protect the data in the SecureDataStorage. A backed up key is a string representation of the key. This string consists of a key label and the key itself in encoded format. The key label is used to identify the key and consists of an identifier, key type and the period that this key is valid.

5.1.4.1 Parameters

keyType – This can either be MAC or encryption key.

5.1.4.2 Preconditions

• An active key of type *keyType* must be set.

5.1.4.3 Postconditions

• A string containing key label and encoded key has been returned. The format of the key label can be found in section 8.5.1.

5.1.4.4 Sequence diagram

no sequence diagram

5.1.5 public static createKey:Key

This method is used to create a new key that can be used as either an encryption key or a MAC key. The key is created by generating a random byte array of appropriate length. And then the class SecretKeySpec is used to turn it into a key of the correct type.

5.1.5.1 Parameters

keyType – This can either be MAC or encryption key.

5.1.5.2 Preconditions

no preconditions

5.1.5.3 Postconditions

• A new Key of type *keyType* has been returned.

5.1.5.4 Sequence diagram

no sequence diagram

5.1.6 public restoreBackup:void

This method is used to restore a backup that was created with the same keys that is active in the SecureDataStorage now. NOTE1: by restoring a backup the current data file and *queue* will be cleared and replaced by the data in the backup. NOTE2: it should not be possible to do any read or write operations while the data file and *queue* is in an inconsistent state.

5.1.6.1 Parameters

backup – The backup that is going to be restored

5.1.6.2 Preconditions

backup must not be null.

5.1.6.3 Postconditions

- If the keys that were used for backup are those that are active in SecureDataStorage.
 - o If the integrity check for *backup* is okay.
 - The data file has been replaced by the content in *backup*.
 - queue has been cleared.
 - Else an IntegrityFailException has been thrown.
- Else an IllegalArgumentException has been thrown.

5.1.6.4 Sequence diagram

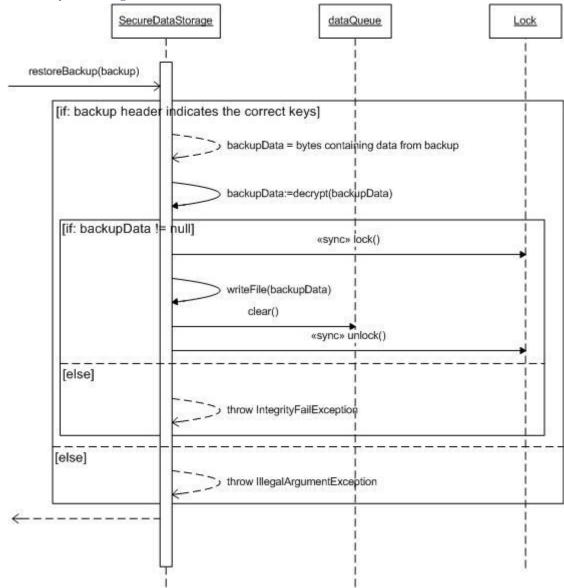


Figure 33: Sequence diagram over restoreBackup(backup:string):void

5.1.7 public restoreBackup:void

This method is used to restore a backup that was created by keys that are not currently in use. See 5.1.6 for more description. NOTE: no read or write operations should be allowed when the state of SecureDataStorage is inconsistent.

5.1.7.1 Parameters

encKey – The encryption key that was used to encrypt the backup.macKey – The key that was used to create the MAC that is ensuring the integrity of the backup

backup – The backup that is going to be restored

5.1.7.2 Preconditions

- encKey must not be null.
- MACKey must not be null.
- backup must not be null.

5.1.7.3 Postconditions

- If encKey and MACKey are valid key strings and they are the keys that were used for backup.
 - o If the integrity check for *backup* is okay.
 - The data file has been replaced with the content in *backup* and encrypted with the keys that are currently active in the SecureDataStorage.
 - queue has been cleared.
 - o Else an IntegrityFailException has been thrown.
- Else an IllegalArgumentException has been thrown.

5.1.7.4 Sequence diagram

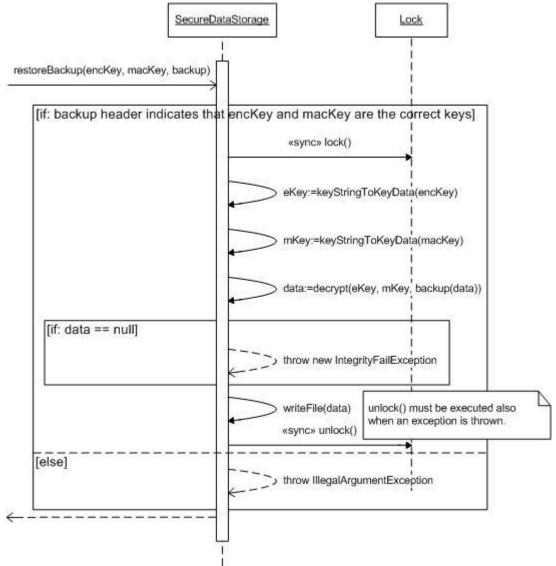


Figure 34: Sequence diagram over restoreBackup(encKey:string, macKey:string, backup:string):void

5.1.8 public restoreKey:void

This method is used to restore a backed up key. The key that is restored will become the active key of the type that is specified in its key label. NOTE: no read or write operations should be allowed when the state of SecureDataStorage is inconsistent.

This method is mainly intended to be used when setting up the system for example after a restart since it does not do anything to the data file. This means that the restored keys must be the keys that the data file is encrypted with or an integrity failure will occur the next time a read or write operation is performed. The method that is intended to be used to change keys during operation is changeKeyAtDate.

5.1.8.1 Parameters

key – A key in the form of a backed up key string.

5.1.8.2 Preconditions

key must not be null.

5.1.8.3 Postconditions

- If key is a valid key
 - o key has been set as the active key of the type that is specified in the key label of key.
- Else an IllegalArgumentException has been thrown.

5.1.8.4 Sequence diagram

no sequence diagram

5.1.9 public setKey:void

This method is used to set a key that is going to be the active key of the specified type. NOTE: no read or write operations should be allowed when the state of SecureDataStorage is inconsistent.

This method should mainly be used to set a newly created key as the active key on a newly started server or if the old key has expired.

5.1.9.1 Parameters

key – The that is going to be set as the active key. keyType – The type of key.

5.1.9.2 Preconditions

- key must not be null.
- *keyType* must be either encryption or MAC.

5.1.9.3 Postconditions

- *key* has been set as the active key of type *keyType*.
- If both types of keys now have been set the data file has been re-encrypted so that it can be used with the new pair of keys. This does not mean that setKey has to be used for both keys when it is used, only that a new pair of keys with one old and one new has been set.

5.1.9.4 Sequence diagram

no sequence diagram

5.1.10 public changeKeyAtDate:void

This method is used to set a key of a specified type that will become the active key of that type at a specified date.

5.1.10.1 Parameters

key – The key that is going to be set as a scheduled key.
 keyType – The type of key that is going to be scheduled for change.
 date – The date/time when the change will occur.

5.1.10.2 Preconditions

- key must not be null.
- keyType must be either encryption or MAC.
- date must not be null and must be a future date.

5.1.10.3 Postconditions

• key has been set as the key that is scheduled to be set as the active keyType key at date.

5.1.10.4 Sequence diagram

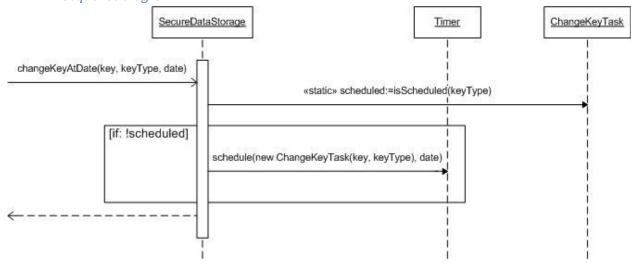


Figure 35: Sequence diagram over changeKeyAtDate(key:Key, keyType:KeyType, date:Date):void

5.1.11 public getKeyExpireDate:Date

This method is used to find out when the key of a specified type have reached the end of its active period (see section 8.5 for more information).

5.1.11.1 Parameters

keyType – The type of the key that the expiration date is going to be returned for.

5.1.11.2 Preconditions

no preconditions

5.1.11.3 Postconditions

- If there is an active key of type *keyType* the date when its crypto period expires has been returned.
- Else null has been returned.

5.1.11.4 Sequence diagram

no sequence diagram

5.1.12 public isOperational:Boolean

This method is used to check if the SecureDataStorage is in a state where it is possible to read or write to it. If read or write operations is used when this method returns false the SecureDataStorage may execute in an unintended way which may cause errors.

5.1.12.1 Parameters

no parameters

5.1.12.2 Preconditions

no preconditions

5.1.12.3 Postconditions

- True has been returned if:
 - o An active encryption key is set and its expire date is more than the current date.
 - An active MAC key is set and its expire date is more than the current date.
- Else false has been returned.

5.1.12.4 Sequence diagram

no sequence diagram

5.1.13 private encrypt:byte []

This method is used to encrypt a byte array and to create a MAC that is attached to the end of the result.

5.1.13.1 Parameters

bytes – The bytes that is going to be encrypted.

5.1.13.2 Preconditions

no preconditions

5.1.13.3 Postconditions

- A byte array has been returned, it consists of:
 - o Initialization vector of length the same as the block size of the cipher.
 - o The encrypted bytes that represent bytes.
 - o The MAC computed from *bytes* of length specified by the MAC-algorithm that is used.

5.1.13.4 Sequence diagram

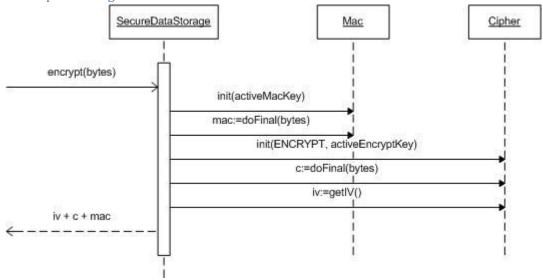


Figure 36: Sequence diagram over encrypt(bytes:byte[]):byte[]

5.1.14 private decrypt:byte []

This method is used to decrypt and check the integrity of the encrypted bytes using the provided keys.

5.1.14.1 Parameters

encKey – The key that is going to be used for decryption

macKey – The key that is going to be used for calculating the MAC of the decrypted message.

bytes – The bytes that is going to be decrypted, according to the format presented in 5.1.13.3.

5.1.14.2 Preconditions

no preconditions

5.1.14.3 Postconditions

- If the calculated MAC was equal to the MAC contained in *bytes* the decrypted data has been returned.
- Else null has been returned.

5.1.14.4 Sequence diagram

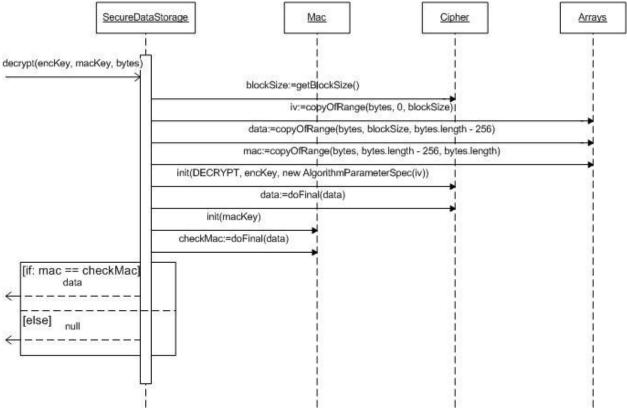


Figure 37: Sequence diagram over decrypt(encKey:Key, macKey:key, bytes:byte[]):byte[]

5.1.15 private decrypt:byte []

This method is the same as 5.1.14 except that it always uses the *activeEncryptKey* for decryption and the *activeMacKey* for calculating the MAC.

5.1.15.1 Parameters

bytes – The bytes that is going to be decrypted, according to the format presented in 5.1.13.3.

5.1.15.2 Preconditions

no preconditions

5.1.15.3 Postconditions

- If the calculated MAC was equal to the MAC contained in bytes the decrypted data has been returned.
- Else null has been returned.

5.1.15.4 Sequence diagram

no sequence diagram

5.1.16 private keyStringToKeyData:KeyData

This method converts a backup key string to a KeyData object, according to the format listed in section 8.5.1.

5.1.16.1 Parameters

key – The string that represents the key.

5.1.16.2 Preconditions

no preconditions

5.1.16.3 Postconditions

- If key is a valid key string a KeyData object has been returned containing the info from key.
- Else an IllegalArgumentException has been thrown.

5.1.16.4 Sequence diagram

no sequence diagram

5.1.17 public scheduleBackup:void

This method is used to schedule backups beginning at a set time and then continues at a set interval.

5.1.17.1 Parameters

target – Specifies the directory where the backups will be stored.

first – Specifies the time for the first backup.

interval – Specifies the interval between backups in milliseconds.

5.1.17.2 Preconditions

- target must be a directory.
- first must be a date in the future.

5.1.17.3 Postconditions

- A new BackupTask has been created and the attributes of it has been set to the same values as the parameter with the same name.
- The BackupTask has been scheduled on *timer* with the first execution at *first* and an interval of *interval*.

5.1.17.4 Sequence diagram

no sequence diagram

5.1.18 public scheduleBackup:void

This method is used to schedule a backup at a set time.

5.1.18.1 Parameters

target – Specifies the directory where the backups will be stored.

time – Specifies the time for the backup.

5.1.18.2 Preconditions

- target must be a directory.
- time must be a date in the future.

5.1.18.3 Postconditions

- A new BackupTask has been created and its attributes *target* and *first* has been set to the same values as the parameters *target* and *time*. The attribute *interval* has been set to zero.
- The BackupTask has been scheduled on *timer* to execute at *time*.

5.1.18.4 Sequence diagram

no sequence diagram

<interface>DataBuilder

This interface is a template for how the data file is going to be put together. Its only method is used when the data in the SecureDataStorage *queue* is combined with the data in the data file. The reason for that there is only an interface for this is that it is rather application specific how the structure of the data is made, and the implementation of this does not belong to the data storage module.

5.1.19 public buildData:String

This method is used to put together data in a string with data in a queue.

5.1.19.1 Parameters

data – The data is already stored in the correct structure.

queue – A queue that includes new data that is going to be inserted into the storage structure.

5.1.19.2 Preconditions

no preconditions

5.1.19.3 Postconditions

• A string containing the data from *data* and *queue* with the same structure as *data* has been returned.

5.1.19.4 Sequence diagram

no sequence diagram

5.2 WriteTask

This class is a subclass of TimerTask and an internal class of SecureDataStorage. It is used to update the data file of SecureDataStorage with the data that is temporarily stored in *queue*. The run method of this class shall be executed in a set interval as long as the SecureDataStorage is in scope.

5.2.1 public run:void

The task of this method is to update the data file with the data that is stored in *queue* of the SecureDataStorage class that this WriteTask is associated with. It is only supposed to be called by a timer that that will call it by a set interval until the SecureDataStorage is out of scope/garbage collected. NOTE: no read or write operations should be allowed when the state of SecureDataStorage is inconsistent.

5.2.1.1 Parameters

no parameters

5.2.1.2 Preconditions

no preconditions

5.2.1.3 Postconditions

- All data that is stored in *queue* of the SecureDataStorage that this WriteTask is associated with has been added to the data file.
- queue is empty.

5.2.1.4 Sequence diagram

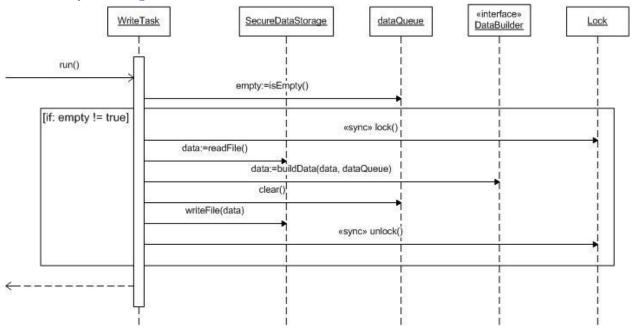


Figure 38: Sequence diagram over run():void

5.3 ChangeKeyTask

This is a subclass of TimerTask and an internal class of SecureDataStorage that is used to change the keys that are active. This task is executed by a timer that has been set by the changeKeyAtDate method and will set the key that is associated to this ChangeKeyTask to the active key of a specified type of the SecureDataStorage that this ChangeKeyTask is associated to.

5.3.1 public run:void

This method will change a key of the SecureDataStorage that this ChangeKeyTask is associated to when it is executed. The key that will be changed is the active key of the type that is specified in this ChangeKeyTask and it will be changed to the key that is associated to this ChangeKeyTask. NOTE: no read or write operations should be allowed when the state of SecureDataStorage is inconsistent.

5.3.1.1 Parameters

no parameters

5.3.1.2 Preconditions

no parameters

5.3.1.3 Postconditions

 key has been set as the active key of type keyType of the SecureDataStorage that this ChangeKeyTask is associated to.

5.3.1.4 Sequence diagram

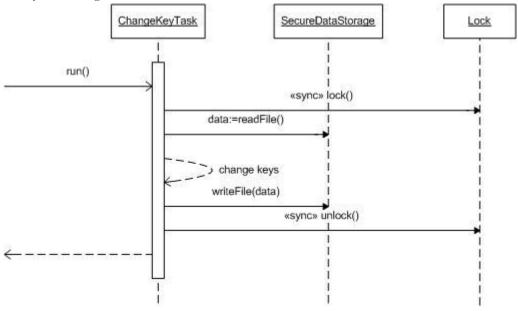


Figure 39: Sequence diagram over run():void

5.4 BackupTask

This is subclass of TimerTask that is used to backup the data file at a specified point in time. Its only task is to execute the backup method of the SecureDataStore and store the backup in a specified folder.

The File object that is associated with this class must be a directory.

5.4.1 public run:void

This method will be executed when the Timer this BackupTask is associated to has reached the scheduled execution time. When executed it will create a backup and save it in the specified folder.

5.4.1.1 Parameters

no parameters

5.4.1.2 Preconditions

• target must be a directory

5.4.1.3 Postconditions

• A backup string according to the format presented in section 7 has been created and saved to a file in the directory specified by *target*.

5.4.1.4 Sequence diagram

no sequence diagram

5.5 KeyData

This class is used to store key data for active keys in SecureDataStorage and keys that are associated with a ChangeKeyTask. All attributes are set at the construction of the class and cannot be changed after that, all of them can be accessed by getters.

5.6 Lock

This class is used by the SecureDataStorage to prevent read and write operations to execute simultaneously with another write operation and to prevent write operations to execute simultaneously with read operations. This is needed to make it possible to use the storage module in a multi-threaded environment.

All lock methods can handle multiple calls by the same thread, without causing a deadlock.

5.6.1 <sync> public lock:void

This method will cause all consecutive calls to lock or writeLock to block until unlock has been called by the thread that used this method.

5.6.1.1 Parameters

no parameters

5.6.1.2 Preconditions

 The thread that is calling this method has not previously called writeLock without calling writeUnlock.

5.6.1.3 Postconditions

- threadId has been set to the id of the calling thread.
- *lock* has been set to true.
- lockDepth has been increased by 1.

5.6.1.4 Sequence diagram

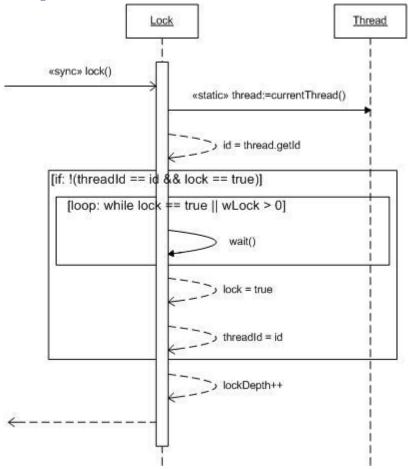


Figure 40: Sequence diagram over lock():void

5.6.2 <sync> public unlock:void

This method is used to remove the block set by the lock method.

5.6.2.1 Parameters

no parameters

5.6.2.2 Preconditions

no preconditions

5.6.2.3 Postconditions

- If the id of the calling thread is equal to threadId and lock is set to true.
 - o lockDepth has been decreased by 1.
 - If lockDepth is 0 (after being decreased)
 - lock has been set to false
 - all waiting threads has been notified that the lock is now free to use.

5.6.2.4 Sequence diagram

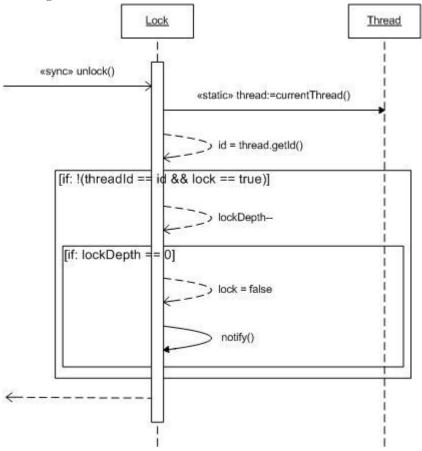


Figure 41: Sequence diagram over unlock():void

5.6.3 <sync> public writeLock:void

This method is used to make calls to lock block until an equal amount of calls to writeUnlock has been made. If the lock method has been called this method will block.

5.6.3.1 Parameters

no parameters

5.6.3.2 Preconditions

• The thread that is calling this method has not previously called lock without calling unlock.

5.6.3.3 Postconditions

wLock has been increased by 1.

5.6.3.4 Sequence diagram

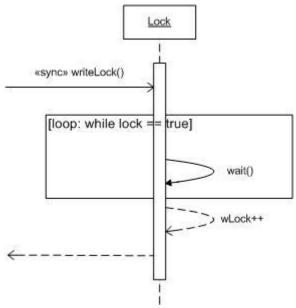


Figure 42: Sequence diagram over writeLock():void

5.6.4 <sync> public writeUnlock:void

This method is used to lower the number of blocks set by the writeLock method.

5.6.4.1 Parameters

no parameters

5.6.4.2 Preconditions

• The calling thread has previously called the writeLock method.

5.6.4.3 Postconditions

• wLock has been decreased by 1.

5.6.4.4 Sequence diagram

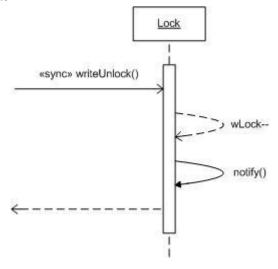


Figure 43: Sequence diagram over writeUnlock():void

6 Message layout

Messages between server and client on security level consist of four parts, where three has fixed length and one has arbitrary length. The different parts are:

- 1. Message number, length is 4 bytes, the same as a 32-bit int in byte format.
- 2. Message code, length is 3 bytes, message codes are encoded in UTF-8 format, possible message codes are MSG, ACK.
- 3. Message (from higher level), Arbitrary length UTF-8 encoded byte string, only messages with the code MSG has length larger than zero.
- 4. Message digest, length is 32 bytes since SHA-256 (see section 8.3) is going to be used.

Message number serves to avoid replaying of messages and the Mac is used to preserve integrity of message. A message of type ACK only has 3 parts because it does not carry any message. The total message overhead is 39 bytes.

7 Data backup layout

For the storage module to recognize a backup and what keys that was used to encrypt it must have a header that tells this to the module. The backup string consists of four parts, those are:

- 1. Encryption key identifier, the identifier of an encryption key.
- 2. MAC key identifier, the identifier of a MAC key.
- 3. Backup date, a long that represents the date when the backup was made
- 4. Backup data, encoded as a hexadecimal string

The different parts should be separated with a linebreak ('\n'). Except that for use for identifying the backup the header (part 1-3) can be used by other parts of an application to show info about the backup to the user.

8 Security design

This section contains descriptions and pseudo-code for the security critical parts of the application. It should be noted that the security of this system depends on the security of the environment that it is running in since the security of the system itself is focused on direct attacks against itself. An example is that the environment should be clean of malware to prevent things like key logging and possible key compromise. The system itself protects the confidentiality and integrity of data, both in storage and transfer, and makes it impossible for an adversary to confirm a guess of the password from data that is transferred between client and server. Also the only way to find passwords from the user file through brute force search of all possible passwords for each user since a random value (salt) is added to the password of each user before it is hashed.

Though there is a weakness in the network module that makes it possible for an adversary to perform a fairly easy Denial of Service attack (DoS). By sending random data to either a client or a certain socket on the server it is possible to effectively make it very unlikely that any legitimate data gets through. Since this would not require heavy amounts of data, only a steady stream, it would be possible for an adversary to send to a lot of sockets on the server to hinder every client to use the system. This issue will however not jeopardize the security of the data used in the system, only the availability of it.

8.1 Authentication scheme

The authentication scheme that is recommended is the APKAS-AMP authentication scheme from the IEEE standard 1363.2-2008 implemented in the discrete logarithm setting which means that its security is based upon the hardness to solve discrete logarithms (to solve $A = a^x$ for x given A and a) for large numbers. The scheme is presented with pseudo-code below.

The pseudo-code is presented in way that should make it possible to implement without any prior knowledge of finite field arithmetic's but it is good for some understanding.

8.1.1 Pseudo-code of APKAS-AMP

All authentication schemes presented in IEEE std 1363.2 uses a similar set of domain parameters. These consists of q that defines a finite field GF(q). Its value is either a prime p, p where p is an integer or p where p is prime and p is an integer. A prime p that divides p-1 and p which is an integer between 1 and p-1 and has an order of p. That p has order of p means that p mod p = 1. Except for this it is also important for the security of APKAS-AMP that p-1 is a prime. This is also specified below.

Domain parameters:

- q Specifies a finite field GF(q) where q is an odd prime p of length at least 1024 bits.
- r A large prime that is designating the order of the desired group and is dividing q-1. The length of r is 160 bits.
- k The cofactor of q and r where q-1 = r*k. Not used directly in this scheme but it is important that the property that k/2 is prime is satisfied because it can cause some security issues otherwise and also it will cause some checks to fail sometimes even if the user uses the correct password.
- q A group element of order r that generates the desired group.

Other parameters:

userId – The id of the user.

- pwd The password of the user
- salt An optional byte array of length 16 that is combined with the password when creating the password verifier.
- u The password and salt combined into an integer.
- v A password verifier, should be created and stored on the server for each user.
- z A shared secret value. If everything goes right, server and client will have the same z at the end.

Hash function:

A hash function is used by a few of the functions below, this hash function should be SHA-256. It is denoted: hash(byte[]):byte[].

Authentication procedure outline:

This describes the procedure from both server and client point of view.

Server:

- 1. Receive userId from client.
- 2. Load user data, v and salt, from user database.
- 3. Send domain parameters and salt to client.
- 4. Receive public key from client, w_c.
- 5. Generate random secret key s from [1, r-1].
- 6. Check that w_c is valid by checking that it is in range [1,q-1].
- 7. Calculate server's public key, w_s. (createPublicKeyServer)
- 8. Send w_s to client.
- 9. Receive key confirmation messag, c_c, from client.
- 10. Calculate shared secret z. (calculateSharedSecretServer)
- 11. Check that z is valid by checking that it is in range [1, q-1] and $z^r \mod q = 1$.
- 12. Convert z into byte array keyMaterial.
- 13. Check that c_c is valid valid by checking that c_c = createConfMsg, where P = 0x04.
- 14. Create key confirmation message c_s . (createConfMsg, where P = 0x03)
- 15. Send c_s to client.

Client:

- 1. send userId to server.
- 2. Receive domain parameters and salt from client.
- 3. Check that domain parameters are valid, according to the specification above.
- 4. Generate random secret key s from [1, r-1].
- 5. Calculate client's public key, w_c. (createPublicKeyClient)
- 6. Send w_c to server.
- 7. Receive public key, w_s, from server.
- 8. Check that w_s is valid by checking that it is in range [2, q-2]
- 9. Convert password and salt to an integer value u. (convertPassword)
- 10. Calculate shared secret z. (calculateSharedSecretClient)
- 11. Convert z into byte array keyMaterial.
- 12. Create key confirmation message c_c . (createConfMsg, where P = 0x04)
- 13. Send c_c to server.
- 14. Receive key confirmation message, c_s, from server.
- 15. Check that c_s is valid by checking that c_s = createConfMsg, where P = 0x03.

CreatePublicKeyServer function:

This function creates a public key for the server. Server step 7.

Input:

A set of valid domain parameters must be available.

 w_c – The public key of the client.

v – The password verifier stored on the server.

userId – The id of the user.

Output:

returns a public key for the server as an int.

Process:

```
function int createPublicKeyServer(int w<sub>c</sub>, byte[] userId, int v) {
    byte[] o = int2ByteArray(w<sub>c</sub>)
    o = hash(o||userId)
    i = byteArray2Int(o) // absolute value
    return ((w<sub>c</sub><sup>i</sup>)*v)<sup>s</sup> mod q
}
```

CalculateSharedSecretServer:

This function calculates the shared secret on the server side. Server step 10.

Input:

A set of valid domain parameters must be available.

 w_c – The public key of the client.

s – The secret key of the server.

Output:

returns the shared secret as an integer.

Process.

```
function int calculateSharedSecretServer(int w_c, int s) { return (w_c * g)^s }
```

CreatePublicKeyClient function:

This function creates a public key for the client. Client step 5.

Input:

A set of valid domain parameters must be available.

s – The random private key of the client.

Output:

returns a client public key, w_c, as an integer.

Process:

```
function int createPublicKeyClient(int s) { w_c = g^s \mod q }
```

ConvertPassword function:

Converts password and salt to an integer value of the desired group generated by g. Client step 9.

Input:

A set of valid domain parameters must be available.

pwd – Password in the form of a byte array.

salt – A random value that stored with the user info on the server.

Output:

returns an integer value.

Process:

```
function int convertPassword(byte [] pwd, byte [] salt) {
   byte [] o = hash(pwd||salt)
   int u = byteArray2int(o) mod r
   return u
}
```

CalculateSharedSecretClient function:

Calculates the shared secret on the client side. Client step 10.

Input:

A set of valid domain parameters must be available.

 w_c – The public key of the client.

 w_s – The public key of the server.

s – The private key of the client.

u – converted password value.

userId – The id of the user as a byte array.

Output:

returns the shared secret in the form of an int.

Process:

```
function int calculateSharedSecretClient(int w_c, int s, byte [] userId) { byte[] o = int2ByteArray(w_c) o = hash(o||userId) int i = byteArray2Int(o) // absolute value int z = (s + 1)((s * i) + u)^{-1} mod r // x^{-1} is the multiplicative inverse // of x (in this case mod r) // it is used to calculate division // in a finite field z = w_s^2 return z }
```

CreateConfMsg function:

Creates a key confirmation message. Step 13 and 14 server. Step 12 and 15 client.

Input:

P – A byte that is used differentiate messages, e.g. server or client confirmation messages.

 w_c – The public key of the client.

w_s – The public key of the server.

Z – The key material.

Output:

returns a confirmation message as a byte array.

Process:

```
function byte[] createConfMsg(byte P, int w_c, int w_s, byte[] Z){
 o_c = int2ByteArray(w_c) \\ o_s = int2ByteArray(w_s) \\ return hash(P||o_c||o_s||Z) }
```

DeriveKey function:

This function is used to derive keys from the shared secret. P should be different for each key.

Input:

keyMaterial – A byte array containing the shared secret.

keyLength – The length of the key that will be returned, in bytes.

P – A key derivation parameter of one byte. This should be different for each key that is derived from the same shared secret.

Process:

```
function byte[] deriveKey(byte[] keyMaterial, int keyLength, byte P) {
   int threshold = (keyLength / output length of hash) + 1
   byte[] key = empty array
   int counter = 1
   for 1 to threshold {
       key = key||hash(P||counter||keyMaterial)
       counter++
   }
}
```

GenerateDomainParams function:

This function is used to generate valid domain parameters for APKAS-AMP. That includes q, r and g.

Input:

_

Output:

returns a set of domain parameters containing q, r and g.

Process:

```
}
}
do {
    int h = a random int of the same bit length as q
    if h >= q {
        continue
    }
    g = h<sup>k</sup> mod q
} while g == 1 || !(g<sup>r</sup> mod q == 1)
    return q, r, g as int array
}
```

createUserData function:

This function is used to create the user data that should be stored on the server.

Input:

pwd – The requested password as a byte array.

Output:

returns salt and v as a byte array.

Process:

```
function byte[] createUserData(byte[] pwd) {
    byte[] salt = random byte array of length 16
    byte[] o = hash(pwd||salt)
    int u = byteArray2Int(o) mod r
    int v = g<sup>u</sup> mod q
return salt||int2ByteArray(v)
```

8.1.2 Implementation notes

- It should not be possible for a user to do unlimited numbers of authentication attempts since it makes it possible for an adversary to launch an online guessing attack. To avoid this, a user should only be allowed to do x number of failed attempts before the account is locked where x should be a relative small number, e.g. 5.
- In this design of the authentication procedure it is intended that registration of new users and unlocking of locked accounts is handled by an authenticated and authorized administrator. It is possible to let the user handle registration by itself but then some other security system has to handle that since this design only protects authenticated users. The reason that an administrator should reactivate an account that has been locked is that a reactivation mechanism that the user can use is potentially weaker than the authentication which makes the system weaker overall.

8.2 Symmetric key encryption

Symmetric key encryption will be used for both secure storage and secure transfer because it offers better efficiency compared to public key encryption from which techniques are used by the authentication scheme. Symmetric key encryption uses two or three different components, a block cipher, a mode of operation and a padding scheme if that is necessary with the used mode of operation. The cipher is used to encrypt and decrypt blocks of data while the mode of operation decides how these blocks will be tied together. The mode of operation is important because if a message is just encrypted block by block and put together with each block in the same order as in the original message it may be possible to put together information about a message since identical

blocks will also be identical as cipher text. Most modes of operation remove this weakness by letting the cipher text of a block also depend on something from the previous block or some other changing parameter. The padding scheme is used to fill up the last block of the message to make it possible to encrypt with the cipher if the mode of operation does not support variable length on the last block.

The choices for these three components are the AES block cipher, CBC mode and PKCS5 padding. The reason for these are that AES is a very strong encryption standard and will be that for a long time, CBC is a quite simple mode of operation while still doing its job and PKCS5 padding because it is one of the padding schemes supported by Java. PKCS5 padding is not an actual padding standard but the padding used the RSA laboratories PKCS #5 standard for Password-based Encryption.

These three components should be implemented in ActionScript using the design presented in section 3 of this document and using the pseudo-code/descriptions in the three subsections below. In Java a Cipher object that will perform the same encryption/decryption can be obtained by using the statement: Cipher.getInstance("AES/CBC/PKCS5Padding").

8.2.1 Advanced Encryption Standard (AES)

The AES block cipher was developed as a competition where scientists could send in their proposed algorithms and the winner of this competition was the Rijndael cipher. Rijndael can be used with blocks and keys of lengths 128, 192 and 256 bits but only blocks of 128 bits are allowed in AES. The cipher will be described below with pseudo-code taken from the document "A Specification for Rijndael, the AES Algorithm" by Dr. Brian Gladman and some information is also taken from the NIST publication FIPS 180-97 which is the standard document describing AES.

The state of AES:

The internal state of AES consists of a two dimensional array of bytes with four columns and four rows which contains the bytes of one block. This state is first created by inserting the first four bytes in the first column top to bottom, and then next four bytes in the second column and so on. When cipher has been run the output block is constructed in the same way. A byte in the state-array is denoted state[r,c] where c is the column number and r is the row number.

Attributes:

- key The key that is used to encrypt data with this cipher. Can be of length 128 (16), 192 (24) or 256 (32) bits (bytes).
- nr The number of rounds that the cipher should use, has value 10, 12 or 14 depending on the length of the key where the longer the key is the more rounds is used.

Constants:

- SBox A byte array of length 256 that is used for substitution of bytes in the SubBytes function. It can be generated with the sboxGen function.
- InvSbox The inverse of SBox that is used by the function InvSubBytes. The values of this is invSBox[SBox[i]] = i, $0 \le i \le 256$.
- FFlog A byte array of length 256 that is used when multiplying bytes. It can be generated by calculating $x = 3^N$ for all 0 < N < 256 and inserting N at position x. However normal multiplication does not work, instead the function mult described below must be used.
- FFpow A byte array of length 510 that is used when multiplying bytes. It can be generated by calculating $E = 3^x$ for all $0 \le x \le 255$ and inserting E at position x. Position 255 509 should be the same as 0 254. Also here the function mult is used for multiplication.

Notes:

- All byte variables below are considered unsigned bytes.

Encrypt function:

The function that is used to encrypt one block of data.

Input:

in – The block of 16 bytes that is going to be encrypted.

keySchedule – An array containing all roundkeys derived from *key*. The length of it is 16 * number of rounds.

Output:

returns a byte array of length 16 containing the encryption of in.

Process:

Decrypt function:

The function that is used to decrypt one block of data.

Input:

in – The block of 16 bytes that is going to be decrypted.

keySchedule – An array containing all roundkeys derived from *key*. The length of it is 16 * number of rounds.

Output:

returns a byte array of length 16 containing the decryption of in.

Process:

```
return state // should be transformed to 16 byte array }
```

Key expansion function:

This function is used to create the round keys from the key. For efficiency it should be used when the key is set and saved for future use.

Input:

key – The key in the form of a byte array of length 16, 24 or 32 nk – The length of *key* divided by 4.

Output:

keyS

Process:

```
function KeyExpansion(byte[4 * nk] key, nk) {
      i = 0
      byte[16 * (nr + 1)] keySchedule
      while i < (nk * 4) {
            keySchedule[i] = key[i]
            i = i + 1
      }
      i = nk
      while i < nk * (nr + 1) {
            byte[4] temp
            for j=0 to 3 {
                  temp[j] = keySchedule[i*4 - 4 + j]
            if i \mod Nk = 0 {
                  temp = SubWord(RotWord(temp)) XOR Rcon(i / nk)
            else if (Nk = 8) and (i \mod nk = 4) {
                  temp = SubWord(temp)
            for j=0 to 3 {
                  keySchedule[i + j] = keySchedule[i - nk + j] XOR temp[j]
            i = i + 1
      }
}
```

Helper functions:

```
state[r,c] = t[c]
      }
function void MixColumns(byte[4,4] state) {
      byte t[4]
      for c = 0 to 3 {
            for r = 0 to 3 {
                  t[r] = state[r,c]
            for r = 0 to 3 {
                  state[r,c] = FFmul(0x02, t[r]) XOR
                        FFmul(0x03, t[(r + 1) mod 4]) XOR
                         t[(r + 2) \mod 4] \times r t[(r + 3) \mod 4]
            }
      }
}
function void XorRoundKey(byte state[4,4], byte[16] rk) {
      for c = 0 to 3 {
            for r = 0 to 3 {
                  state[r,c] = state[r,c] xor rk[c * 4 + r]
            }
      }
}
function void InvShiftRows(byte[4,4] state) {
      byte t[4]
      for r = 1 to 3 {
            for c = 0 to 3 {
                  t[(c + r) \mod 4] = state[r,c]
            }
            for c = 0 to 3 {
                  state[r,c] = t[c]
function void InvSubBytes(byte[4,4] state) {
      for r = 0 to 3 {
            for c = 0 to Nb - 1 {
                  state[r,c] = InvSbox[state[r,c]]
      }
}
function void InvMixColumns(byte[4,4] block) {
      byte t[4]
      for c = 0 to 3 {
            for r = 0 to 3 {
                  t[r] = block[r,c]
            for r = 0 to 3 {
                  block[r,c] = FFmul(0x0e, t[r]) XOR
                        FFmul(0x0b, t[(r + 1) mod 4]) XOR
                        FFmul(0x0d, t[(r + 2) mod 4]) XOR
                        FFmul(0x09, t[(r + 3) mod 4])
            }
      }
```

```
}
function byte FFmul(const byte a, const byte b) {
      if (a != 0) and (b != 0) {
           return FFpow[FFlog[a] + FFlog[b]]
      else {
            return 0
function byte[16] extractRK(byte[] ks, int round) { // RK means Round Key
      byte[16] rk
      for i = 0 to 15 {
           rk[i] = ks[16 * round + i]
      return rk
}
function byte[4] SubWord(byte[4] t)
      for i=0 to 3 {
            t[i] = SBox[t[i]]
}
function byte[4] RotWord(byte[4] t) {
      byte temp =t[0]
      for i=1 to 3
           t[i -1] = t[i]
      t[3] = temp
}
function byte[4] Rcon (int i) {
      byte[4] res = [2^{i-1} \mod 256, 0, 0, 0]
}
function int mult(int f1, int f2) {
      int res = 0, test = 0x01, overflow = 0x100
      if (f2 & test) == test {
            res = f1
      for k = 1 to 8 {
            f1 = f1 << 1
            test = test << 1
            if (f1 & overflow) == overflow {
                  f1 = f1 ^ 0x11B
            if (f2 & test) == test {
                  res = res ^ f1
      }
      return res
}
function byte[256] sboxGen() {
      byte[256] sbox
      for i = 0 to 256 {
            x = i
            if x != 0 {
                  x = res1[x] & 0xff
```

Test:

Test values to confirm that the cipher is working correctly. Values are in hexadecimal format.

AES with 128 bit key

```
      Input:
      32
      43
      f6
      a8
      88
      5a
      30
      8d
      31
      31
      98
      a2
      e0
      37
      07
      34

      Key:
      2b
      7e
      15
      16
      28
      ae
      d2
      a6
      ab
      f7
      15
      88
      09
      cf
      4f
      3c

      Output:
      39
      25
      84
      1d
      02
      dc
      09
      fb
      dc
      11
      85
      97
      19
      6a
      0b
      32

      AES with 192 bit key
      Input:
      00
      11
      22
      33
      44
      55
      66
      77
      88
      99
      aa
      bb
      cc
      dd
      ee
      ff

      Key:
      00
      01
      02
      03
      04
      05
      06
      07
      08
      09
      0a
      0b
      0c
      0d
      0e
      0f

      Input:
      00
      01
      02
      03
      04
      05
      06
      07
      08
      09
      0a
      0b
      0c
      0d
      0e
      0f
```

AES with 256 bit key

```
Input: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

Key: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

Output: 8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89
```

8.2.2 Cipher Block Chaining Mode (CBC)

CBC is a mode that combines the plaintext block with the previous ciphertext block by computing the exclusive-or of them. This block is then passed to the cipher to create the ciphertext for this plaintext block, see figure x. Since there is no ciphertext to combine with the first plaintext an initialization vector is used here. The initialization vector does not need to be secret but it must be changed for every new message and be unpredictable, that is it should be randomly generated.

Decryption is done in the same way but in opposite order, the ciphertext is first decrypted by the cipher and then exclusive-or'ed with the previous ciphertext or the initialization vector if it is the first ciphertext block that is being decrypted, see figure x.

The information about CBC was fetched from NIST Special Publication 800-38a and more information about it and other modes of operation can be found there.

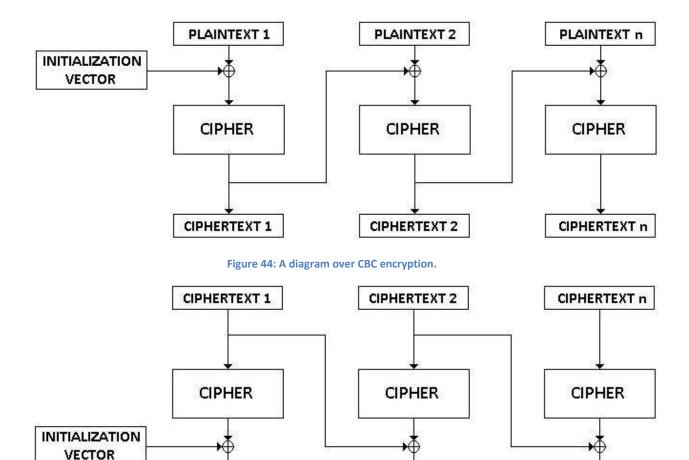


Figure 45: A diagram over CBC decryption.

PLAINTEXT 2

PLAINTEXT n

8.2.3 The PKCS5Padding scheme

PLAINTEXT 1

A padding scheme is in general very simple and that is also the case with the padding scheme from PKCS5. What is done is that the last block is filled up with bytes having value that is equal to the number of bytes that was added. For exemple if the last block contains 9 bytes it is filled up with 7 bytes each having the value 7. If the last block is full, a whole block of padding is added where each byte have the value of 16.

This padding scheme is described in RFC2898 that is a republication of the standard PKCS #5 from RSA Laboratories. More information about the scheme can be found in those documents.

8.3 Cryptographic hash function

A cryptographic hash function is used to generate a hash-value (or message digest) from a message of arbitrary length. There are two important properties that cryptographic hash functions must satisfy, those are:

- One-way function, which means that it is infeasible to acquire the original message from a hash-value computed by the function.
- Collision resistant, this means that there is no other way than a brute force search to find two different messages that produces the same hash-value.

A cryptographic hash function has several different uses; some of these are to check integrity of a message, to mask some secret data when proving knowledge of it and to increase performance with digital signatures by signing the message digest instead of the message itself. This application uses the two first in the network module, the first for secure transfer and the second for storing of user data and key confirmation.

The hash function that should be used in this application is SHA-256 specified in FIPS 180-2 that is issued by NIST. It should be implemented in ActionScript using the design specified in section 3 of this document and the pseudo-code in the subsection below. In Java an object that performs SHA-256 hash computation can be obtain by executing the statement: MessageDigest.getInstance("SHA-256").

8.3.1 Implementation description and pseudo code of SHA-256

This algorithm works on 32-bit unsigned integers instead of bytes but since unsigned integers can be read and written from the ActionScript class ByteArray this should not pose any problems. The reason for using integers instead of bytes is that some operations are done modulo 2^{32} which makes using integers very fast. All the bit strings in this section are written in hexadecimal.

This is a compiled description of the SHA-256 standard provided by NIST FIPS 180-2 that is a standard document for the different versions of the Secure Hash Algorithm (SHA). For more information see that document.

Constants:

64 32-bit integers, stored in the array K from left to right.

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5 d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174 e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da 983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967 27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85 a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070 19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3 748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

8 32-bit integers that is initial hash value, stored in the array H.

```
6a09e667
bb67ae85
3c6ef372
a54ff53a
510e527f
9b05688c
1f83d9ab
5be0cd19
```

The message:

The message length must be a multiple of 512 bits (64 bytes), which means that it must be padded if its not. Set l as the length of the message M. Then the padding shall be as follows: Append a bit 1 at the end of the message followed by k number of zero bits, where k is the smallest non-negative solution to l+1+k := 448 mod 512. The last 64 bits tells the length of the message.

The message is then divided into N 512 bit blocks where each block consists of 16 32-bit integers.

ComputeHash function:

Input:

H - The initial hash value of 8 32-bit unsigned integers, see above.

K - 64 32-bit integer constants see above.

M - The padded message of 16*N unsigned integers.

Output:

The final value of H.

```
process:
```

```
function int[8] computeHash(uint[8] H, uint[64] K, uint[16,N] M) {
for i=0 to N-1 {
      // setup message schedule
      for t=0 to 15 {
            W[t] = M[t,i]
      for t=16 to 63 {
            W[t] = O1(W[t-2]) + W[t-7] + O0(W[t-15]) + W[t-16]
      //Initialize working variables
      a = H[0]
      b = H[1]
      c = H[2]
      d = H[3]
      e = H[4]
      f = H[5]
      g = H[6]
      h = H[7]
      // 64 rounds
      for t=0 to 63 {
            T1 = h + E1(e) + Ch(e, f, q) + Kt + Wt
            T2 = E0(a) + Maj(a,b,c)
            h = q
            g = f
            f = e
            e = d + T1
            d = c
            c = b
            b = a
            a = T1 + T2
      // compute the hash value of this round.
      H[0] = a + H[0]
      H[1] = b + H[1]
      H[2] = c + H[2]
      H[3] = d + H[3]
      H[4] = e + H[4]
      H[5] = f + H[5]
      H[6] = g + H[6]
      H[7] = h + H[7]
```

```
}
// return the resuluting hash of 256 bits.
return H
}
```

Helper functions:

```
function uint Ch(int x, int y, int z){
      return (x & y) ^ (!x & Z)
}
function uint May(int x, int y, int z) {
      return (x & y) XOR (x & z) XOR (y & z)
function uint E0(int x) {
      return ROTR(2,x) ^{\circ} ROTR(13,x) ^{\circ} ROTR(22,x)
function uint E1(int x) {
      return ROTR(6,x) ^{\circ} ROTR(11,x) ^{\circ} ROTR(25.x)
function uint 00(int x) {
      return ROTR(7,x) ^{\circ} ROTR(18,x) ^{\circ} SHR(3,x)
function uint O1(int x) {
      return ROTR(17,x) ^{\circ} ROTR(19,x) ^{\circ} SHR(10,x)
function uint SHR(int n, int x) {
      return x >> n
}
function uint ROTR(uint n, uint x) {
      return (x >> n) | (x << 32 - n)
}
```

Test:

To test if the implementation a test string is provided.

```
This is a message
```

Should produce the message digest (if the message is encoded to bytes according to UTF-8):

a826c7e389ec9f379cafdc544d7e9a4395ff7bfb58917bbebee51b3db1c996a

8.4 Message Authentication Code (MAC)

Message authentication codes have similar uses as cryptographic hash functions since it also takes a message and produces a short piece of information that can be used to identify the message. The biggest difference between MACs and hash-functions is that a MAC algorithm takes a key in addition to the message. This means that a MAC doesn't tell anything about a message as the hash-function does since it always produces the same message digest from the same message. This has the effect that a MAC does not need to be encrypted to be secure when used as an integrity check.

118

Since MAC is not used in the network module (a message digest is used for the integrity check instead because it fits the design better) there is no need to implement any because Java already has a few MAC algorithms of the type HMAC in its cryptographic library. HMAC are MAC algorithms that uses cryptographic hash functions to compute the MAC. The suggestion for this application is to use the algorithm that uses SHA-256 as its hash function since any algorithm with higher security is not necessary in this case but all algorithms with lower security level are considered broken in some way. An object that performs this MAC computation can be obtained by executing the statement: Mac.getInstance("HmacSHA256").

8.5 Key management

A possible weak link when using encryption to secure the storage of data is how the different keys are managed because if an adversary gets hold of a key the data is not secure anymore. As long as the keys are kept safe, encrypted data will also be secure since the AES cipher that is used for encryption is strong enough to consider it infeasible to recover any data that has been encrypted with it without having access to the key that was used for the encryption.

To reduce the risk for keys to getting compromised no key should be used for more than one purpose, for example a key that is used for encryption should not also be used for computing of MAC's. The reasons for this is that a key that is used for multiple purposes may weaken the security of one or more of the cryptographic processes that it is being used for, and the damage will be less if a compromised key is only used for one purpose.

When used in a server application like in this application keys should be stored in memory at all time since it is a lot harder to find a key in the memory than it is to find it on a storage device like a hard disk. The downside with having the keys in the memory is that they must be backed up so they can be recovered if the system would crash. Such backups should be stored in a way that is as secure as the way that the key is stored in the system, preferably in a safe or similar. The storage module of this design features key backups by creating a string that contains the key and some other information about it but it is up to the user of the module to save this string as a file on some storage device and keep it secure.

Another thing that should be taken into consideration is that keys should be changed at regular intervals since the longer time a key is active the larger is the possibility that it has been compromised. This also limits the amount of data that is encrypted using the same key, in this case the number of backups. To limit this is good because it is harder for an adversary to determine a key with less data available. A key used for data encryption is recommended to have an active period of a most 2 years.

For more information about key management see the NIST Special Publication 800-57.

8.5.1 Key backup format

The format for the key backup string produced by the storage module is as follows:

- Identifier, should be unique. One way is to use a random number that is sufficiently large that the chance of it being repeated is small. Can also be a counter.
- Key type, either encryption key (0) or MAC key (1)
- Activation date, a long that represents the date when this key was activated.

- Expiration date, a long that represents the date when this key expires.
- Key, represented by a hexadecimal string.

Each of these posts is separated by a line break, '\n'.

8.6 Random number generation

The security level of the authentication scheme used in this application depends on how good random numbers that can be generated and it is important that a cryptographically secure random number generator is used. The most important property of this generator is that it is infeasible to find out anything about its state when one or more of its generated values are known. The security when using such a generator depends on how good the entropy that is used to seed it is since the randomness of the seed decides the randomness of the generated bytes.

The entropy input should be gathered from sources that produce random noise, for example different hardware devices. Many operating systems have built-in ways of generating random numbers by gathering random noise from many different sources. These are often the best sources to get a good seed but unfortunately it is not possible to access these through ActionScript which means that the client needs another source for its seed. One option here is to use everything available through the API, such as system time, other timers and a couple of system properties that is somewhat random, but it would be insufficient for good security. Another option is to let the user provide the randomness which has the potential to provide a seed with better security even though the user may have to work for it.

8.6.1 Entropy gathering

The collection of randomness can be done by tracking the user's normal use of input devices like keyboard and mouse. In this situation though, that would be a problem since the entropy will be needed only a moment after starting the client application and the time to collect the random data is not enough. Another way to do it is to force the user to provide the needed seed before starting the authentication procedure.

The proposed method of doing this is to let the user move the mouse pointer back and forth on the screen and gather randomness either from the position of the mouse at certain points in time (for example every 200 ms) or the positions of the turning points. One thing that could make the procedure more random is to split the screen up in chunks of 256 pixels to minimize the risk of only using a small space of the screen. It will make it take a little more time though, but it will still gather one byte of randomness at every gathering point.

An alternative is to use another script language that has support for the built in entropy sources in Windows and Linux, e.g. Python, to retrieve entropy and then save it to a temporary file which can be read by the ActionScript implementation. The positive thing about this method is that it does not require any work from the user but on the other hand it is vulnerable against malware. Though it can be argued that if the client has malware installed the adversary will know the user's password by key logging very soon. Also by using this method old sessions can be in danger since it may be possible to retrieve old data from the hard drive.

As a recommendation no less than 128 bit security should be used which means that at least 16 bytes of entropy must be gathered to use as input for the seed operation. Though the method for

generation of pseudo-random bytes that is presented below also requires a random nonce of 8 bytes which also must be generated, this means that 24 bytes must be generated to provide the recommended 128 bits of security.

The random number generator should be implemented in ActionScript using the design in section 3 of this document and the pseudo-code that is presented in the subsection below. In Java an object that produces secure random numbers can be obtained by executing the statement: SecureRandom.getInstance("SHA1PRNG").

8.6.2 Pseudo code for Hash_DRBG

Hash_DRBG is a cryptographically secure random number generator recommended in NIST Special Publication 800-90. The pseudo code presented here is slightly modified to work on full bytes instead of bits which is the in the NIST Publication but except for that it works in exactly the same way.

Internal state:

V – A value of seedlen bytes that is updated on every call for random bytes C – A constant of seedlen bytes that depends on the seed reseedCounter – A counter that is increased on every call to this DRBG

Constants:

seedLen – The length of the seed which should be 55 bytes.

outLen – The output length of the cryptograpfhic hash function used in this algorithm. In this case 32.

reseedInterval – The number of times generate bytes can be called before there is a need for a reseed. The maximum value of this is 2^{48} .

maximalRequestedNumberOfBytes – The maximum number of bytes that it is possible to request in one generate bytes operation. The maximum value of this is 2^{16} .

External functions:

hash(byte [] bytes) – A cryptographic hash function that should be SHA-256.

Instantiate/Constructor function:

This function is used to seed the random number generator with some entropy and must be called before it is possible to generate any random bytes.

Input parameters:

entropyInput – A byte array containing at least 24 random bytes. Note that this input also includes a random nonce of 8 bytes so the actual minimal security strength is 128 bits. To reach 256 bits of security a 40 byte input is needed.

Output:

none

Process:

```
function void instantiate(byte[] entropyInput)
    if len(entropyInput) < 24 {
        Error
        return</pre>
```

```
seedMaterial = entropyInput||timestamp
seed = hashDf(seedMaterial, seedLen)
V = seed
C = hashDf(0x00||V, seedLen)
reseedCounter = 1
```

Reseed function:

This function is used to reseed the generator, which is needed whenever *reseedCounter* has reached the *reseedInterval*.

Input parameters:

entropyInput – A byte array containing at least 16 random bytes.

Output:

none

Process:

```
function void reseed(byte[] entropyInput)
    if len(entropyInput) < 16 {
        Error
        return
    }

    seedMaterial = 0x01||V||entropyInput||timestamp
    seed = hashDf(seedMaterial, seedLen)
    V = seed
    C = hashDf(0x00||V, seedLen)
    reseedCounter = 1
}</pre>
```

GenerateBytes function:

This function is used to generate pseudo-random bytes. The randomness of these bytes depends on the entropy of the seed.

Input parameters:

requestedNumberOfBytes - The number of random bytes that will be generated

Output:

A byte array that contains requestedNumberOfBytes pseudo-random bytes.

Process:

```
function byte[] generateBytes(int requestedNumberOfBytes)
   if requesteNumberOfBytes > maximalRequestedNumberOfBytes {
        Error
        return
}

if reseedCounter >= reseedInterval {
        Error
        return
}

genBytes = hashGen(requestedNumberOfBytes, V)

H = hash(0x03||V)

V = (V + C + H +reseedCounter) mod 256<sup>seedLen</sup>
   reseedCounter++
```

```
return genBytes
```

Helper functions:

```
function byte[] hashgen(int requestedNumberOfBytes, byte[] V) {
      data = V
      W = empty byte array
      while m < requestedNumberOfBytes {</pre>
            wi = hash(data)
            W = W \mid \mid wi
            data = (data + 1) mod 256^{\text{seedlen}}
            m = m + outlen
      return leftmost requestedNumberOfBytes bytes of W
}
function byte[] hash df(byte[] inputBytes, int numberOfBytes) {
      temp = empty byte array
      counter = 1
      while m < numberOfBytes {</pre>
            temp = temp||hash(counter||numberOfBytes||inputBytes)
            m = m + outlen
            counter++
      return leftmost numberOfBytes bytes of temp
}
```