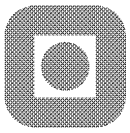
NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY, MATHEMATICS AND ELECTRICAL ENGINEERING



MASTER'S THESIS

Student's name:	Martin Eian - eian@stud.ntnu.no
Area:	Telematics
Title:	**Large Scale Single Sign-on Scheme by Digital Certificates On-the-fly**

Description:

This assignment is based on the project results of Martin Eian: *Public key infrastructure in large scale access control*. The task is to develop an identity management system by refining the proposal specifications for real-time issuing of digital certificates of public keys with short validity time at the system sign-on procedure. This solution is proposed to be practical, low-cost, public-key based access control security that can scale to large organizations with tens of thousands of users. Experimental implementation of selected parts of the scheme should be carried out to substantiate the claims.

Start date:	January 20, 2005
Deadline:	June 16, 2005
Submission date:	June 15, 2005
Department:	Department of Telematics
Supervisor:	Stig Frode Mjølsnes

Trondheim, June 15, 2005

Stig Frode Mjølsnes

Professor

Abstract

This thesis describes a low-cost, scalable, PKI-based system that can be used for single sign-on to networked resources. A functional implementation is presented, as well as performance tests that show that the system is able to scale to tens of thousands of users.

Contents

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Background . . . 10
- 1.3 Problem description . . . 11
- 1.4 Methods . . . 12
- 1.5 Report structure . . . 13

2 Password security **16**
- 2.1 Server attacks . . . 16
- 2.2 Client attacks . . . 18
- 2.3 Network attacks . . . 19
- 2.4 Password security summary . . . 19

3 Single sign-on **21**
- 3.1 Kerberos . . . 21
- 3.2 Extensions to Kerberos . . . 24
- 3.3 SPX . . . 25
- 3.4 SESAME . . . 27
- 3.5 Novell NetWare 4 . . . 27
- 3.6 PKI . . . 29
- 3.7 Single sign-on summary . . . 30

4 Building blocks **31**
- 4.1 PKAS . . . 31
- 4.2 SPEKE . . . 32
- 4.3 SACRED . . . 34
- 4.4 CMP . . . 35

5 Construction **39**
- 5.1 Requirements . . . 39
- 5.2 Design . . . 40

5.3 Security analysis 47
5.4 Performance analysis 51
5.5 Implementation 53
5.6 Performance testing 56

6 Conclusions **64**
6.1 Key learning points 64
6.2 Further work 65
6.3 Summary 66

A Tomcat configuration **74**

B Debug output from server and client **76**

C OpenSSL asn1parse of a PKIMessage **81**

D OpenVPN test **85**

E DSA certificate generation class **89**

F RSA certificate generation class **95**

G SPEKE modulus generation class **100**

H AS/CA implementation **102**

I Client implementation **118**

J AS/CA configuration file **141**

K Client configuration file **143**

L Stress test shell script **146**

M Java stress test classes **147**

N Key generation class **155**

O DSA parameter generation class **158**

P Hardware info **160**

List of Figures

1.1 Identity management system overview 11

3.1 Simplified Kerberos V 5 authentication 22
3.2 Kerberos V 5 authentication 23
3.3 KX.509 authentication 25
3.4 Download of encrypted private key from LEAF 26
3.5 Authentication in Novell NetWare 4 28

4.1 SPEKE 33
4.2 SPEKE authentication 33
4.3 Private key download using SPEKE 35

5.1 Issuing of digital certificate 42
5.2 Issuing of digital certificate and CA root certificate 45
5.3 Active attack on authentication 49

List of Tables

1.1 Notation 15

4.1 PKIMessage and excerpts from PKIBody from RFC 2510 . . . 36
4.2 PKIHeader from RFC 2510 37
4.3 POPSigningKeyInput from RFC 2510 and RFC 2511 38

5.1 CertRequest and CertTemplate from RFC 2511 42
5.2 ProofOfPossession, POPOPrivKey and SubsequentMessage . . 43
5.3 EncryptedValue from RFC 2511 44
5.4 CertRepMessage with contents from RFC 2511 46
5.5 Theoretical performance comparison 52
5.6 Number of performance critical operations 53
5.7 State problems with the Cipher object 55
5.8 Test cases 58
5.9 Initial authentication, issued certificates per second 59
5.10 Renewed certificates per second 60
5.11 Key generation times in milliseconds 61
5.12 DSA parameter generation times in milliseconds 62

Glossary

AES	Advanced Encryption Standard: A symmetric block cipher with a minimum key length of 128 bits
APKAS	Augmented PKAS: PKAS where an attacker must perform a dictionary attack on stolen server data to be able to impersonate a user
AS	Authentication Server: The server responsible for initial authentication of users in a single sign-on system
Authentication	Verifying a claimed identity
Authorization	Granting access based on identity
BPKAS	Balanced PKAS: PKAS where an attacker can directly impersonate a user using stolen server data
CA	Certification Authority: Trusted third party in a PKI that certifies digital certificates by authenticating the entity requesting a certificate, verifying the link between the public key and identity in the certificate, and then digitally signing it
CBC	Cipher Block Chaining: A mode for encrypting blocks of data using a symmetric cipher, where each block of encrypted data is used as input to the next block
CMP	Certificate Management Protocol
CRL	Certificate Revocation List
CRMF	Certification Request Message Format
DH	Diffie-Hellman

Digital certificate	A data structure that contains a public key along with related information, such as the identity of the key owner and expiry date
DL	Discrete Logarithm
DoS	Denial-of-Service: Removing the availability of information resources
DSA	Digital Signature Algorithm: Public-key algorithm that can be used for digital signatures
EKE	Exponential Key Exchange: The first published BPKAS, uses a password-derived key to encrypt the public key in a Diffie-Hellman key exchange
Identification	Presenting a claimed identity
Identity	Unique name of a person or device
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IV	Initialization Vector: The data used as input to the first block when using a symmetric cipher in CBC mode
KDC	Key Distribution Center: The key server in Kerberos
LDAP	Lightweight Directory Access Protocol
LEAF	Login Enrollment Agent Facility: Component in SPX[32] that stores encrypted user private keys on-line
MD5	Message Digest 5: A hash function with an output length of 128 bits
MIT	Massachusetts Institute of Technology
MITM	Man in the middle: An attacker positioned between two legitimate users, relaying and possibly modifying their conversation

NDS	Novell Directory Server: Server that stores user credentials in Novell NetWare 4
OCSP	Online Certificate Status Protocol
PAS	Privilege Attribute Service: The authorization service in SESAME[52]
PDM	Password Derived Modulus: A BPKAS, uses a password-derived value as the modulus in a Diffie-Hellman key exchange
PIN	Personal Identification Number: A number, typically four to ten digits, used instead of a password
PKAS	Password-authenticated Key Agreement Scheme: Key agreement where two parties generate a strong shared secret key based on a weak password
PKI	Public Key Infrastructure: Infrastructure that provides certification, distribution and revocation of digital certificates
PoP	Proof of Possession: Protocol field in CMP that contains proof that the client possesses the private key corresponding to the public key in a Certification Request
RFC	Request For Comments: Internet "standard" published by the IETF
RSA	Rivest Shamir Adleman: Public-key algorithm that can be used for digital signatures and encryption
SACRED	Securely Available Credentials: A protocol framework for secure exchange of credentials
SESAME	A Secure European System for Applications in a Multi-vendor Environment
SHA-1	Secure Hash Algorithm 1: A hash function with an output length of 160 bits

SPEKE	Simple Password Exponential Key Exchange: A BPKAS, uses a password-derived value as the generator of the group in a Diffie-Hellman key exchange
Strong authentication	Authentication by proving knowledge of a secret without revealing the secret itself[32]
TGS	Ticket Granting Server: One of two components of the Kerberos KDC, accepts TGTs and hands out service tickets
TGT	Ticket Granting Ticket: Initial ticket in Kerberos that can be used to obtain service tickets
TLS	Transport Layer Security

Chapter 1

Introduction

1.1 Motivation

According to [11], information security should provide integrity, confidentiality and availability of information. To achieve this goal, access to information must be restricted. Only authorized individuals should be able to modify the information (integrity) or read the information (confidentiality). No unauthorized individual should be able to prevent such access by an authorized individual (availability). Authorization is usually implemented by granting access privileges, and access privileges are associated with a role or an identity. The identity could be a user name for a computer system, a social security number for a digital certificate, or an e-mail address for an e-mail system. To obtain the privileges, you need to prove your identity. The act of proving your identity is referred to as *authentication*.

There are several ways to authenticate an individual. [11] lists five factors that can be used for authentication purposes: Something you know, something you have, who you are, what you do and where you are. The two most commonly used factors are something you know, such as a password, and something you have, such as a smart card or other token.

Strong authentication schemes based on public-key cryptography are widely deployed throughout the world today. Examples of such schemes are the challenge-response authentication exchanges in Transport Layer Security (TLS)[2] and Internet Key Exchange (IKE)[13]. One problem with these schemes is knowing which public key belongs to which identity, because the only thing they prove is that the individual you are communicating with possesses the private key that corresponds to the public key used for authentication. This problem is often solved through the use of digital certificates, such as X.509[15] digital certificates. The certificate contains a public key along with infor-

mation that binds the key to an identity. The certificate is then digitally signed by a trusted third party, often called a Certification Authority (CA). The CA thus performs the authentication, and its signature states that the individual in possession of the private key that corresponds to the public key in the certificate really has the identity listed in the certificate. To make this practical on a large scale, a Public Key Infrastructure (PKI) is required to certify, distribute and revoke digital certificates. Many organizations do not have such an infrastructure, possibly due to the fact that a PKI can be expensive, complex and difficult to maintain.

This thesis describes a practical, low cost and scalable PKI solution that can be used for authentication. It is a refinement of the proposed solution draft from [9].

1.2 Background

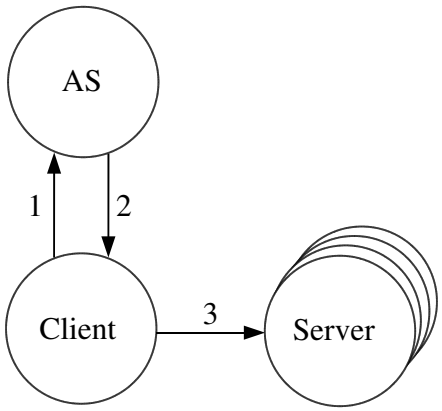
In any large scale computer network environment, users access several different services, often on different servers. When password-only authentication is used, users must enter a password every time they wish to access a service or server. This quickly becomes cumbersome as the number of different services increases, and the need to store passwords on all the servers means that an attacker can get access to password files by compromising one of a large number of servers. Thus the concept of single sign-on was born. Single sign-on basically means that the users authenticate only once, usually by entering their user name and password, to a central service responsible for authentication to network services. Password files are only stored on the servers providing this authentication service. When authenticated, users obtain some sort of digital credentials that can be used to prove their identity to the other services. The net result is that the user only needs one password, and that this password is used only once per session.

It may be argued that single sign-on could in some cases actually reduce the security of a system, because a stolen password in a system using single sign-on leads to a total system compromise for the user in question, as opposed to a partial compromise of an ordinary password authenticated system with a different password for each service. Furthermore, if users in a single sign-on system leave their client while logged on, any malicious individual in the vicinity that has physical access to the client will have full access to all services that the legitimate user has access to, without the need to enter any passwords. These are important issues, but this thesis will not discuss the general security principles of single sign-on versus multiple sign-on any further, because the task at hand is to design and experimentally implement a

single sign-on solution. Whether or not it is wise to actually use this solution is another matter entirely, and should be evaluated separately.

1.3 Problem description

An identity management solution should provide both authentication and authorization. This will be achieved by designing a centralized authentication service with a distributed authorization mechanism. The authentication service provides users with digital credentials, or proof of their identity, while the other servers in the network decide on authorization locally. Figure 1.1 shows how the system should work. First, the client obtains its credentials by authenticating to a central Authentication Server (AS), then it uses those credentials to authenticate to any number of additional servers, with authorization being handled by each server. This thesis will focus on the authentication component of the identity management system. Authorization on the different servers could be performed by existing operating system mechanisms, based on the identity of the user, and will not be further elaborated.



1: $C \rightarrow AS$: Credential request
2: $AS \rightarrow C$: Credential response
3: $C \rightarrow S$: Public-key based authentication

Figure 1.1: Identity management system overview

The assignment specifies that the solution should be practical and low cost, and that it should scale to tens of thousands of users. This was interpreted as saying that the solution must support password-only authentica-

tion, due to the high cost of deploying hardware tokens for all users. This is not saying that it is restricted to password-only authentication, so support for token-based authentication will also be discussed. The scalability is achieved by making sure that the AS does not need to maintain state. A stateless AS could use readily available load-balancing solutions to scale horizontally, just like a cluster of web servers.

The solution presented in this thesis will not provide a mechanism for distributing server certificates to the servers that the users authenticate to. This distribution has to be handled by existing infrastructure, but most large organizations have a configuration management system for their servers, and such a system could be used to distribute server certificates as well as configuration data. For small organizations server certificates could even be handled manually.

Another matter that is considered out of scope for the assignment is the mechanism for initial registration of users and updating user passwords. Initial registration could be achieved by handing out a one-time password that could be used at a secured kiosk terminal or "voting booth" to register and set the initial user password. Updating a password could be achieved through a web page requiring certificate-based authentication. The user signs on using the single sign-on solution, and then authenticates to the password updating service using their digital certificate. To sum it up, the assumption is that user identities and password verifiers are already stored in the AS.

1.4 Methods

The project work was carried out from January 20 to June 15, 2005 at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway. The project plan was divided into three main activities of 6, 10 and 4 weeks[1] respectively.

The first activity was information gathering. The two primary goals were to locate existing solutions that were similar to the one suggested in [9], and to find building blocks that could be used to construct a new system.

Having gathered information about existing systems, to avoid re-inventing the wheel, and located suitable building blocks, the second activity focused on design, analysis, implementation and testing of the proposed solution. The implementation and testing had a dual focus: First of all the implementation was to be a proof of concept, to prove that the system design was feasible in practice. Additionally, performance tests were to give an indication of the

[1]For the observant readers that have noticed that one week is missing: The missing week was the Easter holidays.

hardware needs for such a system based on the number of users and other system parameters. The security analysis was heavily based on the work of others, because the building blocks used and other suggested authentication schemes have been subject to thorough security analysis by the academic community. Therefore, the security analysis focuses on the authentication scheme itself, how it differs from other schemes, and new vulnerabilities introduced, as well as suggested countermeasures.

The last of the three main activities was finishing the documentation: Gathering all the loose ends, and completing the thesis.

An iterative, test-driven approach was used to implement the selected parts of the system. A minimal system was first implemented, then tested for functionality. More functionality was added in small increments, then tested. When the system was functionally complete, it was stress tested, which revealed several weaknesses with regards to scalability. Improvements were made, and the implementation - testing cycle continued until the system was ready for performance testing. Section 5.5 contains a detailed description of the development cycles.

The motivation for using an iterative approach instead of a water fall type approach was to minimize risk. With the iterative approach, a functional implementation was obtained early, so that if unforeseen difficulties had arisen, one would still have an implementation that was usable as a proof of concept. Furthermore, the development of the experimental system was a learning-by-doing exercise with a lot of uncertain elements, which is a situation that an iterative approach lends itself to. The downside of this approach is that the end result tends to be less structured than when using an approach with a lot of design work preceding the actual implementation. If the goal had been to design and implement a production system, then a different approach would have been more appropriate.

1.5 Report structure

Chapter 2 reviews attacks on password authentication and countermeasures used to avoid such attacks. It also indicates the possible advantages of a password-based single sign-on solution compared to conventional password authentication. Readers familiar with the different kinds of attacks on password protocols and their countermeasures can skip this chapter.

Chapter 3 presents existing and suggested systems that currently are or could be used for single sign-on. Any reader that has detailed knowledge of the system listed in a section title will probably not learn anything new from that section.

Chapter 4 describes the components used for the single sign-on solution. Any readers that are familiar with P1363.2, Certificate Management Protocol (CMP), Securely Available Credentials (SACRED), and password-based cryptography, in particular Password-authenticated Key Agreement Scheme (PKAS), can skip this chapter.

Chapter 5 describes the design, security analysis, performance analysis, implementation and testing of the single sign-on solution. It contains an overview and analysis of the system design, implementation details and test results. This chapter should not be skipped.

Chapter 6 summarizes the project, giving recommendations for implementation of a production system and suggestions for further research. This chapter should not be skipped.

A few words on notation: The informal notation described on page 9 of [6] is used for authentication messages throughout this thesis. Additionally, the conventions listed in table 1.1 are used. In all scenarios presented, a client acting on behalf of a user wants to authenticate to a server that is connected to the network. There will be no distinction made between the user and the client computer acting on behalf of the user in the protocol descriptions, and the terms "client" and "user" will be used interchangeably, unless explicitly stated otherwise.

C	Identity (user name) of client
S	Identity (host name) of server
P	Password
S_P	Random value used to "salt" P
TS	Time stamp
V	Protocol version
$H(M)$	One-way (hash) function applied to the message M
g	Generator for the group used in SPEKE, $g = H(P)^2$
p	Modulus used for modular arithmetic
$HMAC_{K_X}(M)$	HMAC of the message M using the secret key K_X
R_X	Random value generated by X
RS_X	Long random string generated by X
$K_{X,Y}$	Shared secret key between X and Y
$T_{X,Y}$	Ticket that grants X access to Y
A_X	Authenticator containing time stamp, generated by X
PK_X	X's public key, forms a key pair together with SK_X
SK_X	X's private key, forms a key pair together with PK_X
$CERT(X, PK_X)$	Digital certificate containing identity and public key of X
$CERT(X, PK_X, SIG_Y)$	Digital certificate signed by Y, $SIG_Y = \{H(CERT(X, PK_X))\}SK_Y$

Table 1.1: Notation

Chapter 2

Password security

Authentication using passwords has been used for as long as multi-user computer systems have existed. In order to understand how password-based authentication can be attacked, and how to protect against such attacks, this chapter will present the arms race between attacks and techniques used to defend against them. This understanding is required to be able to design a secure single sign-on system.

When using password-based authentication, there are three participants: The client, server and the network. An attacker can try to subvert one or more of these to gain knowledge of the password or to authenticate without knowing the password.

2.1 Server attacks

A server that serves multiple users must store enough information to authenticate its users. The simplest approach is to store a file or database with one entry per user, listing the user name and password. To authenticate, a user must present the user name and password to the server, which checks that the password is correct. One vulnerability with this approach is that if an attacker is able to retrieve the password file or database, all user passwords are compromised.

To make the attacker's job more difficult, the server can store a hash of each password instead of storing the password itself. When verifying a password, it then computes the hash value and compares this to the value stored. If they are equal, the user is authenticated. If an attacker is able to retrieve the password file in this scenario, the passwords are not readily available. The possible key space of passwords is large: For 8-character ASCII passwords[1],

[1] Counting only printable ASCII characters

there are 94^8, or 2^{52} possible passwords. It is possible to brute force this key space for an attacker with access to a lot of computational resources, but it is a time consuming task. However, one fact makes the attacker's job a lot easier: Human beings tend to choose passwords that are easy to remember. As such, it is far more likely that a password is 'August01' than '4$|x)!H['. This knowledge forms the basis of a dictionary attack. A dictionary attack is performed by first compiling a dictionary of commonly used words and phrases, and then running each word through the hash function and comparing the result to the hashed passwords in the password file. If a match is found, the attacker knows the password. More advanced dictionary attacks even use permutations of the words in the dictionary, such as semi-random capitalization and substituting numbers for letters. As mentioned, dictionary attacks work because human beings do not choose random passwords. If an attacker is able to retrieve the password file on a multi-user system with tens of thousands of users, several passwords are usually found within a few seconds using freely available tools such as John the Ripper[46]. A commonly used defense against this type of attack is to enforce a password policy, e.g. that passwords must be at least 8 characters long, must contain upper case letters, lower case letters and numbers, must not contain ordinary dictionary words, and must be changed frequently.

Another technique that is used to make a dictionary attack more difficult is the use of salt. For each password in the file, a random value, salt, is concatenated with the password before it is hashed. The salt is stored in clear text along with the password. Using salt gives two benefits: First, it is far more difficult to pre-compute a lookup table of password-hash value pairs, because each password has several possible hash values. Second, it makes the dictionary attack more difficult, because the password guess must be hashed once per salt, instead of hashing once, then comparing the result to all the values in the file. When using salt, two users with the same password might have different hash values.

So far, the only server attack described is stealing the password file, but what if an attacker is able to compromise the server and take full control of the operating system? The authentication software could then be modified to log all passwords in clear text to a file available to the attacker. This problem cannot be solved using conventional password authentication, because the user must present the password in clear text to the server, otherwise the server has no way of actually authenticating the user. The only defense is to try to harden the operating system so that an attacker is unable to compromise the server. Another approach is, of course, to use a single sign-on solution where passwords are never submitted to servers, only to a central authentication service.

Another trend in server password security is to store password hashes in a central database or directory. The users still submit their passwords to the server, but the server performs the password lookup by querying the database instead of a local file. This technique protects against password file compromise from the servers, but it does not protect against Trojan Horse authentication software.

2.2 Client attacks

Client computers generally have fewer users than servers, so attacking a single client does not have the same pay off as a successful server compromise, although an attacker performing a directed attack against a single individual might target the client used by that person. Directed attacks aside, automated attacks, such as worms and viruses, are able to attack a large number of clients simultaneously. These attacks are serious in several ways, but from a password security perspective they pose four major threats.

The first two threats are already described in section 2.1: Compromise of password files and Trojan Horse authentication software that logs passwords in clear text. Even if clients do not store ordinary password files, applications such as web browsers provide convenience functions to store user passwords. The passwords are stored in files, encrypted or in clear text, and those files could be stolen by an attacker. The encryption used in this scenario is usually password-based, which means that it is possible to perform dictionary attacks to retrieve the clear text passwords.

The third threat, which is usually specific to clients, is the use of software or hardware key loggers. These log everything that is entered on the keyboard of the client, including passwords and other sensitive information.

The fourth and final threat described in this section is overwriting configuration data on the client. This could be used to redirect the client to a fake server that harvests passwords or other information. If the client does not authenticate the server it connects to, such an attack is even easier, and could be carried out by DNS cache poisoning, ARP poisoning, or even URL spoofing. Server authentication prevents these trivial attacks, but does not protect against system files being replaced.

Defending against client attacks is extremely hard when using password-only authentication. If the client is compromised, there is no way to avoid password compromise, because the user has to enter the password in clear text on the client to be able to authenticate. One possible line of defense is to centrally manage client configuration and security updates, to minimize the risk of client compromise.

2.3 Network attacks

Network attacks are generally divided into two categories: Active and passive attacks. An active attacker sends or modifies information on the network, while a passive attacker just records traffic.

As described in section 2.1, dictionary attacks are very effective. This means that an authentication protocol should try to prevent off-line dictionary attacks. An off-line attack is an attack where the attacker learns enough information from a protocol run to be able to perform a dictionary attack without having to interact any further with any of the participants in the authentication protocol. In other words, the attacker captures information equivalent to that stored in the password file on the server. On-line attacks, or attacks where the attacker must interact with one of the participants for each password guess, are less serious, because such an attack can be audited on the server or client, and countermeasures such as throttling or restricting the number of failed password guesses can be used.

The most trivial of all network attacks is the passive attack on clear text communications. If the communication between the client and server is not encrypted, an attacker with access to the network is able to retrieve the password by simply recording the network traffic.

One well known active attack is the Man in the middle (MITM) attack. In this attack, the attacker is positioned between the client and server, and is able to capture all traffic as well as modify it. MITM attacks are possible when the authentication protocol does not provide mutual authentication. A conventional password protocol is vulnerable to such an attack, because the client does not authenticate the server. Thus, the attacker can act as the server, redirecting the traffic from the client to itself, then forwarding it to the real server. By doing this, the attacker will be able to capture the clear text password.

Replay attacks are another class of active attacks. The attacker records traffic, such as an authentication message from the client to the server. At a later point in time, the attacker then replays the traffic, to be able to authenticate. One countermeasure against replay attacks is challenge-response authentication.

2.4 Password security summary

In a single sign-on system that uses password-only authentication, it is impossible to defend against client compromise. If an attacker controls the client, then the password will be compromised. However, using a central AS

provides a number of security benefits, especially on the server side. First of all, passwords are only stored on the AS. Compromise of other servers does not give the attacker access to password files. Second, users never submit their passwords to the servers, only to the AS. This prevents the Trojan Horse authentication software scenario from section 2.1. Naturally, this means that the AS must be protected at all costs. Compromise of the AS equals compromise of password data for all users in the system. The reasoning behind centralizing password data is that it is easier to protect a few central servers, running only an authentication service, than to protect hundreds of different servers running all sorts of different services.

The effect on network attacks is not so clear. Some single sign-on solutions, such as Kerberos, are vulnerable to off-line dictionary attacks by performing a passive network attack (eavesdropping). Other solutions, usually based on public-key techniques, are not vulnerable to either passive or active network attacks.

As a conclusion, a password-only single sign-on solution protects against server attacks, although it does this by moving the vulnerability to a central AS. It provides no protection against client attacks. It may provide protection against network attacks, but this must be examined carefully.

Chapter 3

Single sign-on

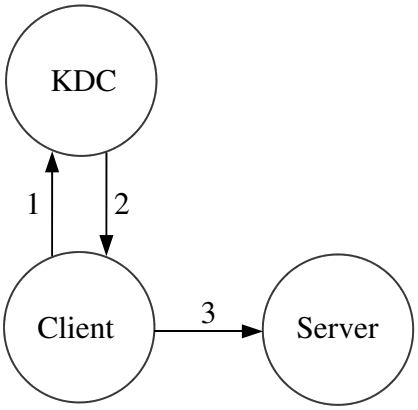
There are several different single sign-on solutions in use today. Some solutions, such as Kerberos, are based on secret-key cryptography, while others are based on public-key cryptography. There are also hybrid systems that combine the two approaches. In this chapter, existing and suggested solutions will be presented and analyzed. The goal is to try to learn from previous work, to avoid common errors and to gather ideas for the system design.

3.1 Kerberos

Kerberos[43] version 5 is probably the most widely deployed single sign-on solution used for computer access to networked resources today. Kerberos was originally developed as part of Project Athena at the Massachusetts Institute of Technology (MIT), [20] gives an overview of the design. Kerberos versions 1 through 3 were only used internally at MIT, while version 4 saw widespread use outside of MIT. The newest version of Kerberos, version 5, is available in several open source implementations[44][45], is standardized as Request For Comments (RFC) 1510[21], and is the default authentication mechanism in Microsoft Windows 2000[56] and 2003[57]. No treatise on single sign-on solutions would be complete without Kerberos.

Kerberos is, as mentioned previously, based on symmetric cryptography. Kerberos uses a Key Distribution Center (KDC) that shares a secret key with all entities in the network. The shared secrets with other computers are strong keys, while the shared secrets with human users are weak keys derived from memorizable passwords. The idea is that the password is only used to authenticate to the KDC, not to other computers. An overview of Kerberos V 5 authentication is presented here, but superfluous details are omitted. For a more detailed description see [21].

When users wish to access a server, they request a digital ticket from the KDC. The KDC generates and sends a session key encrypted with a key derived from the user's password, and a ticket encrypted with the server's secret key. The ticket contains the server's identity, the user's identity, the client's IP address, a session key, a time stamp and a validity period. When the client forwards the ticket to the server, it is validated by decrypting it with the server's secret key. Additionally, an encrypted authenticator value containing the user's identity, client's IP address and a time stamp is sent together with the encrypted ticket. The authenticator is decrypted by the server using the shared session key from the ticket, and the client is authenticated by checking that the time stamp is current. The whole process is illustrated in figure 3.1.

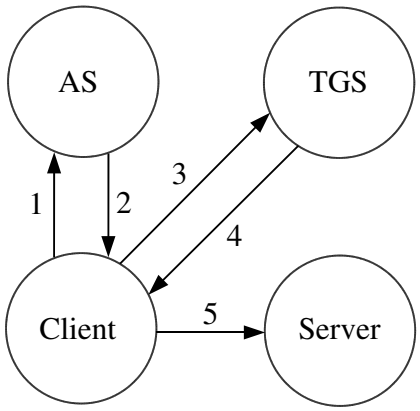


1: $C \rightarrow KDC$: C, S, R_C
2: $KDC \rightarrow C$: $\{K_{C,S}, R_C\}K_C, \{T_{C,S}\}K_S$
3: $C \rightarrow S$: $\{A_C\}K_{C,S}, \{T_{C,S}\}K_S$

Figure 3.1: Simplified Kerberos V 5 authentication

One problem with the simplified Kerberos authentication in figure 3.1 is that the client has to store the password for the system to have the single sign-on property. To avoid storing the password on the client, the KDC is logically split into two separate functions, an AS and a Ticket Granting Server (TGS). In practical implementations, the AS and TGS usually reside on the same physical server. First, the client obtains a Ticket Granting Ticket (TGT) from the AS. This TGT typically has a lifetime of about 8 hours, and is used whenever the client needs a ticket from the TGS. The TGS provides ordinary service tickets, just like the KDC in figure 3.1. The full Kerberos V

5 authentication is illustrated in figure 3.2.



1: $C \rightarrow AS$: C, TGS, R_C
2: $AS \rightarrow C$: $\{K_{C,TGS}, R_C\}K_C, \{T_{C,TGS}\}K_{TGS}$
3: $C \rightarrow TGS$: $\{A_C\}K_{C,TGS}, \{T_{C,TGS}\}K_{TGS}, S, R'_C$
4: $TGS \rightarrow C$: $\{K_{C,S}, R'_C\}K_{C,TGS}, \{T_{C,S}\}K_S$
5: $C \rightarrow S$: $\{A'_C\}K_{C,S}, \{T_{C,S}\}K_S$

Figure 3.2: Kerberos V 5 authentication

Kerberos V 5 has a number of known security issues, referred to as "Environmental assumptions" in RFC 1510[21]. The most serious of these is that an attacker can mount an off-line password dictionary attack just by observing the network traffic when a user receives a TGT, shown as message 2 in figure 3.2. The attack is carried out by guessing a password, deriving the corresponding secret key, and trying to decrypt the captured traffic. If the trial decryption yields a valid message then the password guess is correct. Another vulnerability is that Kerberos relies on the use of time stamps to defeat replay attacks, and therefore all participants must have synchronized clocks. Attacks on the network time protocols may allow an attacker to replay an old, expired authenticator.

There are at least three noteworthy differences between ticket-based authentication in Kerberos and challenge-response authentication schemes based on public-key cryptography: First, in Kerberos, the AS and TGS generate all the session keys, as opposed to e.g. TLS[2], where the client and server mutually generate a shared session key. This means that all traffic on the network can be decrypted by either the AS or the TGS, because one of them knows the session key. The CA in a PKI, on the other hand, is not able to

decrypt any traffic at all. Second, Kerberos relies on cached authenticators and loosely synchronized clocks to avoid replay attacks. Challenge-response authentication using public-key techniques can be constructed so that they are not vulnerable to replay attacks, without any need to cache authenticators or reliance on clock synchronization. Third, the KDC shares secret keys with all principals, while a CA just certifies public keys. This means that a KDC compromise in Kerberos is far more devastating than a CA compromise in a PKI, assuming that the CA does not know the private keys of its users.

3.2 Extensions to Kerberos

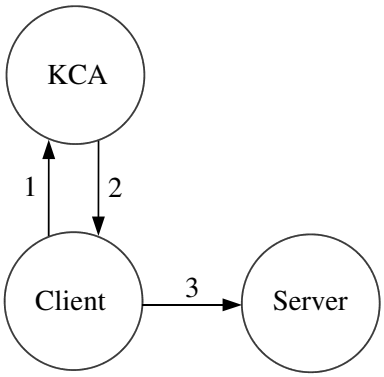
To address vulnerabilities or provide new functionality, several extensions to the Kerberos authentication protocol have been proposed. This section will present an overview of some relevant extensions.

First of all, to avoid the off-line password guessing attack, and to allow authentication to a KDC without the need for a pre-shared secret key, PKInit[33] was designed. This extension uses public key cryptography for the initial authentication (messages 1 and 2 in figure 3.2), which removes the vulnerability where an attacker can verify password guesses just by capturing network traffic. PKInit is not yet standardized as an RFC, it is currently an Internet Draft. PKInit assumes that a PKI is already deployed, which is an assumption that does not hold for a large number of organizations.

KX.509[8][22], designed and implemented at the University of Michigan, is conceptually the opposite of PKInit. Where PKInit uses a PKI to bootstrap Kerberos, KX.509 uses Kerberos to bootstrap a PKI, by using a Kerberos-enabled CA referred to as the KCA. The KCA may be integrated with the KDC, or an independent service altogether. When a client needs a certificate, it will first generate a key pair, then send the public key along with its Kerberos credentials to the KCA. The KCA will verify the credentials, then construct and sign a digital X.509 certificate. KX.509 shares several properties with the proposed solution in [9]: It generates key pairs on demand, signs certificates on-the-fly, and uses password-only authentication to bootstrap a PKI. Another interesting detail is that the UNIX and Mac OS versions include an implementation of a PKCS#11[51] software token, to maximize the number of compatible client applications: If an application supports PKCS#11 cryptographic tokens, then it supports KX.509. The Windows version uses the Windows Registry to store keys.

The process of acquiring a signed certificate in KX.509 is illustrated in figure 3.3. It is assumed that the client has already obtained a service ticket for the KCA by performing steps 1-4 in figure 3.2. Step 1 in figure 3.3

corresponds to step 5 in figure 3.2. Before the first message is sent, the client generates an RSA key pair. The RSA public key is submitted to the KCA along with credentials and a keyed HMAC. The KCA returns a signed X.509 certificate, which can be used for public-key based strong authentication.



1: $C \rightarrow KCA$: $\{A_C\}K_{C,KCA},\ \{T_{C,KCA}\}K_{KCA},\ PK_C,$
$HMAC_{K_{C,KCA}}(V, PK_C)$
2: $KCA \rightarrow C$: $CERT(C, PK_C, SIG_{KCA}),$
$HMAC_{K_{C,KCA}}(V, CERT(C, PK_C, SIG_{KCA}))$
3: $C \rightarrow S$: TLS handshake or equivalent

Figure 3.3: KX.509 authentication

Due to the off-line password guessing vulnerability in Kerberos, there have been a lot of suggestions for alternatives for initial authentication or pre-authentication. There are currently no RFCs on the subject. [35] gives an overview of the different suggestions, but it expired as an Internet Draft on December 22, 2003. Chapter 4 will look further into one of the suggestions from [35]: The use of password authenticated key exchange.

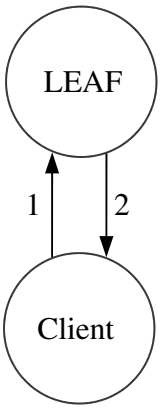
3.3 SPX

On January 25 1991, SPX[32] was published, both as a protocol description and as a reference implementation. SPX was an attempt to provide the same authentication functionality as Kerberos by using public-key cryptography with X.509 certificates. Its aim was to provide a scalable infrastructure for

mutual authentication, without the need for globally trusted third parties or CAs.

SPX is complex, and a thorough analysis will not be performed here, but there are two aspects of SPX that are relevant to the task at hand.

The first one is the Login Enrollment Agent Facility (LEAF). The LEAF stores users' private keys, encrypted using a symmetric algorithm with a hash of the user's password as the encryption key. Additionally, the LEAF stores a hash of the passwords, using a different hash function than the one used for generating the encryption key. Users authenticate to the LEAF using their user name and the hashed password, encrypted with the LEAF public key. The LEAF public key is assumed to be distributed to all clients using an out-of-band mechanism. When authenticated, users are able to retrieve their encrypted private key. The authentication and retrieval is illustrated in figure 3.4.



1: $C \rightarrow LEAF$: $C, \{TS, R_C, H(P)\}PK_{LEAF}$
2: $LEAF \rightarrow C$: $\{\{SK_C\}H'(P)\}R_C$

Figure 3.4: Download of encrypted private key from LEAF

The other interesting aspect of SPX is that instead of storing a long-term private key on an untrusted client computer, a temporary key pair is generated, and the public part of the key pair is wrapped in a certificate and signed by the user's long-term private key. After the certificate is signed, the long-term private key is destroyed. This is a very early reference to on-the-fly generation and signing of key pairs used for public-key based authentication.

3.4 SESAME

Secure European System for Applications in a Multi-vendor Environment (SESAME)[52] was initiated as an effort to implement the standard ECMA-219[54], which was released in January 1995. SESAME is a security framework with both an Authentication Service (AS) and a Privilege Attribute Service (PAS). The PAS provides authorization. Since this thesis deals strictly with authentication, only the AS will be discussed.

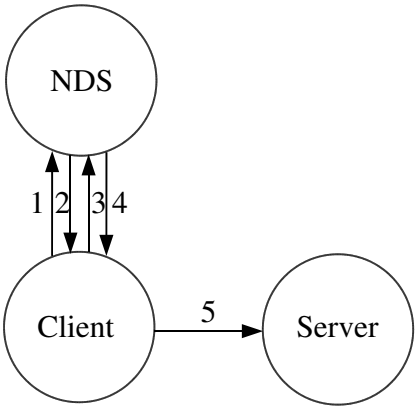
Due to the widespread use of Kerberos at the time, the developers decided that the SESAME implementation should be compatible with Kerberos. The SESAME authentication service was therefore implemented using the same mechanisms as Kerberos, but with the addition of optional public-key based authentication, similar to PKInit[33]. From an authentication viewpoint, this works exactly like the authentication illustrated in figure 3.2 from section 3.1. In a nutshell, SESAME authentication is Kerberos with "PKInit".

3.5 Novell NetWare 4

Novell NetWare version 4 contains a mechanism for password-based single sign-on, using public-key cryptography. [24] gives an overview of the authentication process, which is illustrated in figure 3.5. The Novell Directory Server (NDS) stores users' public keys, encrypted private keys, hashed and salted user passwords and the corresponding random salts, one salt per user. The authentication protocol uses four messages to avoid giving an attacker the encrypted private key before the client is authenticated, this is to avoid off-line password dictionary attacks by trial decryption of the private key. The client authenticates to the NDS in message 3 in figure 3.5, by encrypting the random challenge from the NDS with the hashed password. The server authenticates to the client in message 4, by using the nonce R_C as part of the message. The protocol description in [24] does not specify how the client obtains its public key, although it would be trivial to include this value in message 4. Figure 3.5 therefore includes the public key as well, to make the description complete.

There is one major problem with the authentication protocol used in Novell NetWare 4. [24] states:

> The user will need the server's public RSA key for Step F. This is obtained by querying NDS before Step F and reading the Public RSA Key attribute of the server object. It is important to note that any unauthenticated client has access to the Public RSA key attribute.



1:	$C \rightarrow NDS$:	C
2:	$NDS \rightarrow C$:	S_P, R_{NDS}
	C:	$K_{C,NDS} = \{R_{NDS}\}H(S_P, P)$
3:	$C \rightarrow NDS$:	$\{R_C, RS_C, K_{C,NDS}\}PK_{NDS}$
4:	$NDS \rightarrow C$:	$\{R_C, SK_C \oplus RS_C, PK_C\}K_{C,NDS}$
5:	$C \rightarrow S$:	Public-key based strong authentication

Figure 3.5: Authentication in Novell NetWare 4

Step F corresponds to message 3 in figure 3.5. [24] does not specify how the client verifies that the public key obtained is indeed the correct one. If it does not validate the public key, an active attacker impersonating the NDS could learn enough information to perform an off-line password dictionary attack just by sending message 2, sending its own public key, and then decrypting message 3 with its own private key. It would then be possible to do trial decryption of $\{R_{NDS}\}H(S_P, P)$. The attacker could simulate a network failure after message 3 to avoid detection. This attack can be avoided by distributing the NDS public key to all clients using an out-of-band mechanism.

It should also be noted that, in contrast with the LEAF in SPX, the NDS is able to decrypt the private keys of its users, due to the fact that $H(S_P, P)$, which is stored on the NDS, is used as the encryption key. LEAF makes an effort to avoid this by using two different hash functions, one for authentication and another one for encrypting the private key.

3.6 PKI

The last single sign-on mechanism described in this chapter is public-key based authentication supported by a PKI. If a PKI is in place to manage digital certificates, existing strong authentication mechanisms, such as those provided by TLS, can be utilized. As mentioned in the introduction, the problem is that such an infrastructure is usually not available. Had a PKI been in place, there would not be a need for other single sign-on solutions. The definition of a PKI in the introduction states that it must provide certification, distribution and revocation of digital certificates.

To provide certification, a PKI usually has a CA that certifies digital certificates by signing them. The CA root, or self-signed, certificate is distributed out-of-band to all members of the PKI. When users wish to enroll, they have to prove their identity to the CA. The CA then verifies that the user possesses the private key that corresponds to the public key in the digital certificate before it signs the certificate. All members of the PKI trust that the CA does not sign a fraudulent digital certificate.

Distribution of digital certificates is not a major problem, one way to do it is to put all the certificates in a public directory, and provide access through Lightweight Directory Access Protocol (LDAP)[34]. Another solution is to let all members of the PKI carry their own certificates. When authenticating, they would simply present their certificate to their peer, who would be able to verify the CA signature on the certificate using the public key from the CA root certificate.

Providing revocation of digital certificates is a challenge in any PKI. There are three main approaches towards solving this. The first one is to regularly post a Certificate Revocation List (CRL) that contains all the certificates that are revoked. This approach does not scale well, especially if there are a lot of users in the PKI and certificates have long lifetimes, as the CRL would have to contain all revoked certificates that have not yet expired. Another issue is that real-time revocation of certificates is not possible, clients must download the CRL on regular intervals, not check it every time they perform an authentication. One workaround is the use of a delta CRL, that only specifies the changes from the previous CRL. The second approach is to have an online service that provides certificate status information in real-time. An example of a protocol providing such a service is the Online Certificate Status Protocol (OCSP)[26]. The problem with this approach is that all participants must be on-line whenever they wish to authenticate, although this is for obvious reasons not a problem for authentication to on-line services. Another potential issue is the amount of traffic generated, every single authentication generates an OCSP request. The third, and probably simplest, approach is

to issue certificates with very short lifetimes. To be able to do this, the whole process of issuing certificates needs to be automated, especially if the solution should be able to scale to tens of thousands of users. One problem with using short certificate lifetimes as a revocation mechanism is that it introduces a reliance on synchronized clocks, just as in Kerberos.

3.7 Single sign-on summary

There is one aspect of a secure system that is not addressed by a PKI: Private key management. A PKI handles public keys, but on the assumption that all participants are able to keep their private keys private. One way to assure this is to use hardware tokens to store the private keys. The token itself would perform the cryptographic functions that use the private key, so that the key never leaves the hardware token, even if used on a compromised client. One example of such a token is a smart card, which acts both as storage and as a cryptographic processor. If hardware tokens are not available, one could instead store encrypted private keys on an on-line server. Users would then be able to download their private keys, but only the user that knows the password that decrypts the key will be able to use it. A third approach would be to avoid storing keys altogether, and instead generate everything on-the-fly. If one takes a closer look at the solutions presented in this chapter, as well as the SACRED protocol in section 4.3, one would see that they employ one or more of these three techniques. Plain Kerberos, KX.509 and SESAME generate credentials, in the form of tickets or digital certificates, on-the-fly. In Novell NetWare 4 and SACRED, users download an encrypted private key. SPX uses a hybrid approach where users download an encrypted private key, use that key to sign the public part of a key pair generated on-the-fly, and then destroy the downloaded private key. Finally, the solutions that rely on a PKI, such as SESAME with public-key based authentication and PKInit, as well as the more general public-key based system supported by a PKI, are in principle able to use any one of the three approaches.

Chapter 4

Building blocks

In addition to the solutions presented in chapter 3, there are several other techniques that are useful as building blocks for the design of a PKI-based single sign-on solution. This chapter contains a review of a few useful protocols and schemes, and forms the last part of the background material needed for the construction of the system.

4.1 PKAS

The field of password-based cryptography was mostly non-existent until 1992, when [5], the specification of Exponential Key Exchange (EKE), was published. Following EKE, a large number of protocols for password-authenticated key agreement were developed and published. Today, the Institute of Electrical and Electronics Engineers (IEEE) are currently working on standardizing password-based cryptography through its P1363.2 working group. At this point in time, P1363.2 has not been published as a standard, but a draft[49] is available. Another useful resource is [17], a collection of links to research papers in the field of password-based cryptography, maintained by David P. Jablon, the inventor of Simple Password Exponential Key Exchange (SPEKE).

P1363.2 describes two general approaches: Downloading a private key and PKAS. PKAS is further subdivided into Balanced PKAS (BPKAS) and Augmented PKAS (APKAS). A BPKAS is a scheme where the AS stores data that are password-equivalent. That is, although a password file compromise might not directly reveal the passwords, it reveals enough information for an attacker to impersonate any user. Schemes that are classified as BPKAS include EKE[5], SPEKE[18] and Password Derived Modulus (PDM)[19]. An APKAS, on the other hand, protects against password file compromise. Even

if the password file is stolen, the attacker has to perform a dictionary attack to be able to impersonate a user. Schemes that are classified as APKAS include AMP[23], B-SPEKE[16], A-EKE[4], PDM[1], SRP-3[36] and SRP-6[37]. In general, a scheme classified as APKAS will be more complex and require more computational resources than one classified as BPKAS. The advantage is that the APKAS provides some protection in the event that the password file is compromised.

Most of the password-based key agreement schemes are constructed using a modified Diffie-Hellman (DH) key exchange[7]. In EKE, the DH public key is encrypted using a password-derived key. In SPEKE, the generator g in the DH group is derived from the password. In PDM, the modulus p of the modular arithmetic operations in DH is derived from the password. In general, the different approaches use a password-derived value as a component of one of the parameters of the DH key exchange. Of all the different schemes mentioned, only SPEKE will be presented in detail. The motivation for choosing SPEKE is its simplicity, although any BPKAS would be usable for the purpose of constructing a single sign-on system. The motivation for choosing a BPKAS instead of an APKAS is fully explained in section 5.1.

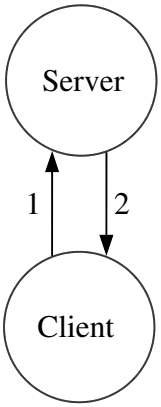
4.2 SPEKE

SPEKE provides authenticated key exchange, and is performed in two steps. The key exchange, shown in figure 4.1, uses the user password to generate the group parameter g in the DH key exchange. The value is computed as $g = H(P)^2$. The server stores user names and corresponding values of g. Additionally, both server and client could be pre-configured with p, the modulus for all the modular arithmetic operations. The SPEKE specification recommends that p should be a safe prime, that is, $p = 2q + 1$, where q is also prime.

After the key exchange, the parties compute a shared secret key. The client computes $K = (g^{R_C})^{R_S}$, while the server computes $K = (g^{R_S})^{R_C}$. To provide forward secrecy, both client and server compute the session key $K_{C,S} = H(K)$. After successfully computing the session key, the client and server may authenticate each other. A simple authentication process is shown in figure 4.2. An HMAC or any other mechanism verifying the knowledge of a shared secret could be used in place of the repeated hashing shown in figure 4.2.

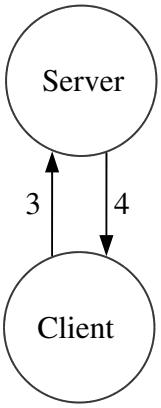
There are several constraints that must be applied to the basic SPEKE illustrated in figures 4.1 and 4.2 to thwart known attacks. One of them is

[1]PDM has two variants, one is a BPKAS while the other is an APKAS



1: $C \rightarrow S$: $C, g^{R_C} \bmod p$
2: $S \rightarrow C$: $g^{R_S} \bmod p$

Figure 4.1: SPEKE



3: $C \rightarrow S$: $H(H(K_{C,S}))$
4: $S \rightarrow C$: $H(K_{C,S})$

Figure 4.2: SPEKE authentication

already applied, by squaring the value $H(P)$, so that $g = H(P)^2$, instead of just using $g = H(P)$. The reason for this is that it forces the exponentials $g^{R_C} \bmod p$ and $g^{R_S} \bmod p$ to be generators of the subgroup of order q. Re-

calling that $p = 2q + 1$, this protects against partition attacks on g^{R_C} and g^{R_S}. One more constraint is that both parties check that $K \neq 1$ after the key exchange is completed, to avoid subgroup confinement of K. Furthermore, each party must check that $g^{R_X} \neq 0$, otherwise an attacker could force a shared session key $K = 0$. SPEKE bases its security on the intractability of the Discrete Logarithm (DL) problem, for a detailed description of the DL problem see section 3.6 of [27]. In practice, this means that p must be sufficiently large to make the DL problem intractable. A p of 1024 bits length is usually considered sufficient. Finally, both parties must verify that p and q are indeed prime, unless p is distributed out-of-band, as mentioned previously. Details of these constraints are presented in section 4 of [18].

One recent attack on SPEKE is described in [38]. The attack is particularly serious when a short Personal Identification Number (PIN) is used in place of a password, although SPEKE using alphanumeric passwords might also be vulnerable. The attack is performed by finding exponential equivalence classes among the passwords, so that several password guesses may be performed in a single attack. [38] recommends the use of a hashing function, together with a large password space, to keep the probability of two hashed passwords being exponentially equivalent as low as possible. When using SPEKE for single sign-on, a reasonably large password space is used, and the passwords are hashed, so this attack should not be practical. However, one should keep an eye on recent developments in finding exponential equivalence classes from the output of well-known hash functions such as Message Digest 5 (MD5)[30] and Secure Hash Algorithm 1 (SHA-1)[53]. To quote [38]:

> After all, hash functions such as SHA-1 and MD5 were not designed to break exponential equivalence between integers.

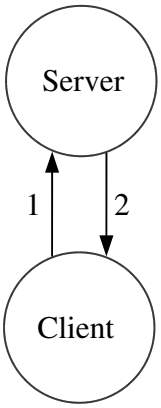
One last comment on SPEKE which is important for anyone that wishes to use it in an implementation is that it is patented in the United States. The US patent number is 6,226,383, and it was issued in 2001, so the patent expires in 2018.

4.3 SACRED

At the 1999 Internet Society Network and Distributed System Security Symposium, Radia Perlman and Charlie Kaufman presented a paper[29] discussing the use of strong password protocols for authenticated download of credentials, such as encrypted private keys used for public-key cryptography. Work outlining a framework for the secure download of credentials continued,

and in 2001 RFC 3157[3] stated the requirements for a protocol specification. Based on these requirements, the Informational RFC 3760[12], an abstract protocol framework for SACRED, was published in April 2004. Finally, in June 2004, the protocol specification for SACRED was published as Internet Engineering Task Force (IETF) RFC 3767[10]. The protocol specifies two different authentication mechanisms: Strong password protocols, such as SPEKE, and TLS mutual authentication.

As a source of building blocks for a single sign-on protocol, [29] is the most interesting, because the SACRED specifications do not specify the authentication protocol flow. Perlman and Kaufman, however, describe a lot of different protocols in their paper, ranging from two to six messages each. The protocols are based on SPEKE and EKE. Their two-message protocol using SPEKE is illustrated in figure 4.3. The shared SPEKE key, $H(g^{R_C R_S} \bmod p)$, is replaced by $K_{C,S}$ to enhance readability. See [29] for a description and security analysis of the protocols, and see section 4.2 for a description of SPEKE.



1: $C \rightarrow S$: $C, g^{R_C} \bmod p$
2: $S \rightarrow C$: $g^{R_S} \bmod p, \{SK_C\}K_{C,S}$

Figure 4.3: Private key download using SPEKE

4.4 CMP

CMP, specified in RFC 2510[1], and Certification Request Message Format (CRMF), from RFC 2511[25], define a protocol framework and message formats for maintaining a PKI. CMP is comprehensive, it covers everything from initial registration to certification and revocation of certificates, with a

large number of different actors. Due to its complexity, only a tiny subset of CMP is needed for the proposed system: The Certification Request and Certification Response messages between a client and the CA, as well as the Error Message.

The central element of CMP is the PKIMessage, which contains a PKIHeader and a PKIBody part, as well as optional PKIProtection and extraCerts fields. All messages in CMP are of type PKIMessage, and the structure of the PKIHeader is the same for all these messages. The PKIBody, however, has a different structure based on the message type, but in the proposed system the only possible values for the PKIBody are CertReqMessages, CertRepMessage and ErrorMsgContent, used for the Certification Request, Certification Response and Error Message respectively. The PKIMessage ASN.1 sequence, as well as the relevant parts of the PKIBody ASN.1 sequence, are presented in table 4.1. The PKIHeader is presented in table 4.2.

```
PKIMessage ::= SEQUENCE {
    header           PKIHeader,
    body             PKIBody,
    protection   [0] PKIProtection OPTIONAL,
    extraCerts   [1] SEQUENCE SIZE (1..MAX) OF Certificate OPTIONAL
}

PKIBody ::= CHOICE {       -- message-specific body elements
    cr      [2]  CertReqMessages,        --Certification Request
    cp      [3]  CertRepMessage,         --Certification Response
    error   [23] ErrorMsgContent         --Error Message
}
```

Table 4.1: PKIMessage and excerpts from PKIBody from RFC 2510

There are two fundamental concepts of CMP that should be thoroughly understood before reviewing the system design: Proof-of-Possession (PoP) and message protection. PoP is described in section 2.3 of RFC 2510. In CMP, the PoP contains a proof that the client making a Certification Request does indeed know the private key associated with the public key in the request. The PoP could be a digital signature, a HMAC, or simply a request for the CA to encrypt the digital certificate issued. Message protection, described in section 3.1.3 of RFC 2510, is a field used to provide message integrity, which is achieved through the use of a HMAC or digital signature.

Section 2.2.2 of RFC 2510 states that a CMP implementation must support the "Basic authenticated scheme" from section 2.2.2.2 to be a conforming

```
PKIHeader ::= SEQUENCE {
    pvno                INTEGER     { ietf-version2 (1) },
    sender              GeneralName,
    -- identifies the sender
    recipient           GeneralName,
    -- identifies the intended recipient
    messageTime     [0] GeneralizedTime         OPTIONAL,
    -- time of production of this message
    protectionAlg   [1] AlgorithmIdentifier     OPTIONAL,
    -- algorithm used for calculation of protection bits
    senderKID       [2] KeyIdentifier           OPTIONAL,
    recipKID        [3] KeyIdentifier           OPTIONAL,
    -- to identify specific keys used for protection
    transactionID   [4] OCTET STRING            OPTIONAL,
    -- identifies the transaction; i.e., this will be the same in
    -- corresponding request, response and confirmation messages
    senderNonce     [5] OCTET STRING            OPTIONAL,
    recipNonce      [6] OCTET STRING            OPTIONAL,
    -- nonces used to provide replay protection
    freeText        [7] PKIFreeText             OPTIONAL,
    -- this may be used to indicate context-specific instructions
    -- (this field is intended for human consumption)
    generalInfo     [8] SEQUENCE SIZE (1..MAX) OF
                            InfoTypeAndValue    OPTIONAL
    -- this may be used to convey context-specific information
    -- (this field not primarily intended for human consumption)
}
```

Table 4.2: PKIHeader from RFC 2510

implementation. As explained in section 5.2, this requirement means that the implementation presented in this thesis does not conform to RFC 2510.

One issue with CMP that should be mentioned is that RFC 2510 and 2511 are inconsistent. This is not a major issue, due to the fact that the inconsistencies are few and, with one notable exception, quite insignificant to the protocol. One such insignificant inconsistency is that RFC 2510 uses the term "Certification Request" for one of the messages, while RFC 2511 uses "Certificate Request". This thesis follows RFC 2510 whenever such inconsistencies occur. The one notable exception is an inconsistency in an ASN.1 sequence, shown in table 4.3, where a tag is missing in the sequence from RFC 2511. As mentioned earlier, this is not a major issue, as long as RFC 2510 is considered authoritative.

RFC 2510, section 3.2.8, page 28:

```
POPOSigningKeyInput ::= SEQUENCE {
    authInfo            CHOICE {
        sender              [0] GeneralName,
        -- from PKIHeader (used only if an authenticated identity
        -- has been established for the sender (e.g., a DN from a
        -- previously-issued and currently-valid certificate))
        publicKeyMAC        [1] PKMACValue
        -- used if no authenticated GeneralName currently exists for
        -- the sender; publicKeyMAC contains a password-based MAC
        -- (using the protectionAlg AlgId from PKIHeader) on the
        -- DER-encoded value of publicKey
    },
    publicKey           SubjectPublicKeyInfo    -- from CertTemplate
}
```

RFC 2511, section 4.4, page 4:

```
POPOSigningKeyInput ::= SEQUENCE {
    authInfo            CHOICE {
        sender              [0] GeneralName,
        -- used only if an authenticated identity has been
        -- established for the sender (e.g., a DN from a
        -- previously-issued and currently-valid certificate)
        publicKeyMAC        PKMACValue },
        -- used if no authenticated GeneralName currently exists for
        -- the sender; publicKeyMAC contains a password-based MAC
        -- on the DER-encoded value of publicKey
    publicKey           SubjectPublicKeyInfo }  -- from CertTemplate
```

Table 4.3: POPSigningKeyInput from RFC 2510 and RFC 2511

Chapter 5

Construction

Having reviewed existing solutions and gathered building blocks, the time has come to construct the single sign-on system. The goal is to be able to bootstrap a PKI using password-only authentication. The sections will go through requirements specification, system design, security analysis, performance analysis, implementation and performance testing. Please note that, as specified in the assignment, the implementation is experimental, and not ready for production use. That being said, much care has been taken to make it as scalable as possible. Its main purpose is as a reference for implementors of the design presented in this thesis.

5.1 Requirements

The first task of the system construction is to specify the system requirements. The original requirements from the assignment state that the system must provide real-time issuing of digital certificates with short validity times. In other words, the system uses the on-the-fly approach described in section 3.7, and short certificate lifetimes is used as the revocation mechanism, as described in section 3.6. Furthermore, the interpretation in section 1.3 states that the system must support password-only authentication, that the system must provide an AS with CA functionality, and that the AS should be stateless. Furthermore, the system will only handle authentication, authorization is performed locally by each server or service. The system must provide both AS and client software.

From the discussion in chapters 2, 3 and 4, it should be possible to formulate a set of security requirements. The system must not be vulnerable to server compromise, the server vulnerability will be moved to the AS. This means that it is generally accepted that an AS compromise equals total sys-

tem compromise. If an attacker has compromised the AS, all users can be impersonated by signing fraudulent digital certificates, and the only thing a password can be used for is just that, obtaining a signed certificate. In other words, there is no point in using an APKAS. If the AS is compromised, the attacker does not gain anything more by having access to clear text equivalent passwords. The result is that one might as well use a BPKAS, as mentioned in section 4.1.

The system must not be vulnerable to passive or active network attacks. More specifically, an active or passive attacker must not be able to learn enough information to perform an off-line dictionary attack, impersonate a user, or to perform a MITM attack, even if the attacker has full control of the entire network. The system will not provide any specific protection against network-based Denial-of-Service (DoS) attacks. As mentioned in section 2.4, a system that relies on password-only authentication is not able to provide protection against client compromise.

5.2 Design

Section 1.3 specifies that the goal of the system is to provide a PKI-based single sign-on solution. The design will focus on messages 1 and 2 from figure 1.1, the communication between the client and the AS. Once this exchange is completed, the client can use its digital certificate with existing public-key based authentication schemes, referred to as message 3. Authorization is, as mentioned previously, distributed, each server or set of servers have their own authorization schemes.

In addition to SPEKE, one cryptographic algorithm for digital signatures and one for symmetric encryption is required. Digital Signature Algorithm (DSA)[41] will be used for the signatures, although Rivest Shamir Adleman (RSA)[50] support will also be implemented to make performance comparisons. Advanced Encryption Standard (AES)[39] will be used as the symmetric algorithm. For SPEKE, the modulus p should be a safe prime, and generation of safe primes takes a long time, so it is not practical to generate p at login time. Furthermore, using a different p for each client would introduce security issues described in section 4.2. In particular, if the client sends p to the server, then the server would have to check that it is indeed a safe prime before continuing with the exchange. There is a simple solution to these issues: Use the same modulus p for the entire system.

The protocol used for communication between the client and AS is CMP. There are several reasons for using CMP. One is that this protocol was designed to maintain a PKI, and that is basically what it will be used for,

although in a very limited fashion. There is no point in inventing a new protocol when an existing one will do just fine. Another reason is that CMP directly supports authentication based on hardware tokens, through the use of digital signatures, so it is easy to extend the system to allow token-based authentication in the future. Furthermore, there are already CMP implementations available, so there is no need to implement an entire protocol from scratch. CMP uses X.509v3 certificates with the PKIX profile specified in [14], so the obvious choice is to use that certificate profile.

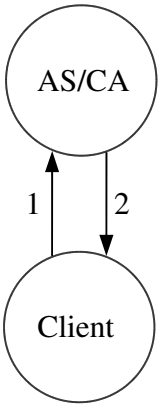
One assumption made is that the CA root certificate is distributed out-of-band to all participants. For an organization that directly controls all clients and servers through configuration management systems, this is not an issue, they can distribute the CA certificate through their existing infrastructure. For other scenarios, like users logging in from their home PCs, this assumption does not necessarily hold. Therefore, an alternative approach is also discussed in this section.

Since a stateless AS is preferred, the most obvious scheme would be a two-message exchange, similar to the one illustrated in figure 4.3, although it must be modified to provide on-the-fly signing of digital certificates, using CMP. The modified scheme is illustrated in figure 5.1. The modifications when compared to the private key download in figure 4.3 are that the client submits its public key to the AS, and that a signed and encrypted digital certificate is returned instead of an encrypted private key. As previously, $K_{C,CA}$ represents the shared secret key generated by SPEKE.

It should be noted that a reduction to two messages makes the system not conform to RFC 2510, because all conforming implementations must support "Basic authenticated scheme", which consists of three messages. The last message of "Basic authenticated scheme" is a Confirmation message from the client to the CA.

The next design challenge is how to use the Certification Request and Certification Response messages as containers for the SPEKE verifiers and the other parameters needed for issuing the digital certificate. The easy part is to determine where to store the identity and public key of the user. Table 5.1 shows the ASN.1 sequences for CertRequest and CertTemplate. The identity is stored as subject, while the public key is stored as publicKey.

For the Certification Request message, the most convenient way to allow the client to request an encrypted certificate is by using the PoP mechanism. CMP offers this to support requests for encryption certificates, but it will be used for signature certificates instead. The relevant ASN.1 sequences from RFC 2511 are listen in table 5.2. Thus, ProofOfPossession is set to keyEncipherment, POPOPrivKey is set to subsequentMessage, and SubsequentMessage is set to encrCert.



1: $C \rightarrow AS$: $C, PK_C, g^{R_C} \bmod p$
2: $AS \rightarrow C$: $g^{R_{CA}} \bmod p, \{CERT(C, PK_C, SIG_{CA})\}K_{C,CA}$

Figure 5.1: Issuing of digital certificate

```
CertRequest ::= SEQUENCE {
    certReqId     INTEGER,          -- ID for matching request and reply
    certTemplate  CertTemplate,  -- Selected fields of cert to be issued
    controls      Controls OPTIONAL }   -- Attributes affecting issuance

CertTemplate ::= SEQUENCE {
    version      [0] Version               OPTIONAL,
    serialNumber [1] INTEGER               OPTIONAL,
    signingAlg   [2] AlgorithmIdentifier   OPTIONAL,
    issuer       [3] Name                  OPTIONAL,
    validity     [4] OptionalValidity      OPTIONAL,
    subject      [5] Name                  OPTIONAL,
    publicKey    [6] SubjectPublicKeyInfo  OPTIONAL,
    issuerUID    [7] UniqueIdentifier      OPTIONAL,
    subjectUID   [8] UniqueIdentifier      OPTIONAL,
    extensions   [9] Extensions            OPTIONAL }
```

Table 5.1: CertRequest and CertTemplate from RFC 2511

The message protection in CMP is not used for password-only authentication. However, if one wishes to use public-key based authentication using hardware tokens, using a digital signature as message protection instead of using PoP makes sense, since the use of message protection provides both message integrity and sender authentication, while PoP provides only the latter. One might wonder why the system should be used at all if hardware

```
ProofOfPossession ::= CHOICE {
    raVerified        [0] NULL,
    -- used if the RA has already verified that the requester is in
    -- possession of the private key
    signature         [1] POPOSigningKey,
    keyEncipherment   [2] POPOPrivKey,
    keyAgreement      [3] POPOPrivKey }

POPOPrivKey ::= CHOICE {
    thisMessage       [0] BIT STRING,
    -- posession is proven in this message (which contains the private
    -- key itself (encrypted for the CA))
    subsequentMessage [1] SubsequentMessage,
    -- possession will be proven in a subsequent message
    dhMAC             [2] BIT STRING }

SubsequentMessage ::= INTEGER {
    encrCert (0),
    -- requests that resulting certificate be encrypted for the
    -- end entity (following which, POP will be proven in a
    -- confirmation message)
    challengeResp (1) }
```

Table 5.2: ProofOfPossession, POPOPrivKey and SubsequentMessage

tokens are already deployed. In general, there are two advantages: The first one is that it allows users to use either password-only or token-based authentication depending on the circumstances, so both authentication mechanisms can coexist. Adding a trust level field to the issued certificate could specify whether the user used password-only authentication or not, so that it would be possible for servers to specify the trust level needed to access its services. The second advantage is that the system provides the certificate revocation mechanism. If a user's long term private key, stored in the hardware token, is compromised, all that has to be done is to disable the user in the AS. The AS will thus refuse to issue any more certificates for that user, so the compromised private key will be worthless. In other words, only the AS needs to know whether a user's long-term key is valid or not.

One of the major issues is where to put the SPEKE public keys g^{R_C} and $g^{R_{CA}}$. After careful consideration, the senderKID field of the PKIHeader from RFC 2510 was used for g^{R_C} and $g^{R_{CA}}$. This optional field should originally be used to identify which key that was used for message protection, not to store the actual key, but it will instead be used to store the SPEKE public keys used for PoP. Other possibilities were considered, but due to the fact

that both the AS and client need to send their SPEKE public key, and that they send different message types with different PKIBody, means that the keys had to be put in the PKIHeader, which all types of PKIMessage contain. The only other realistic alternative was the generalInfo field, but using that one was a more complex option. In the end, simplicity won.

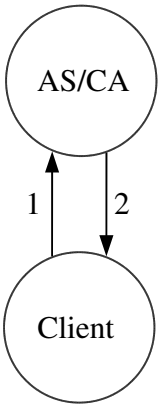
Another design issue is where to put the Initialization Vector (IV) needed for decryption of the certificate when using a cipher in Cipher Block Chaining (CBC) mode. Examining the ASN.1 sequence EncryptedValue in table 5.3, which is used to store an encrypted certificate, one can observe that the encSymmKey contains raw binary data, which is suitable for storing the IV. So, the IV is stored in the encSymmKey field, even though RFC 2511 states that its original purpose is to be a container for the symmetric key needed to decrypt the encValue.

```
EncryptedValue ::= SEQUENCE {
      intendedAlg   [0] AlgorithmIdentifier  OPTIONAL,
      -- the intended algorithm for which the value will be used
      symmAlg       [1] AlgorithmIdentifier  OPTIONAL,
      -- the symmetric algorithm used to encrypt the value
      encSymmKey    [2] BIT STRING           OPTIONAL,
      -- the (encrypted) symmetric key used to encrypt the value
      keyAlg        [3] AlgorithmIdentifier  OPTIONAL,
      -- algorithm used to encrypt the symmetric key
      valueHint     [4] OCTET STRING         OPTIONAL,
      -- a brief description or identifier of the encValue content
      -- (may be meaningful only to the sending entity, and used only
      -- if EncryptedValue might be re-examined by the sending entity
      -- in the future)
        encValue       BIT STRING }
```

Table 5.3: EncryptedValue from RFC 2511

Having designed the mechanism for issuing certificates, how does one solve the CA root certificate distribution problem? Actually, there is a simple way to solve this: If the client does not already have the CA root certificate, it can request it from the AS. The AS then encrypts the root certificate using the SPEKE shared secret key and includes this in the response. Only the AS is able to encrypt a message with the shared secret key, so this provides message authentication. This slightly modified scheme is illustrated in figure 5.2. The modification reduces the total system security slightly, and it increases the load on the AS by making it perform one extra symmetric encryption per request, but the advantage is that it removes the assumption that the CA root certificate must be distributed out-of-band. The performance penalty

should be minimal, due to the relatively low computational cost of symmetric operations compared to the asymmetric operations used for SPEKE and digital signatures.



1: $C \rightarrow AS$: $C, PK_C, g^{R_C} \bmod p$
2: $AS \rightarrow C$: $g^{R_{CA}} \bmod p, \{CERT(C, PK_C, SIG_{CA})\}K_{C,CA}$, $\{CERT(CA, PK_{CA}, SIG_{CA})\}K_{C,CA}$

Figure 5.2: Issuing of digital certificate and CA root certificate

At first sight, it may look like this modification could be trivially implemented in CMP by including the encrypted CA root certificate as the caPubs part of the CertRepMessage from RFC 2510, shown in table 5.4. However, this is not possible, because the caPubs field, of type Certificate, only supports clear text certificates. The solution is to add another CertResponse containing the encrypted CA root certificate to the CertRepMessage. In other words, the CertRepMessage consists of two CertResponses, the first one is the encrypted client certificate and the second the encrypted CA root certificate. In fact, the client does not even need to specifically request this, the encrypted CA certificate could always be included in the response from the AS. That way, if the client does not have the CA root certificate, it can use the encrypted one, otherwise it can just ignore that part of the response.

The whole discussion so far has focused on the initial issuing of a digital certificate using password-only authentication or hardware tokens. This certificate has a short validity period, a typical value could be one hour, due to the fact that short certificate lifetimes is used as the revocation mechanism. This means that there must be a way to renew certificates after the initial exchange. One alternative is to have the user enter the password each time a

```
CertRepMessage ::= SEQUENCE {
    caPubs          [1] SEQUENCE SIZE (1..MAX) OF Certificate OPTIONAL,
    response            SEQUENCE OF CertResponse
}

CertResponse ::= SEQUENCE {
    certReqId           INTEGER,
    -- to match this response with corresponding request (a value
    -- of -1 is to be used if certReqId is not specified in the
    -- corresponding request)
    status              PKIStatusInfo,
    certifiedKeyPair    CertifiedKeyPair    OPTIONAL,
    rspInfo             OCTET STRING        OPTIONAL
    -- analogous to the id-regInfo-asciiPairs OCTET STRING defined
    -- for regInfo in CertReqMsg [CRMF]
}

CertifiedKeyPair ::= SEQUENCE {
    certOrEncCert       CertOrEncCert,
    privateKey      [0] EncryptedValue      OPTIONAL,
    publicationInfo [1] PKIPublicationInfo  OPTIONAL
}

CertOrEncCert ::= CHOICE {
    certificate     [0] Certificate,
    encryptedCert   [1] EncryptedValue
}
```

Table 5.4: CertRepMessage with contents from RFC 2511

certificate is to be renewed, which would destroy the single sign-on property of the system. Another possible solution is to set a validity period of eight hours or more, to make the probability of a certificate renewal request during a single session low. This last technique is used by Kerberos. Nevertheless, the system should provide a way to renew certificates to be able to provide true single sign-on. This is actually rather straightforward, because after the initial exchange, the user has a valid certificate. When it is time to renew the certificate, the user generates a new key pair, then uses the old private key to make a digital signature as message protection in the Certification Request message. The AS can verify the signature through the old, but still valid, certificate, and issue a new certificate based on this. The old certificate should be included in the extraCerts field of the PKIMessage, as specified in RFC 2510. The motivation for using message protection instead of PoP is, as previously mentioned, that it provides both message integrity and sender

authentication, at no extra cost.

5.3 Security analysis

As stated in section 1.4, the security analysis focuses on the authentication scheme of the proposed solution. The fully constrained SPEKE from section 4.2 is used, to thwart all known attacks against SPEKE. The initial authentication to the AS is very similar to the two-message private key download using SPEKE from [29] illustrated in figure 4.3. Using a two-message scheme is a trade-off: It makes the AS stateless, but the cost is that mutual authentication is no longer possible. The AS does not know if the client is legitimate or not, it signs and encrypts a certificate without any client authentication. However, a client that does not know the password will not be able to decrypt the certificate or perform an off-line password-guessing attack, due to SPEKE being randomized. What this means is that an active attacker is able to do unaudited on-line password guessing. But, as Perlman and Kaufman point out in [29]:

> Although Bob cannot distinguish a legitimate download from a password guess, he ought to get suspicious if the same user requests thousands of password downloads within a short time.

The countermeasure for this vulnerability is to log authentication attempts per user, and implement throttling or a maximum number of requests per time period. Care should be taken not to introduce DoS vulnerabilities by doing so. As an example, if the AS only allows three initial authentications per user per day, then an attacker could remove a user's ability to use the system by simply performing the initial authentication three times using a fake password. A truly malicious attacker could disable the whole system for a day by doing the aforementioned attack once per user.

Two different options for CA root certificate distribution have already been discussed: Out-of-band distribution and the inclusion of an encrypted CA certificate in the initial response. By choosing the latter option, one sacrifices a little security to be able to use the system from clients that are not able to establish initial trust out-of-band. The security degradation is subtle: If an attacker is able to get the verifier $H(P)$, or the password P, it is possible to send the client a fake CA certificate by intercepting message 1 from figure 5.2 and sending the fake CA certificate as part of the response. By having the client accept this certificate as authoritative, the attacker can then proceed to redirect all traffic from the client to a server controlled by the attacker, with a server certificate signed by the fake CA private key. The

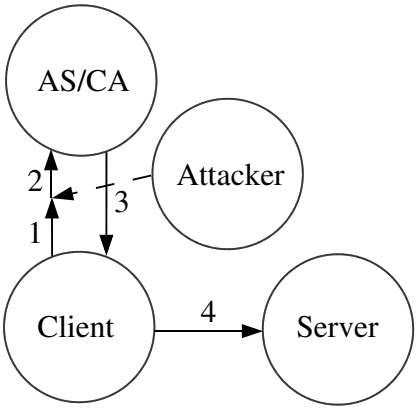
effect is that the attacker is able to trick the client into believing that a malicious server is legitimate. As an example, this could be used to collect personal information about the user to perform identity theft. In other words, if the CA certificate is not distributed out-of band, then an attacker can impersonate a legitimate server[1] to the user, but only *if the attacker already has the user's password*. In some special cases, this distinction could be important. The security recommendation is then, not very surprisingly, to distribute the CA certificate to clients out-of-band where possible, otherwise use the encrypted certificate from the response.

There is, however, a potentially serious vulnerability with the proposed authentication scheme. The origin of the vulnerability is that the messages in the initial authentication do not have message integrity protection. The consequence of this is that an active attacker can modify the messages without detection. To exploit this vulnerability, an attacker could generate a key pair PK_A, SK_A, then proceed as shown in figure 5.3, by modifying the first message from the client to the AS. The effect is that the client receives an encrypted certificate with the legitimate user's identity, but with the attacker's public key. The client then decrypts the certificate. Assume that the client does not check that the data in the certificate is valid, particularly that the public key is the same as the one sent by the client in message 1, and then tries to establish a session to a server using TLS authentication. During the TLS handshake, the client is requested to send its certificate. When it does, the attacker, who is busy capturing traffic, also gets the certificate. The attacker now has a valid certificate with the user's identity and the corresponding private key, and is able to impersonate the user. This attack would be detectable by the user, because the TLS authentication would fail, but it might already be too late by then.

To protect against the active attack illustrated in figure 5.3, the client must check the contents of the certificate received from the AS. Failure to check that the public key is identical to the one sent by the client gives an attacker the opportunity to impersonate the user.

The lack of message integrity protection raises another interesting point. As shown in figure 5.3, an attacker can replace the user's public key, and the AS will still sign the certificate. The consequence is that the AS is not actually acting as a CA when password-only authentication is used, because a CA is per definition supposed to verify that the public key in the certificate really does belong to the corresponding user identity before signing. In password-only mode, the AS should therefore be described as a signing

[1] Or another system user



1: $C \rightarrow AS$: $C, PK_C, g^{R_C} \bmod p$
2: $C \rightarrow AS$: $C, PK_A, g^{R_C} \bmod p$
3: $AS \rightarrow C$: $g^{R_{CA}} \bmod p, \{CERT(C, PK_A, SIG_{CA})\}K_{C,CA}$
4: $C \rightarrow S$: $CERT(C, PK_A, SIG_{CA})$

Figure 5.3: Active attack on authentication

server, or simply AS, rather than as a CA.[2] However, this is not an issue when using hardware tokens, due to the client's digital signature used for message protection, which binds the public key to the user's identity. One recommendation would thus be to add a field to the certificate that specifies the trust level, e.g. password-only or hardware token based. To avoid escalation of trust level, certificate renewal should only be possible when the old and new certificates have the same trust level. Furthermore, if the AS should be able to act as a CA in password-only mode, more than two messages is needed, which would violate the requirement that the AS should be stateless. Using a protocol with four messages would make it possible for the AS to verify the identity of the client and bind it to the public key in the certificate. One way to achieve this is by first performing SPEKE (messages 1 and 2), and then using the shared secret key in a keyed HMAC that provides message integrity for the signing request and response (messages 3 and 4). Using a protocol with more than two messages for the initial password-only authentication is not further elaborated in this thesis.

There are three different parameters for asymmetric cryptography used in the system presented. Those are the SPEKE modulus, the client DSA or

[2]The server is therefore referred to as the AS/CA, because it assumes one or both roles, depending on the circumstances

RSA modulus, and the CA DSA or RSA modulus. If the modulus is so short that an attacker is able to find discrete logarithms, or in the case of RSA, factor the modulus, the security of the system is compromised. The severity of the compromise varies based on which of the three moduli the attack is directed towards.

A successful attack on the client modulus would enable the attacker to impersonate that particular client, but only for as long as the attacker is able to renew the session. The attack gives the attacker access to the short-term private key generated on-the-fly. In other words, a live session is required for successful exploitation, and the attacker must be able to find the discrete logarithm in a shorter amount of time than the certificate life time.

The SPEKE modulus takes a very long time to generate, which makes it impractical to generate it in real time. This is because finding safe primes is very time consuming. Therefore, in the proposed system, the SPEKE modulus p is pre-shared among all participants. A successful attack on this modulus results in the attacker being able to find the exponents of the SPEKE, which then enables compromise of the verifier $H(P)$. This verifier can then be used to obtain a valid certificate from the AS.

Successfully attacking the CA modulus results in a total system compromise: The attacker then has the CA private key, and is able to sign valid certificates without any further interaction with the AS.

From a system-wide perspective, the CA modulus is the most critical, and should be at least as long as any of the others. The SPEKE modulus comes in second place, successful attacks against that one gives the attacker access to the shared secret verifier between a user and the AS. The least critical modulus is the client modulus, successful exploitation leads to session capture, but does not give the attacker enough information to successfully authenticate to the AS in the future, unless the attacker uses a captured session to change the user's password. The moral of the story is that in a production system, the CA modulus should not be shorter than any of the other two, and the SPEKE modulus should not be shorter than the client modulus. Or, to make it simple, just make sure that all three moduli are long enough to make such attacks impractical. 1024 bits would probably do just fine. In an experimental system, however, it may be interesting to test different combinations to see how the key lengths affect performance. More on that in section 5.6.

A few more words on key lengths: On page 166 of [31], a comparison between symmetric and asymmetric key lengths is presented. Since the choice of symmetric algorithm is AES, using the minimum key length of 128 bits, the symmetric key will always be the strongest part of the system as long as the moduli for SPEKE, DSA and RSA are less than 2304 bits in length. For

the performance tests, modulus lengths of 768, 1024 and 2048 bits will be used for asymmetric cryptography. The reasoning behind this is that a 768 bit modulus is the minimum recommended length for short-term security as per today, 1024 bits is considered good enough for many applications, and 2048 bits is considered generally safe. To speed up the SPEKE calculations, shorter exponents are used, as suggested in [28], which recommends that the length of the random exponents, R_C and R_{CA} in SPEKE, should be at least twice the length of the symmetric key produced by the exchange. This means a length of $128 * 2 = 256$ bits, since AES with a key length of 128 bits is used. However, [18] recommends using an exponent of length 336 bits, which is well above the minimum recommended value, but still short enough to provide a significant performance benefit, and this recommendation will be followed.

5.4 Performance analysis

This section provides a theoretical background for the performance tests described in section 5.6. The focus is on the AS, although one aspect of client performance, key generation times, will also be discussed. From the system design in section 5.2, one can observe that the AS has two main modes of operation: Initial issuing of certificates using SPEKE, and renewal of certificates using digital signatures for client authentication. Both modes of operation must be tested for performance. In both cases, the dependent (measured) variable is the number of issued client certificates per second, but the independent (manipulated) variables for the two modes of operation are different. The independent variables for each mode of operation are the parameters that have a significant impact on the performance of the system. An in-depth background on performance in cryptographic algorithms will not be presented here, for a more detailed description and analysis see [27]. The general assumption used is that modular exponentiation using a huge modulus is so computationally expensive that it eclipses the performance impact of other parameters. The main focus will therefore be on the most computationally expensive operations, the asymmetric ones. It is assumed that the symmetric operations and hashing have such a small impact that they can be ignored.

When using SPEKE, the client and server each have to perform two modular exponentiations, as explained in section 4.2. The first exponentiation produces the SPEKE public key, while the second produces the shared secret key. For DSA signing, the server or client has to perform one modular exponentiation. For DSA signature verification, they have to perform two modular exponentiations, although they can be performed simultaneously,

which results in the performance impact being approximately 1.17 modular exponentiations, see pages 453-454 of [27] for details. For RSA signing and signature verifications, one modular exponentiation is performed. However, a small number is often chosen as the encryption exponent e in RSA, which results in signature verification being a lot faster than signing. As a consequence, $e = 2^{16} + 1 = 65537$ is used as the encryption exponent, as recommended on page 291 of [27]. A summary of the performance impacts of the different operations is shown in table 5.5. For the initial authenti-

Mode	Number of modular exponentiations
SPEKE	2
DSA signature	1
DSA verification	1.17
RSA signature	$\gg 1$
RSA verification	$\ll 1$

Table 5.5: Theoretical performance comparison

cation using SPEKE, there are three independent variables: The SPEKE modulus, the CA modulus, and the CA signature algorithm. The only effect of changing the client modulus in the initial authentication is to make the PKIMessage a little longer, so this has a minimal impact on performance. This is why the client modulus and signature algorithm are not included as independent variables. For the initial issuing of a certificate, the AS has to perform two modular exponentiations using the SPEKE modulus, and one digital signature using the CA algorithm and modulus.

For renewal of certificates, there are four independent variables: The CA modulus, CA signature algorithm, client modulus and client signature algorithm. SPEKE is not used for renewal of certificates. In this mode of operation, the AS has to check the client's signature on the new certificate, then check the CA signature on the old certificate, and finally sign the new certificate.

The performance impact of the independent variables for the two different modes of operation are summarized in table 5.6.

Before performing actual tests, it might be enlightening to make a few conjectures about the performance of the various combinations of algorithms presented. First of all, one might assume that the initial issuing of certificates will be faster when using DSA than when using RSA. The argument supporting this is simply that signing using RSA is more time-consuming than when using DSA. When considering renewal of certificates, the situation is a little

Entity	Operation	SPEKE	CA sign	CA verify	Client sign	Client verify
AS	Initial	1	1	0	0	0
Client	Initial	1	0	1	0	0
AS	Renew	0	1	1	0	1
Client	Renew	0	0	1	1	0

Table 5.6: Number of performance critical operations

more complex. One thing that is known is that the AS has to perform two verifications and one signature. Based on the information in table 5.6, the fastest combination of algorithms for renewal would probably be RSA on the client and DSA on the AS. That would result in the AS performing one RSA verification, which is extremely fast, and then one DSA verification and signature. This is at least faster than using DSA on both the client and the AS, and in the last two cases the AS has to sign using RSA, which will probably result in lower performance.

One performance issue that has not yet been discussed is the time used for key generation on the client. For the initial authentication procedure, key generation times of more than a few seconds is probably unacceptable. Users do not wish to have to wait for a long time after entering their password before they are allowed to access the system. Based on conventional knowledge, DSA parameter generation is quite slow. Because of this, the parameters p, q and g used for DSA are often generated when the software is installed, rather than at login time. The key generation itself, when the parameters are available, is very fast when using DSA. With RSA, on the other hand, it is not possible to pre-compute key generation parameters, so key generation will be slower. Two questions that need to be answered through experiments is therefore: Is generation of RSA keys at login time a viable approach? And is the generation of DSA parameters fast enough to be used on-the-fly?

5.5 Implementation

To implement the system design, several technology choices had to be made. First of all, Java[42] was selected as the programming language, partly because of the level of proficiency of the implementor, and partly because of the availability of libraries, particularly a freely available CMP implementation. The version of Java used is J2SE 1.5.0_02, and the CMP implementation used is the NOVOSEC extensions[47] to the Bouncy Castle Crypto APIs[40]. Release 101 of the NOVOSEC extensions and version 1.27 of the Bouncy

Castle Crypto APIs are used. The selected transport mechanism for CMP is HTTP (Content-Type: application/pkixcmp), which makes it extremely easy to scale horizontally by using existing solutions for load balancing of web servers. The AS is implemented as a Java Servlet, and Jakarta Tomcat[55] version 5.5.9 is used as the Servlet container. A short description of the Tomcat configuration is presented in appendix A.

The first iteration of the implementation - test cycle was to provide a functional implementation using DSA as the signature algorithm. Furthermore, to generate CA certificates and server certificates for functionality testing, the Java classes in appendices E and F were used, for DSA and RSA certificates respectively. The difference between CA certificates and server certificates is that the CA certificates are self-signed, as illustrated in the Java class in appendix F. The reason for generating server certificates is that, even if the ordinary servers are defined as out-of-scope for this thesis, server certificates are a prerequisite for functionality testing using third party software.

The Java class in appendix G was used to generate the moduli used for SPEKE. This rather naive method of finding a safe prime is very time-consuming, but it got the job done. Generation of the 2048-bit modulus took over three hours on a server with two Intel Xeon 2.8 GHz CPUs.

The client and server were run on the same physical system, and the server implementation was basically just a doGet() method in a Java Servlet, while the client was run from the command line. All values were hard-coded, so recompilation was needed for configuration changes. Although the system as such was crude, it did provide the desired functionality. The user entered the user name and password into the client software, the client then initiated SPEKE and got its signed certificate from the server. Keys and certificates were stored as ordinary system files.

Quality assurance of the functionality was performed in several stages: First, the output of debug routines in the software was inspected, as shown in appendix B. Next, the resulting certificates and messages were inspected using the 'openssl asn1parse' command, see appendix C for an example. Finally, the client private key and signed certificate were used to set up SSL connections to an OpenVPN[48] server. This last experiment was performed to assure that the certificates and keys produced by the system could be used with third-party software, and example results are presented in appendix D.

Having completed a functional system, the time had come to stress test it. An ordinary workstation with an AMD Athlon XP 1800 CPU was set up as the server, and another workstation, with a VIA Eden 500 MHz CPU, was configured to just read a PKIMessage from disk and then spawn 10 threads that fired requests at the server as quickly as possible. The result was that the Java runtime environment on the server ran out of heap memory, the

garbage collector was simply not able to remove dereferenced objects quickly enough. The end result was that the server was unable to respond to any more queries.

The next iteration of the development focused on the scalability of the server. Initial instantiation of shared objects in the Servlet were moved from the doGet() method to the init() method. After these adjustments, the server performed better under heavy loads, at least it did not run out of memory. To test the performance of the system using realistic server hardware, two Dell PowerEdge 2650 machines were used, one as server and the other as client. The server and client operating system was SUSE Linux Enterprise Server 9. This setup was used for all the subsequent performance tests. Hardware specifications for the Dell servers are listed in appendix P.

During the tests, several errors occurred. This was traced back to certain objects, particularly Cipher and SHA1Digest, being instantiated in the init() method, which means that they were shared among several Servlet instances. The problem was the way that the objects were used, each Servlet had to make several method calls in sequence to the object. This is illustrated by the example code from the server in table 5.7. As an example, if one Servlet calls the Cipher.init() method just before another Servlet calls the doFinal() method, the second Servlet will experience a "java.lang.IllegalStateException: Cipher not initialized", because the state of the myCipher object is different from what the second Servlet expected.

In the init() method:

```
Cipher myCipher =
  Cipher.getInstance("AES/CBC/PKCS7Padding" , bcProvider);
```

In the doGet() method:

```
myCipher.init(Cipher.ENCRYPT_MODE , myAESKeySpec , secureRandom);
byte[] myAESIV = myCipher.getIV();
byte[] encryptedValue = myCipher.doFinal(myX509Cert.getEncoded());
```

Table 5.7: State problems with the Cipher object

This problem might not be detected on a single-threaded system, but it was definitely noticeable on an SMP system with hyper threading, where several Servlets were executed simultaneously and interfered with each other. The instantiation of the objects in question was moved to the doGet() method, such that they were instantiated once per request, and this solved the problem. The lesson learned was that one must be very careful when deciding which objects should be instantiated in the init() method, and which objects

should be instantiated once per request. If a Servlet does more than one method call in sequence to an object while depending on the state of said object, it must instantiate its own object for every request.

When the system performed well under heavy load, the next step was to modularize it and to add support for the RSA signature algorithm. After this had been successfully implemented, the time had come to stress test the system thoroughly to gather performance statistics for different key lengths and algorithms. Such statistics are useful for future consideration of hardware needs for the AS. Before initiating stress tests with different configuration parameters, the client and server were modified to read their configuration parameters from file during run-time, to avoid having to recompile the whole system for each round of testing. An example AS configuration file is listed in appendix J, and the corresponding client configuration file in appendix K. The finished AS implementation is included as appendix H, and the client implementation as appendix I. An in-depth description of the testing methodology and the results is presented in section 5.6.

5.6 Performance testing

With functionality covered, the focus can now be shifted to performance. The test results and the following analysis presented in this section provide the information needed to select proper hardware based on expected system load. Additionally, a comparison between benefits and drawbacks of RSA and DSA is presented. In the description of the performance tests, the term "server" should be understood as the AS/CA of the proposed system, not as one of the other servers in figure 1.1.

For the performance testing, the test setup was the same as described in section 5.5, two Dell PowerEdge 2650 machines running SUSE Linux Enterprise Server 9. Details are listed in appendix P.

The performance tests used two components, the Java classes referenced in appendix M and the shell script referenced in appendix L. What the shell script does is to run the ordinary client once, with or without renewing the certificate, and then write the PKIMessage that was sent to the server to a file. Then, it starts forking Java processes, one per minute. The Java processes read the PKIMessage from the file and repeatedly send it to the server. The output is logged to a file. If the maximum number of parallel processes is reached, the script instead initiates a single request to check the response time of the server. The motivation for implementing it this way was to be able to observe the increasing load on the server as new stress test processes were started, and to make sure that new threads were started

regularly, as opposed to all of them starting, and possibly finishing, at once. An alternative approach might have been to just start everything at once, and to make the Java processes continue indefinitely instead of exiting after a specific number of requests.

One potential source of errors in this test setup is that if the same calculation is performed many times on the server, the parameters may be cached, resulting in a faster response than in a real-world scenario with several thousand clients sending different messages. To avoid this, care was taken to modify the messages for each request, so that the calculations would be different. For the initial authentication, the stresstest.java class generates a new SPEKE public key g^{R_C} every time, and the AS generates a new R_{CA} so that the server has to compute a new session key for every request. Additionally, the server increments the serial number of the issued certificate for each request, so that the hash of the client certificate is modified. Thus, the server signs a different value for each request. For the renewal tests, the client generates a new nonce for the PKIHeader for each request, then signs the message. The result is that the AS has to check a different signature for each request.[3] The only computationally intensive operation that could be cached in these tests is the verification of the CA signature on the client's old certificate.

By using the 'top' command on the server, one could observe that each client process was only able to use between 22 and 24% of the available CPU resources. This is probably due to the fact that the server has two CPUs, with each CPU being able to run two threads in parallel. With 5 processes, the CPU load went above 90%, and with 10 processes the client was very close to maximizing the CPU load on the server. The limit used for the number of parallel processes in the tests is therefore 10, to maximize server load.

Six different test cases were defined, these are listed in table 5.8. The cases listing SPEKE as the client algorithm are initial issuing of a certificate, while the others are renewal. For each of the cases, one test run was performed for each possible combination of the moduli 768, 1024 and 2048, for a total of nine test runs per test case. Each test run was allowed to execute for one hour after the test had reached the maximum number of parallel processes. A typical execution was to start the test run at e.g. 11:30. When the time approached 12:00, the load on the server was checked, to verify that it was close to maximum CPU usage. The test was left to run until 13:00, then

[3]In some of the test runs, where the client had to sign using RSA with a large modulus, the client was simply not able to generate enough load to stress test the server properly. In those test runs, marked with a dagger (†) in the test results, the client did not modify the message.

performance statistics were collected for the period 12:00-13:00.

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Client algorithm	SPEKE	SPEKE	DSA	RSA	RSA	DSA
CA algorithm	DSA	RSA	DSA	RSA	DSA	RSA

Table 5.8: Test cases

As soon as a test run was completed, the log on the client was checked for errors. The Java classes record the number of invalid responses from the server while they run, and when they exit a summary is written to the log file. If no errors had occurred, then the performance statistics were extracted from the access log on the server. The client's log file was thus used to verify that the server had returned only valid responses, while the access log on the server was used to measure the number of issued certificates. In the aforementioned example, assuming that the test was run on May 12, and that the client IP address was 129.241.11.162, the log lines of issued certificates were extracted with the following command from the Tomcat installation directory:

```
grep '12/May/2005:12' logs/localhost_access_log.2005-05-12.txt | \
grep ^129.241.11.162 > stresstest_renew_client1024rsa_ca1024dsa.txt
```

Further processing of the log file was then performed:

```
awk '{print $4}'< stresstest_renew_client1024rsa_ca1024dsa.txt | \
sort | uniq -c | awk '{print $1}' \
> stresstest_renew_client1024rsa_ca1024dsa_results.txt
```

What this command does is, for each second in the original log file, write a line that contains the number of requests answered during that second. The resulting file, with the number of issued certificates per second, should thus have 3600 lines. Unfortunately, Tomcat seems to buffer logging when the server is under heavy load. The result is that some seconds are missing, in the tests performed this number was somewhere between 25 and 180 lines, depending on the key lengths used. To correct this, new lines of 0 issued certificates were added to the file until the total number of lines reached 3600. The net result is that the average is correct, but the standard deviation is probably higher than it should have been. Finally, the file was loaded into a spreadsheet to perform statistical analysis.

After computing averages and standard deviation, the results were aggregated. Table 5.9 shows the number of issued certificates for the initial authentication for different modulus lengths and algorithms. Table 5.10 shows corresponding results for certificate renewal. The syntax of the table entries is $average \pm stdev$.

SPEKE modulus	CA modulus	DSA	RSA
2048	2048	20.0 ± 4.9	14.5 ± 4.1
2048	1024	23.4 ± 5.1	22.6 ± 5.1
2048	768	24.2 ± 5.4	24.0 ± 5.3
1024	2048	43.6 ± 6.7	24.3 ± 5.3
768	2048	53.5 ± 7.2	26.9 ± 5.5
1024	1024	65.3 ± 8.0	62.2 ± 7.5
1024	768	71.4 ± 8.3	72.1 ± 8.2
768	1024	89.8 ± 9.3	82.6 ± 8.7
768	768	101.5 ± 10.0	102.8 ± 9.9

Table 5.9: Initial authentication, issued certificates per second

A few interesting conclusions can be drawn by observing the results for initial issuing of certificates in table 5.9. Looking back to section 5.4, it was conjectured that the number of issued certificates per second would be highest if the CA used DSA as the signature algorithm. The experiment partly confirms this, but it has a significant impact only when the CA modulus is larger than the SPEKE modulus or if both moduli are 2048 bits long. The most obvious explanation for this is that the SPEKE computations are so computationally expensive that when the SPEKE modulus is the same or larger than the DSA or RSA modulus, the time used to sign the certificates becomes insignificant in comparison. One other interesting observation is that, for 2048 bit moduli, RSA seems to have approximately the same performance impact as SPEKE. This can be seen in the results on lines two to five. When either the SPEKE modulus or the RSA modulus is 2048, there is hardly any difference at all in the performance whether the other one is 1024 or 768 bits long. This could mean that RSA signing scales poorly to large moduli, the performance deteriorates faster than it does for SPEKE and DSA when the modulus length is increased. The results for modulus lengths 768/768, 1024/1024 and 2048/2048 also support this. The relative performance difference between RSA and DSA increases together with the modulus length. At 768/768 there is hardly any performance difference between using RSA or DSA (1%), at 1024/1024 the difference is noticeable (5%), and at 2048/2048 there is a significant difference (38%).

From the test results for certificate renewal in table 5.10, one can finally arrive at a conclusion with regards to the relative speeds of DSA versus RSA. One question not answered by the analysis in section 5.4 was: Exactly how much slower is RSA at signing, and how much faster is it at verifying

Client	CA	DSA/DSA	RSA/RSA	RSA/DSA	DSA/RSA
2048	2048	20.9 ± 5.0	31.0 ± 6.0†	34.0 ± 6.0†	19.8 ± 4.8
2048	1024	35.5 ± 6.1	151.5 ± 12.4†	105.1 ± 9.7†	39.7 ± 6.5
2048	768	39.4 ± 6.5	237.8 ± 16.9†	153.9 ± 12.1†	43.7 ± 6.8
1024	2048	29.6 ± 5.8	31.4 ± 5.9	35.1 ± 6.0	27.4 ± 5.6
1024	1024	70.9 ± 8.1	170.8 ± 13.1†	108.9 ± 11.0	88.3 ± 9.2
1024	768	90.4 ± 9.3	290.5 ± 17.9†	173.0 ± 13.0†	115.5 ± 10.7
768	2048	31.6 ± 5.9	31.9 ± 5.9	34.6 ± 6.2	28.9 ± 5.7
768	1024	83.0 ± 8.7	170.9 ± 14.7	114.0 ± 10.2	111.9 ± 10.3
768	768	112.4 ± 10.1	301.3 ± 18.5†	174.1 ± 14.5	151.9 ± 12.6

(RSA/DSA = Client signs using RSA, CA signs using DSA)
† = This test run did not generate a new signature for each request. The result is that these numbers might be somewhat higher than the performance in a real-world scenario because of caching on the server.

Table 5.10: Renewed certificates per second

when compared to DSA? The answer cannot be determined directly from the results, but recalling that for each renewal, the server has to sign once and verify once using its own signature algorithm, one can tell whether a signature and verification together using RSA are faster than when using DSA. To find the answer, first look at RSA/RSA versus RSA/DSA. Here, the client algorithm is the same, so the only performance difference should be the change of server algorithm. The results show that as long as the server modulus is less than 2048 bits long, RSA is a lot faster than DSA. In other words, it is faster to do one signature and one verification using RSA than when using DSA. For a 2048 bit server modulus, however, there is close to no difference between the performance of the two algorithms. Once again, the assumption that RSA signing scales poorly to large moduli is confirmed. Looking at DSA/DSA versus DSA/RSA, the results once again show that RSA is faster, except for a modulus length of 2048 bits.

The effect of the choice of client algorithm is obvious: The server has a higher performance when the client uses RSA. This is not a surprise, looking back to table 5.6, one can observe that the server performs one verification using the client algorithm and modulus. Since RSA is faster at verification, using that algorithm for client signatures increases server performance.

Another observation from the results is that a client using DSA with a 2048 bit modulus has a huge negative performance impact on the server. The results for 2048/768 and 2048/1024 illustrate this clearly, the difference

between 2048-bit RSA and 2048-bit DSA on the client is striking. This might be an indication that DSA signature verification has the same problem as RSA signing with regards to scaling to large moduli.

The most severe errors from the conjectures in section 5.4 were that the time used for RSA signing was over-estimated, and the impact of DSA signature verification was under-estimated. This resulted in the assumption that DSA on the server with RSA on the client would be the fastest system. The test results clearly show that this is not the case. The highest performance is obtained by using RSA on both the client and server, as long as the server modulus is less than 2048 bits long.

With the review of server performance completed, it is time to take a closer look at the client. As mentioned in section 5.4, key generation times on the client should be tested. To simulate a medium to low end client PC, a Dell Latitude X200 laptop with an Intel Pentium 3 M 800 MHz CPU was used for key generation. The Java class in appendix N was used to test key generation times. The test was run 15 times for each modulus length, with 100 iterations for each run, for a total of 1500 key generations for each combination of algorithm and modulus. The whole test took approximately nine hours.

Modulus	DSA avg	DSA max	RSA avg	RSA max
768	13 ± 2	32	907 ± 418	3962
1024	22 ± 2	35	1819 ± 923	6518
2048	80 ± 3	128	16915 ± 10213	69238

Table 5.11: Key generation times in milliseconds

Looking at the results in table 5.11, one can see that the assumption made in section 5.4 was correct: Generating RSA keys takes significantly longer than generating DSA keys. The key generation times for DSA are so short, irrespective of the modulus length, that they have close to no impact on the system performance. With a modulus length of 2048, the worst case was just above 1/8 of a second. For RSA, however, key generation times is a real issue. The time needed to generate a key with 1024 bit modulus is borderline, the worst case of more than six seconds is a long time for a user to wait for the login. With a modulus length of 2048 bits, the performance of RSA is simply not acceptable, with an average of nearly 17 seconds and a worst case of over a minute. The conclusion is that for initial login, DSA seems to be the best choice for the client. Using RSA incurs an extreme performance penalty for key generation. To answer the first of the two questions at the end of

section 5.4: Generating RSA keys at login time might be a viable approach, but not when using a modulus length of more than 1024 bits.

The reason why DSA key generation is so much faster than RSA is, as mentioned in section 5.4, that the key generation parameters p, q and g are not generated on-the-fly, but loaded from a file. This kind of pre-computation is not possible with RSA. However, it would also be interesting to see how long generation of these DSA parameters takes on the client system described above. To this end, the Java class in appendix O was used. The same Dell X200 laptop was used, and 10 tests of 10 parameter generations per modulus was performed, for a total of 100 key parameter generations for each modulus. The entire test run took approximately 14 hours, and the results are presented in table 5.12

Modulus	average±stdev	max
768	17019 ± 18213	95651
1024	53777 ± 51132	211277
2048	541369 ± 540657	2893329

Table 5.12: DSA parameter generation times in milliseconds

From the results listed in table 5.12, it is obvious that DSA parameter generation on-the-fly at login time is not practical. The huge standard deviation illustrates that the generation sometimes is very fast, while other times extremely slow. The worst case, even for a modulus length of 768 bits, is over one and a half minute, far more than an acceptable delay at login time. The conclusion is that the DSA parameters should be generated when the client application is installed, and stored on the client. The answer to the last question in section 5.4 is thus: No, it is definitely not fast enough.

In summary, from a server performance perspective, the CA and clients should in most cases use RSA as the signature algorithm. The only exception is when the system load consists mostly of initial authentication with a modulus of 2048 bits length. For renewal of certificates, RSA generally outperforms DSA, especially when using modulus lengths of less than 2048 bits. The problem with using RSA on the client is that the key generation is time-consuming, which may result in too long delays when users log in. One possible solution to this problem is to use DSA for the initial client certificate, then use RSA keys when renewing the certificate. The RSA keys used for new certificates could be generated as a background process, so that the key generation delay would not be noticeable to the user. This means that the first renewal will have message protection using DSA, while all subsequent

messages will use RSA, which makes sense for a system where users frequently renew their certificates. On the other hand, if one wishes to use only one signature algorithm for everything, the best choice is probably DSA, due to the short key generation times on the client and acceptable performance on the server side, especially when one considers the fact that the AS can scale horizontally by adding more servers and a load balancer.

It is difficult to provide a good estimate for the load generated by e.g. ten thousand users of such a system without observing an actual production system, but a short discussion of this is in order. If one assumes that all of those ten thousand users log on at the start of a work day, distributed evenly from 08:00 to 08:30, the system load would be $10000/(60 * 30)$, or approximately 5.6, issued certificates per second. If they all log on between 08:00 and 08:05, the load would be approximately 33 issued certificates per second. As seen in table 5.9, if we use modulus lengths of 1024 bits, one single Dell PE 2650 server is able to handle almost twice that load, which translates to nearly twenty thousand users in this scenario. For renewal of certificates, the performance is even better, and as mentioned previously, the AS can be trivially scaled horizontally by using HTTP load balancers. The conclusion is that performance-wise, the implemented system fulfills the specified requirements.

Chapter 6

Conclusions

The work described in this thesis has resulted in a functional and scalable implementation, but there is a lot more work to do, both research and system development, before it is ready to be used as a production system. This chapter gives suggestions for the road ahead, and repeats the key points learned during the project work.

6.1 Key learning points

The key points that can be extracted from the design, analysis, implementation and testing of the system are the following: The solution works, it protects user credentials when ordinary servers are compromised, it protects against network attacks, and it provides strong authentication with single sign-on using a PKI.

However, as the security analysis in section 5.3 shows, the proposed system design creates new vulnerabilities, particularly the active attack illustrated in figure 5.3. Furthermore, in password-only mode, the AS does not act as a CA, because it does not verify the link between the identity and the public key. The AS may therefore sign an invalid certificate. The end result is that some of the responsibilities of the CA is shifted to the client, because the client has to validate the contents of its own certificate. The client should always do this nevertheless, but with a proper CA, failure to do so will not enable an attacker to impersonate the user.

The key learning point from the performance analysis and testing is that for a typical installation using 1024-bit modulus lengths for all participants, the system implementation is definitely practical. First of all, the one server used in the tests is able to handle close to twenty thousand users, and second, it can be scaled horizontally. When choosing a public-key algorithm,

using RSA in place of DSA will generally result in better performance on the AS. However, key generation times for RSA keys are a potential problem on typical client hardware, while DSA key generation times are not. DSA *parameter* generation is so time-consuming that it is not practical to do this on-the-fly. If the AS performance is a problem, so that RSA should be used as the client signature algorithm, one might use a hybrid solution where the clients use DSA for their initial authentication, and RSA for all subsequent certification requests. This results in a "best of both worlds" scenario: Fast initial key generation on the client and a higher performance on the AS.

6.2 Further work

The road ahead can be summarized as five tasks or areas where more work is needed.

First of all, the client must be developed further. Implementing the client as a PKCS#11 software token, as KX.509 does, might be a good idea. Furthermore, extensive testing with existing applications should be carried out, to verify that those applications are able to use certificates generated in real-time.

Second, support for hardware token based authentication should be implemented and tested. The suggestion presented here is to use the message protection mechanism in CMP for this purpose. If this approach is used, token based authentication will look like a certificate renewal request to the AS, and the client should only need minor modifications. The AS, however, will need a mechanism for specifying trust levels, either by including it as a data field in the issued certificate, or by using different CA root certificates for different trust levels.

Third, more realistic performance tests should be carried out, preferably with several hundred, or even thousand, clients. As part of this work, horizontal scaling using HTTP load balancers could also be implemented and tested.

Fourth, a user database or directory should be developed, and the server implementation modified so that it performs its verifier lookups by querying the database. LDAP might be a viable option for the protocol used to fetch user verifiers. Mechanisms for throttling or limiting initial authentication should also be designed, to avoid the security vulnerability where an attacker can perform unaudited on-line password guesses.

Fifth, an alternative password-only initial authentication using four messages could be designed, to make the AS act as a proper CA in password-only mode. This will probably violate the assumption that the AS should be state-

less, which creates new challenges for the scalability of the system.

Finally, standardization work could be carried out. One possible approach is to submit an experimental RFC to the IETF. Then, one might consider further work to write a standards track RFC that formally specifies the system. To be able to standardize the scheme as an IETF RFC, one might have to use a different BPKAS than SPEKE due to patent issues.

6.3 Summary

As a final summary, the system performs well enough to be used in organizations with tens of thousands of users. The implementation provides a simple mechanism for bootstrapping a PKI using password-only authentication. However, the provided PKI can be used only for authentication, or possibly digital signatures, due to its on-the-fly generation of keys and certificates. To be able to encrypt a message to a user, one has to know that user's public key in advance, and this is not possible with the suggested system design. Still, being able to provide a cheap, scalable PKI solution that can be used for authentication is a goal in itself. As a matter of fact, that was the specific task given in the assignment that resulted in this thesis.

Bibliography

[1] C. Adams, S. Farrell. *Internet X.509 Public Key Infrastructure Certificate Management Protocols.* IETF RFC 2510.
Online: http://www.ietf.org/rfc/rfc2510.txt (Last checked 2005-05-16)

[2] C. Allen, T. Dierks. *The TLS Protocol Version 1.0.* IETF RFC 2246.
Online: http://www.ietf.org/rfc/rfc2246.txt (Last checked 2005-04-27)

[3] A. Arsenault, S. Farrell. *Securely Available Credentials - Requirements.* IETF RFC 3157.
Online: http://www.ietf.org/rfc/rfc3157.txt (Last checked 2005-05-15)

[4] Steven M. Bellovin, Michael Merritt. *Augmented Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks and Password File Compromise.*
Online: http://www.alw.nih.gov/Security/FIRST/papers/crypto/aeke.ps
(Last checked: 2005-02-28)

[5] Steven M. Bellovin, Michael Merritt. *Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks.* Proceedings: IEEE Symposium on Research in Security and Privacy, 1992.
Online: http://www.alw.nih.gov/Security/FIRST/papers/crypto/neke.ps
(Last checked: 2005-03-14)

[6] Michael Burrows, Martin Abadi, Roger Needham. *A Logic of Authentication*
Online: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-39.pdf
(Last checked: 2005-05-08)

[7] Whitfield Diffie, Martin E. Hellman. *New Directions in Cryptography.* Stanford University, 1976.
Online: http://crypto.csail.mit.edu/classes/6.857/papers/diffie-hellman.pdf
(Last checked: 2005-05-16)

[8] William Doster, Marcus Watts, Dan Hyde. *The KX.509 Protocol.* Online: http://www.citi.umich.edu/techreports/reports/citi-tr-01-2.pdf

[9] Martin Eian. *Public key infrastructure in large scale access control.* Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, 2004.

[10] S. Farrell, Ed. *Securely Available Credentials Protocol.* IETF RFC 3767. Online: http://www.ietf.org/rfc/rfc3767.txt (Last checked 2005-05-15)

[11] Dieter Gollmann. *Computer Security.* John Wiley & Sons, Ltd., 1999. ISBN 0-471-97844-2.

[12] D. Gustafson, M. Just, M. Nystrom. *Securely Available Credentials (SACRED) - Credential Server Framework.* IETF RFC 3760. Online: http://www.ietf.org/rfc/rfc3760.txt (Last checked 2005-05-15)

[13] D. Harkins, D. Carrel. *The Internet Key Exchange (IKE).* IETF RFC 2409. Online: http://www.ietf.org/rfc/rfc2409.txt (Last checked 2005-04-27)

[14] R. Housley, W. Ford, W. Polk, D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile.* IETF RFC 2459. Online: http://www.ietf.org/rfc/rfc2459.txt (Last checked 2005-05-25)

[15] ITU-T. *Recommendation X.509. Information Technology - Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks.* International Telecommunication Union, 2000.

[16] David P. Jablon. *Extended Password Key Exchange Protocols Immune to Dictionary Attack.* Online: http://www.jablon.org/jab97.pdf (Last checked: 2005-03-14)

[17] David P. Jablon. *Research Papers on Password-based Cryptography.* Online: http://www.jablon.org/passwordlinks.html (Last checked: 2005-05-15)

[18] David P. Jablon. *Strong Password-Only Authenticated Key Exchange.* Online: http://www.jablon.org/jab96.pdf (Last checked: 2005-03-14)

[19] Charlie Kaufman, Radia Perlman. *PDM: A New Strong Password-Based Protocol.* Proceedings: 10^{th} USENIX Security Symposium, USENIX Association, 1997.
Online:

http://www.usenix.org/events/sec01/full_papers/kaufman/kaufman.pdf (Last checked: 2005-04-14)

[20] J. Kohl, C. Neuman. *The Evolution of the Kerberos Authentication Service*
Online: ftp://athena-dist.mit.edu/pub/kerberos/doc/krb_evol.PS (Last checked: 2005-05-08)

[21] J. Kohl, C. Neuman. *The Kerberos Network Authentication Service (V5).* IETF RFC 1510.
Online: http://www.ietf.org/rfc/rfc1510.txt (Last checked: 2005-04-27)

[22] Olga Kornievskaia, Peter Honeyman, Bill Doster, Kevin Coffman. *Kerberized Credential Translation: A Solution to Web Access Control.*
Online: http://www.citi.umich.edu/techreports/reports/citi-tr-01-5.pdf

[23] Taekyoung Kwon. *Authentication and Key Agreement Via Memorable Password.* Proceedings: Network and Distributed System Security Symposium 2001, Internet Society.
Online: http://www.isoc.org/isoc/conferences/ndss/01/2001/papers/kwon.pdf (Last checked: 2005-05-16)

[24] Rich Lee, Jay E. Israel. *Understanding the Role of Identification and Authentication in NetWare 4.* Novell, Inc., 1994.
Online: http://developer.novell.com/research/appnotes/1994/october/02/ (Last checked: 2005-05-15)

[25] M. Myers, C. Adams, D. Solo, D. Kemp. *Internet X.509 Certificate Request Message Format.* IETF RFC 2511.
Online: http://www.ietf.org/rfc/rfc2511.txt (Last checked 2005-05-16)

[26] M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP.* IETF RFC 2560.
Online: http://www.ietf.org/rfc/rfc2560.txt (Last checked 2005-05-16)

[27] Alfred J. Menenez, Paul C. van Oorschot, Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997. ISBN 0-8493-8523-7.
Online: http://www.cacr.math.uwaterloo.ca/hac (Last checked: 2005-05-17)

[28] Paul C. van Oorschot, Michael J. Wiener. *On Diffie-Hellman Key Agreement with Short Exponents.* Proceedings: EUROCRYPT '96.

Online: http://www3.sympatico.ca/wienerfamily/Michael/MichaelPapers/dhshortexp.pdf (Last checked: 2005-05-23)

[29] Radia Perlman, Charlie Kaufman. *Secure Password-Based Protocol for Downloading a Private Key.* Proceedings: 1999 Network and Distributed System Security Symposium, Internet Society.
Online: http://www.isoc.org/isoc/conferences/ndss/99/proceedings/papers/perlman.pdf (Last checked: 2005-02-13)

[30] R. Rivest. *The MD5 Message-Digest Algorithm.* IETF RFC 1321.
Online: http://www.ietf.org/rfc/rfc1321.txt (Last checked: 2005-05-29)

[31] Bruce Schneier. *Applied Cryptography, Second Edition, Protocols, Algorithms and Source Code in C.* John Wiley & Sons, Inc., 1996. ISBN 0-471-12845-7.

[32] Joseph J. Tardo, Kannan Alagappan. *SPX: Global Authentication Using Public Key Certificates.* Proceedings : IEEE Symposium on Security and Privacy, 2001.
Online: http://ftp.digital.com/pub/Digital/SPX/SPX.v2.0-doc.tar.Z (Last checked: 2005-01-25)

[33] B. Tung, L. Zhu. *Public Key Cryptography for Initial Authentication in Kerberos.* IETF Internet Draft.
Online: http://www.ietf.org/internet-drafts/draft-ietf-cat-kerberos-pk-init-25.txt (Last checked: 2005-05-09)

[34] M. Wahl, T. Howes, S. Kille. *Lightweight Directory Access Protocol (v3).* IETF RFC 2251.
Online: http://www.ietf.org/rfc/rfc2251.txt (Last checked: 2005-05-16)

[35] Joel N. Weber II. *Kerberos Initial Authentication Methods.* IETF Internet Draft.
Online: http://mirrors.isc.org/pub/www.watersprings.org/pub/id/draft-weber-krb-wg-kerberos-initial-authentication-00.txt (Last checked: 2005-05-10)

[36] Thomas Wu. *The Secure Remote Password Protocol.* Stanford University, 1997.
Online: ftp://srp.stanford.edu/pub/srp/srp.ps (Last checked: 2005-02-08)

[37] Thomas Wu. *SRP-6: Improvements and Refinements to the Secure Remote Password Protocol.* Arcot Systems, 2002.
Online: http://srp.stanford.edu/srp6.ps (Last checked: 2005-02-08)

[38] Muxiang Zhang. *Analysis of the SPEKE Password-Authenticated Key Exchange Protocol.* IEEE Communication Letters, Vol. 8, No. 1, January 2004.
Online:
http://ieeexplore.ieee.org/iel5/4234/28208/01261928.pdf?isNumber=28208&prod=JNL&arnumber=1261928&arSt=+63&ared=+65&arAuthor=Muxiang+Zhang
(Last checked: (2005-05-17)

[39] *Advanced Encryption Standard (AES).* FIPS Publication 197, National Institute of Standards and Technology, 2001.
Online: http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf
(Last checked: 2005-05-28)

[40] *The Bouncy Castle Crypto APIs.* Legion of the Bouncy Castle.
Online: http://www.bouncycastle.org/ (Last checked: 2005-05-22)

[41] *Digital Signature Standard (DSS).* FIPS Publication 186-2, National Institute of Standards and Technology, 2000.
Online: http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf
(Last checked: 2005-05-28)

[42] *Java 2 Platform Standard Edition 5.0.* Sun Microsystems Inc.
Online: http://java.sun.com/j2se/1.5.0/index.jsp (Last checked: 2005-05-22)

[43] *Kerberos: The Network Authentication Protocol.*
Online: http://web.mit.edu/kerberos/www/ (Last checked 2005-05-08)

[44] *MIT Kerberos Distribution Page.*
Online: http://web.mit.edu/kerberos/www/dist/index.html (Last checked 2005-05-08)

[45] *The Heimdal Kerberos 5 implementation.*
Online: http://www.pdc.kth.se/heimdal/ (Last checked 2005-05-08)

[46] *John the Ripper password cracker.* Openwall Project.
Online: http://www.openwall.com/john/ (Last checked: 2005-05-16)

[47] *NOVOSEC Bouncy Castle Extensions.* NOVOSEC AG.
Online: http://novosec-bc-ext.sourceforge.net/ (Last checked: 2005-05-22)

[48] *OpenVPN - An Open Source SSL VPN Solution.* OpenVPN Solutions LLC.
Online: http://openvpn.net/ (Last checked: 2005-05-31)

[49] *P1363.2: Standard Specifications for Password-Based Public-Key Cryptographic Techniques, version D20 (draft).*
Online: http://grouper.ieee.org/groups/1363/passwdPK/draft.html (Last checked: 2005-05-15)

[50] *PKCS#1 v2.1: RSA Cryptography Standard.* RSA Laboratories, 2002.
Online: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf (Last checked: 2005-05-28)

[51] *PKCS#11 v2.20: Cryptographic Token Interface Standard.* RSA Laboratories, 2004.
Online: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf (Last checked: 2005-05-09)

[52] *A Secure European System for Applications in a Multi-vendor Environment.*
Online: https://www.cosic.esat.kuleuven.ac.be/sesame/ (Last checked 2005-05-08)

[53] *Secure Hash Standard (SHS).* FIPS Publication 180-2, National Institute of Standards and Technology, 2002.
Online: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf (Last checked: 2005-05-29)

[54] *Standard ECMA-219. Authentication and Privilege Attribute Security Application with related Key Distribution Functions - Part 1, 2 and 3, 2nd edition.* ECMA, 1996.
Online:
http://www.ecma-international.org/publications/files/ecma-st/ECMA-219.pdf (Last checked 2005-05-12)

[55] *Apache Jakarta Tomcat.* The Apache Software Foundation.
Online: http://jakarta.apache.org/tomcat/index.html (Last checked: 2005-05-22)

[56] *Windows 2000 Kerberos Authentication.* Microsoft Corporation.
Online:
http://www.microsoft.com/windows2000/techinfo/howitworks/security/kerberos.asp (Last checked: 2005-05-08)

[57] *Kerberos Authentication in Windows Server 2003.* Microsoft Corporation.
Online:
http://www.microsoft.com/windowsserver2003/technologies/security/kerberos/
(Last checked: 2005-05-08)

Appendix A

Tomcat configuration

First, Tomcat 5.5.9 was downloaded and unpacked. Then, a new directory 'sso' was created in the 'webapps' subdirectory. All directory references in this appendix are relative to the Tomcat installation directory. The server was packed as a jar-file, diplom.jar, and deployed as a web application together with the required Bouncy Castle libraries in the webapps/sso/WEB-INF/lib directory. Directory structure of the webapps/sso directory:

```
eian@sigilion:~/jakarta-tomcat-5.5.9$ ls -lR webapps/sso/
webapps/sso/:
total 12
drwxr-xr-x  2 eian eian 4096 Apr 13 18:24 META-INF
drwxr-xr-x  3 eian eian 4096 May  1 19:37 WEB-INF
drwxr-xr-x  2 eian eian 4096 Mar 26 20:22 images

webapps/sso/META-INF:
total 4
-rw-r--r--  1 eian eian 327 Mar 26 20:22 context.xml

webapps/sso/WEB-INF:
total 8
drwxr-xr-x  2 eian eian 4096 May 19 20:33 lib
-rw-r--r--  1 eian eian  767 May  1 19:37 web.xml

webapps/sso/WEB-INF/lib:
total 2044
-rw-r--r--  1 eian eian  115303 Apr 13 18:45 bcmail-jdk15-127.jar
-rw-r--r--  1 eian eian  231518 Apr 13 18:45 bcpg-jdk15-127.jar
-rw-r--r--  1 eian eian 1066809 Apr 13 18:45 bcprov-jdk15-127.jar
-rw-r--r--  1 eian eian  462670 Apr 13 18:45 bctest-jdk15-127.jar
-rw-r--r--  1 eian eian   36386 Apr 13 18:45 bctsp-jdk15-127.jar
-rw-r--r--  1 eian eian  114016 May 24 11:47 diplom.jar

webapps/sso/images:
total 0
```

Contents of the web.xml file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

  <display-name>Single Sign-On</display-name>
  <description>
    Single Sign-On
  </description>

  <!-- SSO definition -->
  <servlet>
    <servlet-name>server</servlet-name>
    <servlet-class>no.ntnu.item.ttm4900.server</servlet-class>
    <load-on-startup>10</load-on-startup>
  </servlet>

  <!-- SSO mapping -->
  <servlet-mapping>
    <servlet-name>server</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Relevant section of the conf/server.xml configuration file:

```
    <Connector port="8080" maxHttpHeaderSize="8192"
               maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
               enableLookups="false" redirectPort="8443" acceptCount="100"
               connectionTimeout="20000" disableUploadTimeout="true" />
```

Lines added to the beginning of bin/startup.sh and bin/shutdown.sh:

```
JAVA_HOME=/usr/local/jdk1.5.0_02
export JAVA_HOME
```

Appendix B

Debug output from server and client

Example debug and timing output for initial authentication and two renewals. The parameters are 1024 bit modulus for SPEKE, client 1024 bit RSA signatures and CA 1024 bit DSA signatures. The hardware used is a Dell Latitude X200 laptop with a Pentium 3 M 800 MHz CPU as both server and client.

Client output:

```
H(P) = 653878565946713713149629104275478104571867727804
H(P)^2 = 427557179004530834336469072637328194198138589757732486381
698282038165706347500705335709834662416
Generating RSA key pair...OK
RSA key generation took 1982 milliseconds.
H(P)^(2RA) = 480429431665629502691088914691546308245911203206999981
927023647747298578722341172282882221575811488963581437718930971387
779250205671568555796390090430630164655587047476464435468885403519
730566843973927994410760863593672637367600866409749692729507065278
712619547408558146372884939016884695082357646804094397 40
H(P)^(2RB) = 918181733678821525592461650199626673076425832199076598
947392492002043862746375608956611121868847367695351607363182316 91
235711725976865017866377219631237070183824553408393316364434124440
990933394427775570395323320252475479983628768152972752835572703277
886228604321684535512794612384773635188441741621802433097
H(P)^(4RA*RB) = 103320580060848361586683658316135569168705700809 72
255144259163261746061587393266633567468641948418802038011680000427
418594720124895948341658560922916123391306730929537832214774580028
848287585986366176183116793298636040152945005476797320090681459738
141548511464236830750037669051454366482704602675565154454281 9
Using encrypted CA root certificate from Certification Response.
Checking DSA signature...
CA self-signature verified!
Checking DSA signature...
CA signature verified!
```

```
Certificate contains correct RSA public key and name.
Renewing certificate...
Generating RSA key pair...OK
RSA key generation took 2279 milliseconds.
Checking DSA signature...
CA signature verified!
Certificate contains correct RSA public key and name.
Renewing certificate...
Generating RSA key pair...OK
RSA key generation took 685 milliseconds.
Checking DSA signature...
CA signature verified!
Certificate contains correct RSA public key and name.
```

Server output (from logs/catalina.out in the tomcat installation directory):

```
CONTENT-TYPE application/pkixcmp
Subject name: DC=no,DC=ntnu,OU=people,UID=eian
OID: 0.9.2342.19200300.100.1.1
OIDValue: eian
User name: eian
Verifier: 65387856594671371314962910427547810457186772780 4
Signing client certificate took 29 milliseconds.
RB =  91226974228730798024415015595840139577336335363331397668623 0
22406115558368282911171659741677810731 07
Generation of RB took 1 milliseconds.
H(P)^(2RA) = 48042943166562950269108891469154630824591120320699981
92702364774729857872234117228288222157581148896358143771893097138 7
77925020567156855579639009043063016465558704747646443546888540351 9
73056684397392799441076086359367263736760086640974969272950706527 8
71261954740855814637288493901688469508235764680409439740
H(P)^(2RB) = 91818173367882152559246165019962667307642583219907659
89473924920020438627463756089566111218688473676953516073631823169 1
23571172597686501786637721963123707018382455340839331636443412444 0
99093339442777557039532332025247547998362876815297275283557270327 7
88622860432168453551279461238477363518844174162180243309 7
H(P)^(4RA*RB) = 10332058006084836158668365831613556916870570080972
25514425916326174606158739326663356746864194841880203801168000042 7
41859472012489594834165856092291612339130673092953783221477458002 8
84828758598636617618311679329863604015294500547679732009068145973 8
14154851146423683075003766905145436648270460267556515445428 19
SPEKE key calculation took 81 milliseconds.
AES encryption of client certificate took 8 milliseconds.
AES encryption of CA certificate took 14 milliseconds.
CONTENT-TYPE application/pkixcmp
Client signature verified!
Verification of client signature took 8 milliseconds.
CA signature verified!
Verification of CA signature took 42 milliseconds.
Subject name: DC=no,DC=ntnu,OU=people,UID=eian
```

```
OID: 0.9.2342.19200300.100.1.1
OIDValue: eian
User name: eian
Verifier: 653878565946713713149629104275478104571867727804
Signing client certificate took 29 milliseconds.
CONTENT-TYPE application/pkixcmp
Client signature verified!
Verification of client signature took 2 milliseconds.
CA signature verified!
Verification of CA signature took 46 milliseconds.
Subject name: DC=no,DC=ntnu,OU=people,UID=eian
OID: 0.9.2342.19200300.100.1.1
OIDValue: eian
User name: eian
Verifier: 653878565946713713149629104275478104571867727804
Signing client certificate took 26 milliseconds.
```

Example debug and timing output for initial authentication and two renewals. The parameters are 2048 bit modulus for SPEKE, client 768 bit DSA signatures and CA 2048 bit RSA signatures. The hardware used is a Dell Latitude X200 laptop with a Pentium 3 M 800 MHz CPU as both server and client.

Client output:

```
H(P) = 653878565946713713149629104275478104571867727804
H(P)^2 = 427557179004530834336469072637328194198138589757732486381
698282038165706347500705335709834662416
Generating DSA key pair...OK
DSA key generation took 21 milliseconds.
H(P)^(2RA) = 849598844694394670389692080726068046152008243882438283
784062711788956887488537253671192233058765849148634338480015795175
928191839034181949177815229567329383236527025596960915048327439924
661007099992961265483500553448745704928885143874094301832335719033
960146518624355883610318698412554719159092419164699454524849532865
144787576548433439693287317651463577231624599723557672998340987949
298393484431921048561101933924937168484626767860651251537447083623
539587207350950476760084955641357476245874195809730706355272562335
748635063519742889859106873489205531057688498034680049713060664851
155354765783552241165380666566369 89
H(P)^(2RB) = 160937186316207326572543529026016702163479317065014572
605530861691742389181108492085235460215689735958819580686741932327
473183964787579413203194083345524240640909368068382829996975681474
970761295456641645646521418396581411273239979243612619809949518977
770609135989999077891083635664153989096988012672736287292976776929
952026720116470828716077027647330532039132831013940824498857391804
527500708305770795120581626877221973184258457266254448750266711735
037924115093328931462396919548580047720781162598218207318376677372
117142672677857650206418212982914568344841708209884328208311189536
432674932007390019575510264599434454
H(P)^(4RA*RB) = 211806570272437906761774053914906886844296769499 12
```

```
3028383908953085210949445689708861105239131463346332223268124342146258762461858397082009232952816836564491559106846141755766644225559432641968976926292459945384072921988713236381173928237355780549102890232356821267615488155401231900310068687723936043773685854662007794059421325177779859979892916664897903835366524893227208368710690497518018036880555074700631731060094234506926998902793591266886349539394918649028346680052404063202688366077356512199011782705730489757415406948042970862444585973378290792558394094948593585144475091670186000663451915846365147339133367 25
Using encrypted CA root certificate from Certification Response.
Checking RSA signature...
CA self-signature verified!
Checking RSA signature...
CA signature verified!
Certificate contains correct DSA public key and name.
Renewing certificate...
Generating key pair...OK
Key generation took 12 milliseconds.
Checking RSA signature...
CA signature verified!
Certificate contains correct DSA public key and name.
Renewing certificate...
Generating key pair...OK
Key generation took 12 milliseconds.
Checking RSA signature...
CA signature verified!
Certificate contains correct DSA public key and name.
```

Server output (from logs/catalina.out in the tomcat installation directory):

```
CONTENT-TYPE application/pkixcmp
Subject name: DC=no,DC=ntnu,OU=people,UID=eian
OID: 0.9.2342.19200300.100.1.1
OIDValue: eian
User name: eian
Verifier: 65387856594671371314962910427547810457186772780 4
Signing client certificate took 271 milliseconds.
RB =  867841658393189978691184481451209358898304680510995470895876872708257513076072490709460221280073516 9
Generation of RB took 0 milliseconds.
H(P)^(2RA) = 849598844694394670389692080726068046152008243882438287840627117889568874885372536711922330587658491486343384800157951759281918390341819491778152295673293832365270255969609150483274399246610070999929612654835005534487457049288851438740943018323357190339601465186243558836103186984125547191590924191646994545248495328651447875765484334396932873176514635772316245997235576729983409879492983934844319210485611019339249371684846267678606512515374470836235395872073509504767600849556413574762458741958097307063552725623357486350635197428898591068734892055310576884980346800497130606648511553547657835522411653806665663698 9
```

```
H(P)^(2RB) = 160937186316207326572543529026016702163479317065014572605530861691742389181108492085235460215689735958819580687419323274731839647875794132031940833455242406409093680683828299969756814749707612954566416456465214183965814112732399792436126198099495189777706091359899990778910836356641539890969880126727362872929767769299520267201164708287160770276473305320391328310139408244988573918045275007083057707951205816268772219731842584572662544487502667117350379241150933289314623969195485800477207811625982182073183766773721171426726778576502064182129829145683448417082098843282083111895364326749320073900195755102645994344543
H(P)^(4RA*RB) = 211806570272437906761774053914906886844296769499123028383908953085210949445689708861105239131463346332223268124342146258762461858397082009232952816836564491559106846141755766644225559432641968976926292459945384072921988713236381173928237355780549102890232356821267615488155401231900310068687723936043773685854662007794059421325177779859979892916664897903835366524893227208368710690497518018036880555074700631731060094234506926998902793591266886349539394918649028346680052404063202688366077356512199011782705730489757415406948042970862444585973378290792558394094948593585144475091670186000663451915846365147339133367255183600066345191584636514733913336725
SPEKE key calculation took 334 milliseconds.
AES encryption of client certificate took 4 milliseconds.
AES encryption of CA certificate took 2 milliseconds.
CONTENT-TYPE application/pkixcmp
Client signature verified!
Verification of client signature took 27 milliseconds.
CA signature verified!
Verification of CA signature took 20 milliseconds.
Subject name: DC=no,DC=ntnu,OU=people,UID=eian
OID: 0.9.2342.19200300.100.1.1
OIDValue: eian
User name: eian
Verifier: 653878565946713713149629104275478104571867727804
Signing client certificate took 263 milliseconds.
CONTENT-TYPE application/pkixcmp
Client signature verified!
Verification of client signature took 28 milliseconds.
CA signature verified!
Verification of CA signature took 16 milliseconds.
Subject name: DC=no,DC=ntnu,OU=people,UID=eian
OID: 0.9.2342.19200300.100.1.1
OIDValue: eian
User name: eian
Verifier: 653878565946713713149629104275478104571867727804
Signing client certificate took 267 milliseconds.
```

Appendix C

OpenSSL asn1parse of a PKIMessage

This is an example of a PKIMessage, a Certification Request using an RSA signature for message protection. The old certificate included in the request is signed with a DSA signature from the CA.

```
eian@sigilion:~/fag/diplom/sw$ openssl asn1parse -inform der <certreq.der
    0:d=0  hl=4 l=1633 cons: SEQUENCE
    4:d=1  hl=3 l= 196 cons: SEQUENCE
    7:d=2  hl=2 l=   1 prim: INTEGER           :01
   10:d=2  hl=2 l=  76 cons: cont [ 4 ]
   12:d=3  hl=2 l=  74 cons: SEQUENCE
   14:d=4  hl=2 l=  18 cons: SET
   16:d=5  hl=2 l=  16 cons: SEQUENCE
   18:d=6  hl=2 l=  10 prim: OBJECT            :domainComponent
   30:d=6  hl=2 l=   2 prim: PRINTABLESTRING   :no
   34:d=4  hl=2 l=  20 cons: SET
   36:d=5  hl=2 l=  18 cons: SEQUENCE
   38:d=6  hl=2 l=  10 prim: OBJECT            :domainComponent
   50:d=6  hl=2 l=   4 prim: PRINTABLESTRING   :ntnu
   56:d=4  hl=2 l=  15 cons: SET
   58:d=5  hl=2 l=  13 cons: SEQUENCE
   60:d=6  hl=2 l=   3 prim: OBJECT            :organizationalUnitName
   65:d=6  hl=2 l=   6 prim: PRINTABLESTRING   :people
   73:d=4  hl=2 l=  13 cons: SET
   75:d=5  hl=2 l=  11 cons: SEQUENCE
   77:d=6  hl=2 l=   3 prim: OBJECT            :commonName
   82:d=6  hl=2 l=   4 prim: PRINTABLESTRING   :eian
   88:d=2  hl=2 l=  57 cons: cont [ 4 ]
   90:d=3  hl=2 l=  55 cons: SEQUENCE
   92:d=4  hl=2 l=  18 cons: SET
   94:d=5  hl=2 l=  16 cons: SEQUENCE
   96:d=6  hl=2 l=  10 prim: OBJECT            :domainComponent
  108:d=6  hl=2 l=   2 prim: PRINTABLESTRING   :no
```

```
  112:d=4  hl=2 l=  20 cons: SET
  114:d=5  hl=2 l=  18 cons: SEQUENCE
  116:d=6  hl=2 l=  10 prim: OBJECT            :domainComponent
  128:d=6  hl=2 l=   4 prim: PRINTABLESTRING   :ntnu
  134:d=4  hl=2 l=  11 cons: SET
  136:d=5  hl=2 l=   9 cons: SEQUENCE
  138:d=6  hl=2 l=   3 prim: OBJECT            :commonName
  143:d=6  hl=2 l=   2 prim: PRINTABLESTRING   :ca
  147:d=2  hl=2 l=  17 cons: cont [ 0 ]
  149:d=3  hl=2 l=  15 prim: GENERALIZEDTIME   :20050528143109Z
  166:d=2  hl=2 l=  15 cons: cont [ 1 ]
  168:d=3  hl=2 l=  13 cons: SEQUENCE
  170:d=4  hl=2 l=   9 prim: OBJECT            :sha1WithRSAEncryption
  181:d=4  hl=2 l=   0 prim: NULL
  183:d=2  hl=2 l=  18 cons: cont [ 5 ]
  185:d=3  hl=2 l=  16 prim: OCTET STRING
  203:d=1  hl=4 l= 281 cons: cont [ 2 ]
  207:d=2  hl=4 l= 277 cons: SEQUENCE
  211:d=3  hl=4 l= 273 cons: SEQUENCE
  215:d=4  hl=4 l= 269 cons: SEQUENCE
  219:d=5  hl=2 l=  16 prim: INTEGER           :1C76EA7F40FA365A56
A5182216D718AC
  237:d=5  hl=3 l= 248 cons: SEQUENCE
  240:d=6  hl=2 l=   1 prim: cont [ 0 ]
  243:d=6  hl=2 l=  81 cons: cont [ 5 ]
  245:d=7  hl=2 l=  18 cons: SET
  247:d=8  hl=2 l=  16 cons: SEQUENCE
  249:d=9  hl=2 l=  10 prim: OBJECT            :domainComponent
  261:d=9  hl=2 l=   2 prim: PRINTABLESTRING   :no
  265:d=7  hl=2 l=  20 cons: SET
  267:d=8  hl=2 l=  18 cons: SEQUENCE
  269:d=9  hl=2 l=  10 prim: OBJECT            :domainComponent
  281:d=9  hl=2 l=   4 prim: PRINTABLESTRING   :ntnu
  287:d=7  hl=2 l=  15 cons: SET
  289:d=8  hl=2 l=  13 cons: SEQUENCE
  291:d=9  hl=2 l=   3 prim: OBJECT            :organizationalUnitName
  296:d=9  hl=2 l=   6 prim: PRINTABLESTRING   :people
  304:d=7  hl=2 l=  20 cons: SET
  306:d=8  hl=2 l=  18 cons: SEQUENCE
  308:d=9  hl=2 l=  10 prim: OBJECT            :userId
  320:d=9  hl=2 l=   4 prim: PRINTABLESTRING   :eian
  326:d=6  hl=3 l= 159 cons: cont [ 6 ]
  329:d=7  hl=2 l=  13 cons: SEQUENCE
  331:d=8  hl=2 l=   9 prim: OBJECT            :rsaEncryption
  342:d=8  hl=2 l=   0 prim: NULL
  344:d=7  hl=3 l= 141 prim: BIT STRING
  488:d=1  hl=3 l= 132 cons: cont [ 0 ]
  491:d=2  hl=3 l= 129 prim: BIT STRING
  623:d=1  hl=4 l=1010 cons: cont [ 1 ]
```

```
  627:d=2  hl=4 l=1006 cons: SEQUENCE
  631:d=3  hl=4 l=1002 cons: SEQUENCE
  635:d=4  hl=4 l= 645 cons: SEQUENCE
  639:d=5  hl=2 l=   3 cons: cont [ 0 ]
  641:d=6  hl=2 l=   1 prim: INTEGER           :02
  644:d=5  hl=2 l=   1 prim: INTEGER           :01
  647:d=5  hl=4 l= 299 cons: SEQUENCE
  651:d=6  hl=2 l=   7 prim: OBJECT            :dsaWithSHA1
  660:d=6  hl=4 l= 286 cons: SEQUENCE
  664:d=7  hl=3 l= 129 prim: INTEGER           :854CA950FA74969FC4
7029336D466FBC58C8D458A4CD6CB45E7C33F5F05B61920D343A4E8249ED64EDA2
6EFEBF8DAE985AA98B04B0D3301EB108AAB99992336DE061C61F7AF3400B97A3A7
AADDFE95F93E4C6DFB9DC6FCF2EAF9C56CB5A6FBAE884EF46F3AC18C24149A7B30
ABD6EFB6EE0463A15A0C2169EE866C7AD8961D37
  796:d=7  hl=2 l=  21 prim: INTEGER           :DFA669BD2AD265EE73
3DF6E5A239C13ACEF7837D
  819:d=7  hl=3 l= 128 prim: INTEGER           :0713CCC777B66D9905
81B144CB5047349F34A404A8F3E6C28440E89993F53C2E8C5956845E2EB8908BEE
2F084C4EAFDA9FA6D81609C878E23365A0E6EDE010CD59DC1FB0CC27627F463192
34DB229E1A6C19985CEA63F7CD86F062AB767FBDD0ADAAC03DF92374F1B53A1B58
F9127BFB0ACFB4D776B1BB870278F0873A024D82
  950:d=5  hl=2 l=  55 cons: SEQUENCE
  952:d=6  hl=2 l=  18 cons: SET
  954:d=7  hl=2 l=  16 cons: SEQUENCE
  956:d=8  hl=2 l=  10 prim: OBJECT            :domainComponent
  968:d=8  hl=2 l=   2 prim: PRINTABLESTRING   :no
  972:d=6  hl=2 l=  20 cons: SET
  974:d=7  hl=2 l=  18 cons: SEQUENCE
  976:d=8  hl=2 l=  10 prim: OBJECT            :domainComponent
  988:d=8  hl=2 l=   4 prim: PRINTABLESTRING   :ntnu
  994:d=6  hl=2 l=  11 cons: SET
  996:d=7  hl=2 l=   9 cons: SEQUENCE
  998:d=8  hl=2 l=   3 prim: OBJECT            :commonName
 1003:d=8  hl=2 l=   2 prim: PRINTABLESTRING   :ca
 1007:d=5  hl=2 l=  30 cons: SEQUENCE
 1009:d=6  hl=2 l=  13 prim: UTCTIME           :050528143108Z
 1024:d=6  hl=2 l=  13 prim: UTCTIME           :050529003108Z
 1039:d=5  hl=2 l=  81 cons: SEQUENCE
 1041:d=6  hl=2 l=  18 cons: SET
 1043:d=7  hl=2 l=  16 cons: SEQUENCE
 1045:d=8  hl=2 l=  10 prim: OBJECT            :domainComponent
 1057:d=8  hl=2 l=   2 prim: PRINTABLESTRING   :no
 1061:d=6  hl=2 l=  20 cons: SET
 1063:d=7  hl=2 l=  18 cons: SEQUENCE
 1065:d=8  hl=2 l=  10 prim: OBJECT            :domainComponent
 1077:d=8  hl=2 l=   4 prim: PRINTABLESTRING   :ntnu
 1083:d=6  hl=2 l=  15 cons: SET
 1085:d=7  hl=2 l=  13 cons: SEQUENCE
 1087:d=8  hl=2 l=   3 prim: OBJECT            :organizationalUnitName
```

```
 1092:d=8  hl=2 l=   6 prim: PRINTABLESTRING   :people
 1100:d=6  hl=2 l=  20 cons: SET
 1102:d=7  hl=2 l=  18 cons: SEQUENCE
 1104:d=8  hl=2 l=  10 prim: OBJECT            :userId
 1116:d=8  hl=2 l=   4 prim: PRINTABLESTRING   :eian
 1122:d=5  hl=3 l= 159 cons: SEQUENCE
 1125:d=6  hl=2 l=  13 cons: SEQUENCE
 1127:d=7  hl=2 l=   9 prim: OBJECT            :rsaEncryption
 1138:d=7  hl=2 l=   0 prim: NULL
 1140:d=6  hl=3 l= 141 prim: BIT STRING
 1284:d=4  hl=4 l= 299 cons: SEQUENCE
 1288:d=5  hl=2 l=   7 prim: OBJECT            :dsaWithSHA1
 1297:d=5  hl=4 l= 286 cons: SEQUENCE
 1301:d=6  hl=3 l= 129 prim: INTEGER           :854CA950FA74969FC4
7029336D466FBC58C8D458A4CD6CB45E7C33F5F05B61920D343A4E8249ED64EDA2
6EFEBF8DAE985AA98B04B0D3301EB108AAB99992336DE061C61F7AF3400B97A3A7
AADDFE95F93E4C6DFB9DC6FCF2EAF9C56CB5A6FBAE884EF46F3AC18C24149A7B30
ABD6EFB6EE0463A15A0C2169EE866C7AD8961D37
 1433:d=6  hl=2 l=  21 prim: INTEGER           :DFA669BD2AD265EE73
3DF6E5A239C13ACEF7837D
 1456:d=6  hl=3 l= 128 prim: INTEGER           :0713CCC777B66D9905
81B144CB5047349F34A404A8F3E6C28440E89993F53C2E8C5956845E2EB8908BEE
2F084C4EAFDA9FA6D81609C878E23365A0E6EDE010CD59DC1FB0CC27627F463192
34DB229E1A6C19985CEA63F7CD86F062AB767FBDD0ADAAC03DF92374F1B53A1B58
F9127BFB0ACFB4D776B1BB870278F0873A024D82
 1587:d=4  hl=2 l=  48 prim: BIT STRING
```

Appendix D

OpenVPN test

The client was running on a Dell Latitude X200 laptop with Debian Linux (sid), while the server was running on a Dell Optiplex GX260 workstation with OpenBSD 3.5. The client used a certificate with a 1024 bit RSA key, and the server and CA used a 1024 bit DSA key.

OpenVPN client configuration (openvpn_client.conf):

```
client
dev tun0
proto udp
remote 129.241.75.217 53
resolv-retry infinite
nobind
user eian
group eian
persist-key
persist-tun
ca ca_1024_crt.pem
cert client_crt.pem
key client_key.pem
verb 3
```

OpenVPN server configuration (openvpn_server.conf):

```
port 53
proto udp
dev tun0
ca /home/eian/fag/diplom/sw/ca_1024_crt.pem
cert server_crt.pem
key server_key.pem
dh dh1024.pem
server 10.8.0.0 255.255.255.0
ifconfig-pool-persist ipp.txt
keepalive 10 120
user nobody
group nobody
```

```
persist-key
persist-tun
status openvpn-status.log
verb 3
```

Client output:

```
eian@sigilion:~/fag/diplom/sw$ sudo openvpn --config openvpn_client.conf
Password:
Sat May 28 17:08:59 2005 OpenVPN 2.0 i386-pc-linux [SSL] [LZO] \
[EPOLL] built on May  4 2005
Sat May 28 17:08:59 2005 IMPORTANT: OpenVPN's default port number \
is now 1194, based on an official port number assignmentby IANA.  \
OpenVPN 2.0-beta16 and earlier used 5000 as the default port.
Sat May 28 17:08:59 2005 WARNING: No server certificate verification \
method has been enabled.  See http://openvpn.net/howto.html#mitm for \
more info.
Sat May 28 17:08:59 2005 Control Channel MTU parms [ L:1541 D:138 \
EF:38 EB:0 ET:0 EL:0 ]
Sat May 28 17:08:59 2005 Data Channel MTU parms [ L:1541 D:1450 \
EF:41 EB:4 ET:0 EL:0 ]
Sat May 28 17:08:59 2005 Local Options hash (VER=V4): '3514370b'
Sat May 28 17:08:59 2005 Expected Remote Options hash (VER=V4): \
'239669a8'
Sat May 28 17:08:59 2005 NOTE: UID/GID downgrade will be delayed \
because of --client, --pull, or --up-delay
Sat May 28 17:08:59 2005 UDPv4 link local: [undef]
Sat May 28 17:08:59 2005 UDPv4 link remote: 129.241.75.217:53
Sat May 28 17:08:59 2005 TLS: Initial packet from 129.241.75.217:53, \
sid=6275a9e6 a7f79385
Sat May 28 17:08:59 2005 VERIFY OK: depth=1, /DC=no/DC=ntnu/CN=ca
Sat May 28 17:08:59 2005 VERIFY OK: depth=0, /DC=no/DC=ntnu/CN=server
Sat May 28 17:08:59 2005 Data Channel Encrypt: Cipher 'BF-CBC' \
initialized with 128 bit key
Sat May 28 17:08:59 2005 Data Channel Encrypt: Using 160 bit message \
hash 'SHA1' for HMAC authentication
Sat May 28 17:08:59 2005 Data Channel Decrypt: Cipher 'BF-CBC' \
initialized with 128 bit key
Sat May 28 17:08:59 2005 Data Channel Decrypt: Using 160 bit message \
hash 'SHA1' for HMAC authentication
Sat May 28 17:08:59 2005 Control Channel: TLSv1, cipher TLSv1/SSLv3 \
DHE-DSS-AES256-SHA, 1024 bit DSA
Sat May 28 17:08:59 2005 [server] Peer Connection Initiated with \
129.241.75.217:53
Sat May 28 17:09:01 2005 SENT CONTROL [server]: 'PUSH_REQUEST' \
(status=1)
Sat May 28 17:09:01 2005 PUSH: Received control message: \
'PUSH_REPLY,route 10.8.0.1,ping 10,ping-restart 120,ifconfig \
10.8.0.6 10.8.0.5'
Sat May 28 17:09:01 2005 OPTIONS IMPORT: timers and/or timeouts modified
```

```
Sat May 28 17:09:01 2005 OPTIONS IMPORT: --ifconfig/up options modified
Sat May 28 17:09:01 2005 OPTIONS IMPORT: route options modified
Sat May 28 17:09:01 2005 TUN/TAP device tun0 opened
Sat May 28 17:09:01 2005 /sbin/ifconfig tun0 10.8.0.6 pointopoint \
10.8.0.5 mtu 1500
Sat May 28 17:09:01 2005 /sbin/route add -net 10.8.0.1 netmask \
255.255.255.255 gw 10.8.0.5
Sat May 28 17:09:01 2005 GID set to eian
Sat May 28 17:09:01 2005 UID set to eian
Sat May 28 17:09:01 2005 Initialization Sequence Completed

eian@sigilion:~$ /sbin/ifconfig -a tun0
tun0      Link encap:UNSPEC  HWaddr \
00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:10.8.0.6  P-t-P:10.8.0.5  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:336 (336.0 b)  TX bytes:336 (336.0 b)

eian@sigilion:~$ ping 10.8.0.1
PING 10.8.0.1 (10.8.0.1): 56 data bytes
64 bytes from 10.8.0.1: icmp_seq=0 ttl=255 time=23.897 ms
64 bytes from 10.8.0.1: icmp_seq=1 ttl=255 time=21.179 ms
64 bytes from 10.8.0.1: icmp_seq=2 ttl=255 time=90.488 ms
64 bytes from 10.8.0.1: icmp_seq=3 ttl=255 time=24.472 ms
--- 10.8.0.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 21.179/40.009/90.488/29.171 ms
```

Server output:

```
bash-2.05b# /usr/local/sbin/openvpn --config \
/usr/local/etc/openvpn_server.conf
Sat May 28 17:05:28 2005 OpenVPN 2.0 i386-unknown-openbsd3.5 [SSL] \
built on Apr 26 2005
Sat May 28 17:05:28 2005 Diffie-Hellman initialized with 1024 bit key
Sat May 28 17:05:28 2005 TLS-Auth MTU parms [ L:1541 D:138 EF:38 EB:0 \
  ET:0 EL:0 ]
Sat May 28 17:05:28 2005 gw 129.241.75.1
Sat May 28 17:05:28 2005 /sbin/ifconfig tun0 destroy
Sat May 28 17:05:28 2005 /sbin/ifconfig tun0 create
Sat May 28 17:05:28 2005 NOTE: Tried to delete pre-existing tun/tap \
instance -- No Problem if failure
Sat May 28 17:05:28 2005 /sbin/ifconfig tun0 10.8.0.1 10.8.0.2 mtu \
1500 netmask 255.255.255.255 up
Sat May 28 17:05:28 2005 TUN/TAP device /dev/tun0 opened
Sat May 28 17:05:28 2005 /sbin/route add -net 10.8.0.0 10.8.0.2 \
-netmask 255.255.255.0
```

```
add net 10.8.0.0: gateway 10.8.0.2
Sat May 28 17:05:28 2005 Data Channel MTU parms [ L:1541 D:1450 EF:41 \
  EB:4 ET:0 EL:0 ]
Sat May 28 17:05:28 2005 GID set to nobody
Sat May 28 17:05:28 2005 UID set to nobody
Sat May 28 17:05:28 2005 UDPv4 link local (bound): [undef]:53
Sat May 28 17:05:28 2005 UDPv4 link remote: [undef]
Sat May 28 17:05:28 2005 MULTI: multi_init called, r=256 v=256
Sat May 28 17:05:28 2005 IFCONFIG POOL: base=10.8.0.4 size=62
Sat May 28 17:05:28 2005 IFCONFIG POOL LIST
Sat May 28 17:05:28 2005 Initialization Sequence Completed
Sat May 28 17:08:51 2005 MULTI: multi_create_instance called
Sat May 28 17:08:51 2005 80.202.29.148:31704 Re-using SSL/TLS context
Sat May 28 17:08:51 2005 80.202.29.148:31704 Control Channel MTU parms \
[ L:1541 D:138 EF:38 EB:0 ET:0 EL:0 ]
Sat May 28 17:08:51 2005 80.202.29.148:31704 Data Channel MTU parms \
[ L:1541 D:1450 EF:41 EB:4 ET:0 EL:0 ]
Sat May 28 17:08:51 2005 80.202.29.148:31704 Local Options hash \
(VER=V4): '239669a8'
Sat May 28 17:08:51 2005 80.202.29.148:31704 Expected Remote Options \
hash (VER=V4): '3514370b'
Sat May 28 17:08:51 2005 80.202.29.148:31704 TLS: Initial packet from \
80.202.29.148:31704, sid=19da93d9 34fd344f
Sat May 28 17:08:52 2005 80.202.29.148:31704 VERIFY OK: depth=1, \
/DC=no/DC=ntnu/CN=ca
Sat May 28 17:08:52 2005 80.202.29.148:31704 VERIFY OK: depth=0, \
/DC=no/DC=ntnu/OU=people/UID=eian
Sat May 28 17:08:52 2005 80.202.29.148:31704 Data Channel Encrypt: \
Cipher 'BF-CBC' initialized with 128 bit key
Sat May 28 17:08:52 2005 80.202.29.148:31704 Data Channel Encrypt: \
Using 160 bit message hash 'SHA1' for HMAC authentication
Sat May 28 17:08:52 2005 80.202.29.148:31704 Data Channel Decrypt: \
Cipher 'BF-CBC' initialized with 128 bit key
Sat May 28 17:08:52 2005 80.202.29.148:31704 Data Channel Decrypt: \
Using 160 bit message hash 'SHA1' for HMAC authentication
Sat May 28 17:08:52 2005 80.202.29.148:31704 Control Channel: TLSv1, \
cipher TLSv1/SSLv3 DHE-DSS-AES256-SHA, 1024 bit RSA
Sat May 28 17:08:52 2005 80.202.29.148:31704 [] Peer Connection \
Initiated with 80.202.29.148:31704
Sat May 28 17:08:52 2005 80.202.29.148:31704 MULTI: Learn: 10.8.0.6 -> \
80.202.29.148:31704
Sat May 28 17:08:52 2005 80.202.29.148:31704 MULTI: primary virtual IP \
for 80.202.29.148:31704: 10.8.0.6
Sat May 28 17:08:53 2005 80.202.29.148:31704 PUSH: Received control \
message: 'PUSH_REQUEST'
Sat May 28 17:08:53 2005 80.202.29.148:31704 SENT CONTROL [UNDEF]: \
'PUSH_REPLY,route 10.8.0.1,ping 10,ping-restart 120,ifconfig 10.8.0.6 \
10.8.0.5' (status=1)
```

Appendix E

DSA certificate generation class

```
// SPEKE/CMP on-the-fly SSO implementation
//
// CA and server DSA certificate generator
// Author: Martin Eian


package no.ntnu.item.ttm4900;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.InputStream;
import java.io.OutputStream;

import java.math.BigInteger;

import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.Signature;

import java.security.spec.DSAPrivateKeySpec;

import java.util.Date;
import java.util.Vector;

import org.bouncycastle.asn1.ASN1EncodableVector;
import org.bouncycastle.asn1.ASN1InputStream;
import org.bouncycastle.asn1.DERBitString;
import org.bouncycastle.asn1.DERInteger;
import org.bouncycastle.asn1.DERObjectIdentifier;
import org.bouncycastle.asn1.DEROutputStream;
import org.bouncycastle.asn1.DERSequence;
```

```
import org.bouncycastle.asn1.util.ASN1Dump;
import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.asn1.x509.SubjectPublicKeyInfo;
import org.bouncycastle.asn1.x509.TBSCertificateStructure;
import org.bouncycastle.asn1.x509.Time;
import org.bouncycastle.asn1.x509.V3TBSCertificateGenerator;
import org.bouncycastle.asn1.x509.X509CertificateStructure;
import org.bouncycastle.asn1.x509.X509Name;

import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
import org.bouncycastle.crypto.digests.SHA1Digest;
import org.bouncycastle.crypto.generators.DSAKeyPairGenerator;
import org.bouncycastle.crypto.generators.DSAParametersGenerator;
import org.bouncycastle.crypto.params.DSAKeyGenerationParameters;
import org.bouncycastle.crypto.params.DSAParameters;
import org.bouncycastle.crypto.params.DSAPrivateKeyParameters;
import org.bouncycastle.crypto.params.DSAPublicKeyParameters;
import org.bouncycastle.crypto.signers.DSASigner;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.jce.provider.JDKDSAPublicKey;
import org.bouncycastle.jce.provider.X509CertificateObject;

import org.bouncycastle.openssl.PEMWriter;

public class gencert
{
  static SecureRandom secureRandom;

  // bit length of modulus for digital signatures
  static final int strength = 1024;
  // certainty for prime generation,
  // 2^-certainty probability of error
  static final int certainty = 100;
  // How many milliseconds a certificate should be valid
  // This will be checked on the server side as well
  // 1 hour = 3600 seconds
  static final long validityPeriod = 3600000;
  // 1 year
  static final long CAvalidityPeriod = 31536000000l;

  static final X509Name issuerX509Name =
    new X509Name("dc=no,dc=ntnu,cn=ca");
  static final boolean debug = true;

  public static void main(String[] args)
  {
    try {
      Security.addProvider(new BouncyCastleProvider());
```

```
secureRandom = new SecureRandom();
// START server generation
X509CertificateStructure caCert =
  X509CertificateStructure.getInstance(
    new ASN1InputStream(
      new FileInputStream("ca_1024.crt")
    ).readObject()
  );
DERInteger caDERX =
  DERInteger.getInstance(
    new ASN1InputStream(
      new FileInputStream("ca_1024.key")
    ).readObject()
  );
DERInteger caDERY =
  (DERInteger)
    caCert.getSubjectPublicKeyInfo().getPublicKey();

DERSequence caDERSequence =
  (DERSequence)
    caCert.getSignatureAlgorithm().getParameters();
DERInteger caDERP =
  (DERInteger)caDERSequence.getObjectAt(0);
DERInteger caDERQ =
  (DERInteger)caDERSequence.getObjectAt(1);
DERInteger caDERG =
  (DERInteger)caDERSequence.getObjectAt(2);

DSAPublicKeyParameters caPublicKey =
  new DSAPublicKeyParameters(
    caDERY.getValue(),
    new DSAParameters(
      caDERP.getValue(),
      caDERQ.getValue(),
      caDERG.getValue()
    )
  );

DSAPrivateKeyParameters caPrivateKey =
  new DSAPrivateKeyParameters(
    caDERX.getValue(),
    new DSAParameters(
      caDERP.getValue(),
      caDERQ.getValue(),
      caDERG.getValue()
    )
  );
// END server generation
if(debug) {System.out.print("Generating key pair...");}
```

```
DSAParametersGenerator dpg = new DSAParametersGenerator();
dpg.init(strength , certainty , secureRandom);
DSAKeyGenerationParameters kgp =
  new DSAKeyGenerationParameters(
    secureRandom,
    dpg.generateParameters()
  );
DSAKeyPairGenerator keyGen = new DSAKeyPairGenerator();
keyGen.init(kgp);
AsymmetricCipherKeyPair keyPair =
  keyGen.generateKeyPair();
DSAPrivateKeyParameters privateKey =
  (DSAPrivateKeyParameters)keyPair.getPrivate();
DSAPublicKeyParameters publicKey =
  (DSAPublicKeyParameters)keyPair.getPublic();

DERInteger[] dssPublicParameters =
  {new DERInteger(publicKey.getParameters().getP()),
   new DERInteger(publicKey.getParameters().getQ()),
   new DERInteger(publicKey.getParameters().getG())};
// DSA public key
AlgorithmIdentifier myAlgorithmIdentifier =
  new AlgorithmIdentifier(
    new DERObjectIdentifier("1.2.840.10040.4.1"),
    new DERSequence(dssPublicParameters)
  );

SubjectPublicKeyInfo mySubjectPublicKeyInfo =
  new SubjectPublicKeyInfo(
    myAlgorithmIdentifier,
    new DERInteger(publicKey.getY())
  );

if(debug) {System.out.println("OK");}

Date now = new Date();
V3TBSCertificateGenerator certGenerator =
  new V3TBSCertificateGenerator();
certGenerator.setIssuer(issuerX509Name);
certGenerator.setEndDate(
  new Time(new Date(now.getTime() + CAvalidityPeriod))
);
certGenerator.setStartDate(new Time(now));
certGenerator.setSerialNumber(new DERInteger(0));

DERInteger[] caDssPublicParameters =
  {new DERInteger(caPublicKey.getParameters().getP()),
   new DERInteger(caPublicKey.getParameters().getQ()),
   new DERInteger(caPublicKey.getParameters().getG())};
```

```
//DSAWITHSHA1
myAlgorithmIdentifier =
  new AlgorithmIdentifier(
    new DERObjectIdentifier("1.2.840.10040.4.3"),
    new DERSequence(caDssPublicParameters)
  );

certGenerator.setSignature(myAlgorithmIdentifier);
certGenerator.setSubject(
  new X509Name("dc=no,dc=ntnu,cn=server")
);
certGenerator.setSubjectPublicKeyInfo(
  mySubjectPublicKeyInfo
);
TBSCertificateStructure myTBSCert =
  certGenerator.generateTBSCertificate();
ASN1EncodableVector myASN1EncodableVector =
  new ASN1EncodableVector();

SHA1Digest mySHA1Digest = new SHA1Digest();
byte[] digestIn = myTBSCert.getEncoded();
mySHA1Digest.update(digestIn , 0 , digestIn.length);
byte[] digestOut = new byte[mySHA1Digest.getDigestSize()];
mySHA1Digest.doFinal(digestOut , 0);

DSASigner myDSASigner = new DSASigner();
myDSASigner.init(true , caPrivateKey);
BigInteger[] mySignature = new BigInteger[2];
mySignature = myDSASigner.generateSignature(digestOut);
DERInteger[] myDERIntegerArray =
  {new DERInteger(mySignature[0]),
   new DERInteger(mySignature[1]) };

DERSequence myDERSignature =
  new DERSequence(myDERIntegerArray);

myASN1EncodableVector.add(myTBSCert);
myASN1EncodableVector.add(myAlgorithmIdentifier);
myASN1EncodableVector.add(
  new DERBitString(myDERSignature)
);

X509CertificateStructure CACert =
  new X509CertificateStructure(
    new DERSequence(myASN1EncodableVector)
  );

DEROutputStream certFile =
  new DEROutputStream(new FileOutputStream("server.crt"));
```

```
      certFile.writeObject(CACert);
      certFile.close();

      PEMWriter pemCertFile =
        new PEMWriter(new FileWriter("server_crt.pem"));
      pemCertFile.writeObject(new X509CertificateObject(CACert));
      pemCertFile.close();

      DEROutputStream keyFile =
        new DEROutputStream(new FileOutputStream("server.key"));
      keyFile.writeObject(new DERInteger(privateKey.getX()));
      keyFile.close();

      KeyFactory myKeyFactory =
        KeyFactory.getInstance("DSA");
      PrivateKey myDSAPrivateKey =
        myKeyFactory.generatePrivate(
          new DSAPrivateKeySpec(
            privateKey.getX(),
            privateKey.getParameters().getP(),
            privateKey.getParameters().getQ(),
            privateKey.getParameters().getG()
          )
        );

      PEMWriter pemKeyFile =
        new PEMWriter(new FileWriter("server_key.pem"));
      pemKeyFile.writeObject(myDSAPrivateKey);
      pemKeyFile.close();

    } // end try

    catch (Exception e) {
      e.printStackTrace();
    } // end catch


  } // end main

}// end class gencert
```

Appendix F

RSA certificate generation class

```
// SPEKE/CMP on-the-fly SSO implementation
//
// CA and server RSA certificate generator
// Author: Martin Eian

package no.ntnu.item.ttm4900;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.InputStream;
import java.io.OutputStream;

import java.math.BigInteger;

import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.Signature;

import java.security.spec.RSAPrivateCrtKeySpec;

import java.util.Date;
import java.util.Vector;

import org.bouncycastle.asn1.ASN1EncodableVector;
import org.bouncycastle.asn1.ASN1InputStream;
import org.bouncycastle.asn1.DERBitString;
import org.bouncycastle.asn1.DERInteger;
import org.bouncycastle.asn1.DERNull;
import org.bouncycastle.asn1.DERObjectIdentifier;
import org.bouncycastle.asn1.DEROutputStream;
import org.bouncycastle.asn1.DERSequence;
```

```
import org.bouncycastle.asn1.util.ASN1Dump;
import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.asn1.x509.SubjectPublicKeyInfo;
import org.bouncycastle.asn1.x509.TBSCertificateStructure;
import org.bouncycastle.asn1.x509.Time;
import org.bouncycastle.asn1.x509.V3TBSCertificateGenerator;
import org.bouncycastle.asn1.x509.X509CertificateStructure;
import org.bouncycastle.asn1.x509.X509Name;

import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
import org.bouncycastle.crypto.digests.SHA1Digest;
import org.bouncycastle.crypto.engines.RSAEngine;
import org.bouncycastle.crypto.generators.RSAKeyPairGenerator;
import org.bouncycastle.crypto.params.RSAKeyGenerationParameters;
import org.bouncycastle.crypto.params.RSAKeyParameters;
import org.bouncycastle.crypto.params.RSAPrivateCrtKeyParameters;
import org.bouncycastle.crypto.signers.PSSSigner;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.jce.provider.X509CertificateObject;

import org.bouncycastle.openssl.PEMWriter;

public class gencert_rsa
{
  static SecureRandom secureRandom;

  // bit length of modulus for digital signatures
  static final int strength = 768;
  // certainty for prime generation,
  // 2^-certainty probability of error
  static final int certainty = 100;
  // How many milliseconds a certificate should be valid
  // This will be checked on the server side as well
  // 1 hour = 3600 seconds
  static final long validityPeriod = 3600000;
  // 1 year
  static final long CAvalidityPeriod = 31536000000l;

  static final X509Name issuerX509Name =
    new X509Name("dc=no,dc=ntnu,cn=ca");
  static final boolean debug = true;

  public static void main(String[] args)
  {
    try {
      Security.addProvider(new BouncyCastleProvider());
      secureRandom = new SecureRandom();
```

```
if(debug) {System.out.print("Generating key pair...");}
RSAKeyGenerationParameters rkgp =
  new RSAKeyGenerationParameters(
    new BigInteger("65537"),
    secureRandom,
    strength,
    certainty
  );
RSAKeyPairGenerator rpg = new RSAKeyPairGenerator();
rpg.init(rkgp);
AsymmetricCipherKeyPair RSAKeyPair =
  rpg.generateKeyPair();
RSAPrivateCrtKeyParameters privateKeyRSA =
  (RSAPrivateCrtKeyParameters)RSAKeyPair.getPrivate();
RSAKeyParameters publicKeyRSA =
  (RSAKeyParameters)RSAKeyPair.getPublic();

DERInteger[] rsaPublicParameters =
  {new DERInteger(publicKeyRSA.getModulus()),
   new DERInteger(publicKeyRSA.getExponent())};
DERSequence myDERSequence =
  new DERSequence(rsaPublicParameters);

System.out.println(strength + " bit n: " +
  publicKeyRSA.getModulus().toString());
System.out.println("Exponent e: " +
  publicKeyRSA.getExponent().toString());
// rsaEncryption
AlgorithmIdentifier myAlgorithmIdentifier =
  new AlgorithmIdentifier(
    new DERObjectIdentifier("1.2.840.113549.1.1.1"),
    new DERNull()
  );
SubjectPublicKeyInfo mySubjectPublicKeyInfo =
  new SubjectPublicKeyInfo(
    myAlgorithmIdentifier,
    myDERSequence
  );
if(debug) {System.out.println("OK");}

Date now = new Date();
V3TBSCertificateGenerator certGenerator =
  new V3TBSCertificateGenerator();
certGenerator.setIssuer(issuerX509Name);
certGenerator.setEndDate(
  new Time(new Date(now.getTime() + CAvalidityPeriod)));
certGenerator.setStartDate(new Time(now));
certGenerator.setSerialNumber(new DERInteger(0));
```

```
//sha-1WithRSAEncryption
myAlgorithmIdentifier =
  new AlgorithmIdentifier(
    new DERObjectIdentifier("1.2.840.113549.1.1.5"),
    new DERNull()
  );

certGenerator.setSignature(myAlgorithmIdentifier);
certGenerator.setSubject(
  new X509Name("dc=no,dc=ntnu,cn=ca"));
certGenerator.setSubjectPublicKeyInfo(
  mySubjectPublicKeyInfo);
TBSCertificateStructure myTBSCert =
  certGenerator.generateTBSCertificate();
ASN1EncodableVector myASN1EncodableVector =
  new ASN1EncodableVector();

// 64 = length of salt in number of bytes
// Usually number of bytes in hash function.
// or SHA-1: 160 / 8 = 20
PSSSigner myPSSSigner =
  new PSSSigner(new RSAEngine(), new SHA1Digest(), 20);
myPSSSigner.init(true, privateKeyRSA);
myPSSSigner.update(
  myTBSCert.getEncoded(),
  0,
  myTBSCert.getEncoded().length
);
byte[] myRSASignature = myPSSSigner.generateSignature();

myASN1EncodableVector.add(myTBSCert);
myASN1EncodableVector.add(myAlgorithmIdentifier);
myASN1EncodableVector.add(
  new DERBitString(myRSASignature));

X509CertificateStructure CACert =
  new X509CertificateStructure(
    new DERSequence(myASN1EncodableVector)
  );

DEROutputStream certFile =
  new DEROutputStream(
    new FileOutputStream("ca_rsa_"+strength+".crt")
  );
certFile.writeObject(CACert);
certFile.close();

PEMWriter pemCertFile =
  new PEMWriter(
```

```
        new FileWriter("ca_rsa_"+strength+"_crt.pem")
      );
    pemCertFile.writeObject(
      new X509CertificateObject(CACert));
    pemCertFile.close();

    KeyFactory myKeyFactory = KeyFactory.getInstance("RSA");
    PrivateKey myRSAPrivateKey =
      myKeyFactory.generatePrivate(
        new RSAPrivateCrtKeySpec(
          privateKeyRSA.getModulus(),
          privateKeyRSA.getPublicExponent(),
          privateKeyRSA.getExponent(),
          privateKeyRSA.getP(),
          privateKeyRSA.getQ(),
          privateKeyRSA.getDP(),
          privateKeyRSA.getDQ(),
          privateKeyRSA.getQInv()
        )
      );
    PEMWriter pemKeyFile =
      new PEMWriter(
        new FileWriter("ca_rsa_"+strength+"_key.pem")
      );
    pemKeyFile.writeObject(myRSAPrivateKey);
    pemKeyFile.close();

  } // end try

  catch (Exception e) {
    e.printStackTrace();
  } // end catch


 } // end main

}// end class gencert_rsa
```

Appendix G

SPEKE modulus generation class

```
// SPEKE/CMP on-the-fly SSO implementation
//
// Class that generates safe primes that can
// be used as SPEKE moduli
// Author: Martin Eian

package no.ntnu.item.ttm4900;

import java.math.BigInteger;

import java.security.SecureRandom;
import java.security.Security;

import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class genmoduli
{
  static SecureRandom secureRandom;

  static BigInteger ONE = BigInteger.valueOf(1);
  static BigInteger TWO = BigInteger.valueOf(2);

  static final int certainty = 100;

  public static void main(String[] args)
  {
    try {
      Security.addProvider(new BouncyCastleProvider());
      secureRandom = new SecureRandom();

      // We want to create a safe prime p, which means that
```

```
      // p = 2q + 1 for some prime q
      int mLength =
        (args.length > 0 ? Integer.parseInt(args[0]) : 768);

      BigInteger p, q;

      while (true) {

        q = BigInteger.probablePrime(mLength - 1,
                                     secureRandom);
        System.out.print(".");
        p = q.multiply(TWO).add(ONE);
        if (p.isProbablePrime(certainty)) {
          System.out.println(
            mLength + " bit modulus: " + p.toString()
          );
          break;
        }
      } // end while loop generating q and p

    } // end try

    catch (Exception e) {
      e.printStackTrace();
    } // end catch


  } // end main

}// end class genmoduli
```

Appendix H

AS/CA implementation

```
// SPEKE/CMP SSO implementation
//
// This class implements the AS/CA
// Author: Martin Eian

package no.ntnu.item.ttm4900;

import com.novosec.pkix.asn1.cmp.*;
import com.novosec.pkix.asn1.crmf.*;

import java.io.FileInputStream;
import java.io.FileReader;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;

import java.math.BigInteger;

import java.util.Properties;

import java.security.AlgorithmParameters;
import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.Provider;
import java.security.SecureRandom;
import java.security.Security;
import java.security.Signature;

import java.util.Date;
import java.util.Properties;
import java.util.Vector;
```

```
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

import javax.servlet.*;
import javax.servlet.http.*;

import org.bouncycastle.asn1.ASN1EncodableVector;
import org.bouncycastle.asn1.ASN1InputStream;
import org.bouncycastle.asn1.DERBitString;
import org.bouncycastle.asn1.DERGeneralizedTime;
import org.bouncycastle.asn1.DERInteger;
import org.bouncycastle.asn1.DEROctetString;
import org.bouncycastle.asn1.DEROutputStream;
import org.bouncycastle.asn1.DERSequence;
import org.bouncycastle.asn1.util.ASN1Dump;
import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.asn1.x509.GeneralName;
import org.bouncycastle.asn1.x509.SubjectPublicKeyInfo;
import org.bouncycastle.asn1.x509.TBSCertificateStructure;
import org.bouncycastle.asn1.x509.Time;
import org.bouncycastle.asn1.x509.V3TBSCertificateGenerator;
import org.bouncycastle.asn1.x509.X509CertificateStructure;
import org.bouncycastle.asn1.x509.X509Name;

import org.bouncycastle.crypto.CryptoException;
import org.bouncycastle.crypto.digests.SHA1Digest;
import org.bouncycastle.crypto.engines.AESFastEngine;
import org.bouncycastle.crypto.engines.RSAEngine;
import org.bouncycastle.crypto.params.DSAParameters;
import org.bouncycastle.crypto.params.DSAPrivateKeyParameters;
import org.bouncycastle.crypto.params.DSAPublicKeyParameters;
import org.bouncycastle.crypto.params.RSAKeyParameters;
import org.bouncycastle.crypto.params.RSAPrivateCrtKeyParameters;
import org.bouncycastle.crypto.params.KeyParameter;
import org.bouncycastle.crypto.signers.DSASigner;
import org.bouncycastle.crypto.signers.PSSSigner;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.jce.provider.JDKDSAPublicKey;
import org.bouncycastle.jce.provider.JCERSAPrivateCrtKey;
import org.bouncycastle.jce.provider.JCERSAPrivateKey;
import org.bouncycastle.jce.provider.RSAUtil;
import org.bouncycastle.jce.provider.X509CertificateObject;

import org.bouncycastle.openssl.PEMReader;


public class server extends HttpServlet
{
```

```
//static final String securityProvider = "BC";
static SecureRandom secureRandom;

static final String propertiesFile = "ca.properties";

// certainty for prime generation, 2^(-certainty)
// probability of error
static final int certainty = 100;
static final X509Name issuerX509Name =
  new X509Name("dc=no,dc=ntnu,cn=ca");
static boolean debug = false;
static boolean timing = false;
static boolean rsa = false;
static BigInteger ZERO = BigInteger.valueOf(0);
static BigInteger ONE = BigInteger.valueOf(1);
static BigInteger TWO = BigInteger.valueOf(2);

// H(P) used for SPEKE
static BigInteger P = null;

// modulus for SPEKE
static BigInteger p;

static BigInteger serialNumber = new BigInteger("1");

// User name - verifier mapping
static Properties passwordTable = new Properties();

String userName = "";
Date startTime, stopTime;
DSASigner signDSASigner = new DSASigner();
Provider bcProvider = new BouncyCastleProvider();
static DERInteger caDERX, caDERY, caDERP, caDERQ, caDERG;
static DSAPublicKeyParameters caPublicKey;
static DSAPrivateKeyParameters caPrivateKey;
static X509CertificateStructure caCert;
static int modulusLength,randlength,strength;
static long validityPeriod,CAvalidityPeriod;
static String modulus,certfile,keyfile,CAcertfile,domain;
static RSAPrivateCrtKeyParameters privateKeyRSA = null;
static AlgorithmIdentifier caSigAlgorithmIdentifier;

public void init(ServletConfig config) throws ServletException
{
  try{

    // Start loading configuration file
    Properties myProperties = new Properties();
    myProperties.load(new FileInputStream(propertiesFile));
```

```
// the modulus p used for SPEKE
modulusLength = Integer.parseInt(
  myProperties.getProperty("modulusLength"));
// bit length of random parameters (A , B) in DH
randlength = Integer.parseInt(
  myProperties.getProperty("randLength"));
// bit length of modulus for digital signatures
strength = Integer.parseInt(
  myProperties.getProperty("strength"));
// How many milliseconds a certificate should be valid
validityPeriod = Long.parseLong(
  myProperties.getProperty("validityPeriod"));
// 1 year
CAvalidityPeriod = Long.parseLong(
  myProperties.getProperty("CAvalidityPeriod"));
domain = myProperties.getProperty("domain");
String debugString = myProperties.getProperty("debug");
String timingString = myProperties.getProperty("timing");
String rsaString = myProperties.getProperty("rsa");

// SPEKE modulus
if (modulusLength == 2048) {
  modulus = myProperties.getProperty("2048modulus");
}
else if (modulusLength == 1024) {
  modulus = myProperties.getProperty("1024modulus");
}
else {
  modulus = myProperties.getProperty("768modulus");
}
p = new BigInteger(modulus);

if (debugString.equalsIgnoreCase("true")) {
  debug = true;
}
if (timingString.equalsIgnoreCase("true")) {
  timing = true;
}

if (rsaString.equalsIgnoreCase("true")) {
  // CA will use RSA for digital signatures
  rsa = true;
}

// Done loading configuration file

Security.addProvider(bcProvider);
secureRandom = new SecureRandom();
```

```
if(rsa){
  // CA uses RSA
  certfile = "ca_rsa_" + strength + ".crt";
  keyfile = "ca_rsa_" + strength + "_key.pem";
  CAcertfile = "ca_rsa_" + strength + ".crt";
  // Load private/public key pair
  PEMReader myPEMReader =
    new PEMReader(new FileReader(keyfile));
  KeyPair keyPairRSA =
    (KeyPair)myPEMReader.readObject();
  privateKeyRSA =
    (RSAPrivateCrtKeyParameters)
      RSAUtil.generatePrivateKeyParameter(
        (JCERSAPrivateKey)keyPairRSA.getPrivate());
  caCert = X509CertificateStructure.getInstance(
    new ASN1InputStream(
      new FileInputStream(certfile)).readObject());
}
else{
  // CA uses DSA
  certfile = "ca_" + strength + ".crt";
  keyfile = "ca_" + strength + ".key";
  CAcertfile = "ca_" + strength + ".crt";
  // Load private/public key pair
  caCert = X509CertificateStructure.getInstance(
    new ASN1InputStream(
      new FileInputStream(certfile)).readObject());
  caDERX = DERInteger.getInstance(
    new ASN1InputStream(
      new FileInputStream(keyfile)).readObject());
  caDERY = (DERInteger)caCert.getSubjectPublicKeyInfo().
    getPublicKey();

  DERSequence caDERSequence =
    (DERSequence)caCert.getSignatureAlgorithm().
      getParameters();
  caDERP = (DERInteger)caDERSequence.getObjectAt(0);
  caDERQ = (DERInteger)caDERSequence.getObjectAt(1);
  caDERG = (DERInteger)caDERSequence.getObjectAt(2);

  caPublicKey = new DSAPublicKeyParameters(
    caDERY.getValue(),
    new DSAParameters(caDERP.getValue(),
                      caDERQ.getValue(),
                      caDERG.getValue()));

  caPrivateKey = new DSAPrivateKeyParameters(
    caDERX.getValue(),
    new DSAParameters(caDERP.getValue(),
```

```
                          caDERQ.getValue(),
                          caDERG.getValue()));
      signDSASigner.init(true , caPrivateKey);
    }
    caSigAlgorithmIdentifier = caCert.getSignatureAlgorithm();
    // password is SHA-1("test123")
    passwordTable.setProperty("eian" ,
      "65387856594671371314962910427547810457186772780 4");
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}// end init()

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{


  if (request.getContentType() != null &&
    request.getContentType().equals("application/pkixcmp"))
  {
    try{

      if(debug) {System.out.println(
        "CONTENT-TYPE application/pkixcmp");}

      // true if this is a certificate renewal request
      // or token-based, false otherwise
      boolean renew = false;
      DSASigner myDSASigner = new DSASigner();
      PSSSigner myRSASigner =
        new PSSSigner(new RSAEngine() , new SHA1Digest() , 20);
      Date timeNow;
      PKIMessage myPKIMessage =
        PKIMessage.getInstance(
          new ASN1InputStream(
            request.getInputStream()).readObject());

      // See if message has message protection
      DERBitString verDERBitString =
        myPKIMessage.getProtection();
      if(myPKIMessage.getProtection() != null) {
        // Message protection found, client authenticates
        // using digital signature
        renew = true;
        X509CertificateStructure oldClientCert =
          myPKIMessage.getExtraCert(0);
```

```
// Verify message protection (digital signature)

if(timing) {startTime = new Date();}

if(checkSignature(myRSASigner,
          myDSASigner,
          oldClientCert,
          myPKIMessage.getProtection().getBytes(),
          myPKIMessage.getProtectedBytes(),
          myPKIMessage.getHeader().getProtectionAlg()))
{
  if(debug) {System.out.println(
    "Client signature verified!");}
}
else {
  if(debug) {System.out.println(
    "Client signature verification FAILED!");}
  throw new Exception("Message protection invalid.");
}
if(timing) {stopTime = new Date();}
if(timing) {System.out.println(
  "Verification of client signature took " +
  (stopTime.getTime() - startTime.getTime()) +
  " milliseconds.");}

// Now check the CA signature on the client certificate
if(timing) {startTime = new Date();}

if(checkSignature(myRSASigner,
        myDSASigner,
        caCert,
        oldClientCert.getSignature().getBytes(),
        oldClientCert.getTBSCertificate().getEncoded(),
        oldClientCert.getSignatureAlgorithm()))
{
  if(debug) {System.out.println(
    "CA signature verified!");}
}
else {
  if(debug) {System.out.println(
    "CA signature verification FAILED!");}
  throw new Exception(
    "CA signature on certificate invalid.");
}
if(timing) {stopTime = new Date();}
if(timing) {System.out.println(
  "Verification of CA signature took " +
  (stopTime.getTime() - startTime.getTime()) +
```

```
    " milliseconds.");}

  // Must check that the certificate and
  // template have the same subject
  // and that the certificate has not expired

  // Check that the subject is the same for
  // cert and template
  X509Name oldX509Name = oldClientCert.getSubject();
  if(!oldX509Name.toString().equals(
    myPKIMessage.getBody().getCr().getCertReqMsg(0).
    getCertReq().getCertTemplate().getSubject().
    toString()))
  {
    throw new Exception(
      "Client is trying to renew a certificate with a " +
      "different user name!");
  }
  timeNow = new Date();
  // Check that the certificate has not expired
  if (timeNow.after(
    oldClientCert.getEndDate().getDate()))
  {
    throw new Exception(
      "Client certificate has expired!");
  }

  // See if the certificate is not yet valid
  if (timeNow.before(
    oldClientCert.getStartDate().getDate()))
  {
    throw new Exception(
      "Client certificate is not yet valid!");
  }
}

// Get the certificate template
CertTemplate myCertTemplate =
  myPKIMessage.getBody().getCr().getCertReqMsg(0).
  getCertReq().getCertTemplate();
SubjectPublicKeyInfo mySubjectPublicKeyInfo =
  myCertTemplate.getPublicKey();
X509Name myX509Name = myCertTemplate.getSubject();
if(debug) {System.out.println(
  "Subject name: " + myX509Name.toString());}
Vector oids = myX509Name.getOIDs();
Vector oidValues = myX509Name.getValues();
// Get the user name
// Lame check, should check domain as well
```

```
if(debug) {System.out.println(
  "OID: " + oids.lastElement().toString());}
if(debug) {System.out.println("OIDValue: " +
  oidValues.lastElement().toString());}
// Lookup user name, find verifier H(P)
if (oids.lastElement().toString().equalsIgnoreCase(
  "0.9.2342.19200300.100.1.1"))
{
        userName = oidValues.lastElement().toString();
        P = new BigInteger(passwordTable.getProperty(
          userName));
}
if(debug) {System.out.println("User name: " + userName);}
if(debug) {System.out.println(
  "Verifier: " + P.toString());}

// Create CertResponse
// PKIStatusInfo = 1 means that request is granted,
// but the certificate template was modified
CertResponse myCertResponse =
  new CertResponse(
    myPKIMessage.getBody().getCr().getCertReqMsg(0).
    getCertReq().getCertReqId(),
    new PKIStatusInfo(new DERInteger(1)));
// Used to transfer the encrypted CA root certificate
CertResponse CACertResponse =
  new CertResponse(
    myPKIMessage.getBody().getCr().getCertReqMsg(0).
    getCertReq().getCertReqId(),
    new PKIStatusInfo(new DERInteger(1)));

// Create signed and (possibly) encrypted certificate
// and add to response
Date now = new Date();
V3TBSCertificateGenerator certGenerator =
  new V3TBSCertificateGenerator();
certGenerator.setIssuer(issuerX509Name);
certGenerator.setEndDate(
  new Time(new Date(now.getTime() + validityPeriod)));
certGenerator.setStartDate(new Time(now));
certGenerator.setSerialNumber(
  new DERInteger(serialNumber));

// Just increment the serial number by one.
// Not very clever, but this is an experimental
// implementation
serialNumber.add(ONE);

certGenerator.setSignature(
```

```
  caSigAlgorithmIdentifier);
certGenerator.setSubject(myX509Name);
certGenerator.setSubjectPublicKeyInfo(
  mySubjectPublicKeyInfo);
TBSCertificateStructure myTBSCert =
  certGenerator.generateTBSCertificate();

ASN1EncodableVector myASN1EncodableVector =
  new ASN1EncodableVector();

myASN1EncodableVector.add(myTBSCert);
myASN1EncodableVector.add(caSigAlgorithmIdentifier);
if(timing) {startTime = new Date();}
if(rsa){
  myASN1EncodableVector.add(
    doRSASignature(myRSASigner,
                   privateKeyRSA,
                   myTBSCert.getEncoded()));
}
else {
  myASN1EncodableVector.add(
    doDSASignature(caPrivateKey,
                   myTBSCert.getEncoded()));
}
if(timing) {stopTime = new Date();}
if(timing) {System.out.println(
  "Signing client certificate took " +
  (stopTime.getTime() - startTime.getTime()) +
  " milliseconds.");}

X509CertificateStructure myX509Cert =
  new X509CertificateStructure(
    new DERSequence(myASN1EncodableVector));

// Set sender and recipient
PKIHeader myPKIHeader =
  new PKIHeader(new DERInteger(1),
                new GeneralName(
                  new X509Name("dc=no,dc=ntnu,cn=ca")),
                new GeneralName(
                  new X509Name(domain+",cn="+userName)));

if(renew){
  myCertResponse.setCertifiedKeyPair(
    new CertifiedKeyPair(
      new CertOrEncCert(myX509Cert , 0)));
}
else{
  // Client requested that the certificate should be
```

```
// encrypted with the SPEKE session key

// Generate random exponent, compute shared session key
// Compute SPEKE key, encrypt certificates

// Generate random value RB
if(timing) {startTime = new Date();}
BigInteger RB =
  new BigInteger(randlength , secureRandom);
if(debug) {System.out.println(
  "RB =  " + RB.toString());}

if(timing) {stopTime = new Date();}
if(timing) {System.out.println(
  "Generation of RB took " +
  (stopTime.getTime() - startTime.getTime()) +
  " milliseconds.");}
// Compute SPEKE verifier H(password)^{2*B}

if(timing) {startTime = new Date();}
BigInteger ha = new BigInteger(
  myPKIMessage.getHeader().getSenderKID().getOctets());
// SPEKE constraint
if (ha.compareTo(ZERO) == 0)
  {throw new Exception("g^R_A is 0, exiting.");}
if(debug) {System.out.println(
  "H(P)^(2RA) = " + ha.toString());}

BigInteger hb = P.modPow(RB.multiply(TWO) , p);
if(debug) {System.out.println(
  "H(P)^(2RB) = " + hb.toString());}

BigInteger k = ha.modPow(RB.multiply(TWO) , p);
if(debug) {System.out.println(
  "H(P)^(4RA*RB) = " + k.toString());}
// SPEKE constraint
if (k.compareTo(ONE) == 0)
  {throw new Exception("K is 1, exiting.");}
// Hash the key to provide forward secrecy
SHA1Digest keySHA1Digest = new SHA1Digest();
keySHA1Digest.update(
  k.toByteArray() , 0 , k.toByteArray().length);
byte[] K_S = new byte[keySHA1Digest.getDigestSize()];
keySHA1Digest.doFinal(K_S , 0);

if(timing) {stopTime = new Date();}
if(timing) {System.out.println(
  "SPEKE key calculation took " +
  (stopTime.getTime() - startTime.getTime()) +
```

```
  " milliseconds.");}

if(timing) {startTime = new Date();}
// 16 bytes = 128 bit AES key
SecretKeySpec myAESKeySpec =
  new SecretKeySpec(K_S , 0 , 16 , "AES");
// Moving this one to init() leads to
// java.lang.IllegalStateException:
// Cipher not initialized
// under high load
Cipher myCipher =
  Cipher.getInstance(
    "AES/CBC/PKCS7Padding" , bcProvider);
myCipher.init(Cipher.ENCRYPT_MODE,
              myAESKeySpec,
              secureRandom);
byte[] myAESIV = myCipher.getIV();

EncryptedValue myEncryptedCert =
  new EncryptedValue(
    new DERBitString(
      myCipher.doFinal(myX509Cert.getEncoded())));
// Store the Initialization Vector for AES
myEncryptedCert.setEncSymmKey(
  new DERBitString(myAESIV));
if(timing) {stopTime = new Date();}
if(timing) {System.out.println(
  "AES encryption of client certificate took " +
  (stopTime.getTime() - startTime.getTime()) +
  " milliseconds.");}

if(timing) {startTime = new Date();}
myCipher.init(Cipher.ENCRYPT_MODE,
              myAESKeySpec,
              secureRandom);
myAESIV = myCipher.getIV();

EncryptedValue CAEncryptedCert =
  new EncryptedValue(
    new DERBitString(
      myCipher.doFinal(caCert.getEncoded())));
CAEncryptedCert.setEncSymmKey(
  new DERBitString(myAESIV));

if(timing) {stopTime = new Date();}
if(timing) {System.out.println(
  "AES encryption of CA certificate took " +
  (stopTime.getTime() - startTime.getTime()) +
  " milliseconds.");}
```

```
      // Change 1 to 0 for unencrypted cert
      myCertResponse.setCertifiedKeyPair(new
        CertifiedKeyPair(
          new CertOrEncCert(myEncryptedCert , 1)));
      CACertResponse.setCertifiedKeyPair(
        new CertifiedKeyPair(
          new CertOrEncCert(CAEncryptedCert , 1)));
      // Store the SPEKE public key
      myPKIHeader.setSenderKID(
        new DEROctetString(hb.toByteArray()));
      myPKIHeader.setRecipKID(
        new DEROctetString(ha.toByteArray()));
    }

    BigInteger myNonce =
      new BigInteger(128 , secureRandom);

    myPKIHeader.setMessageTime(
      new DERGeneralizedTime(new Date()));
    myPKIHeader.setSenderNonce(
      new DEROctetString(myNonce.toByteArray()));
    myPKIHeader.setRecipNonce(
      myPKIMessage.getHeader().getSenderNonce());

    CertRepMessage myCertRepMessage =
      new CertRepMessage(myCertResponse);
    if(!renew){
      // Initial authentication, include encrypted
      // CA root certificate
      myCertRepMessage.addResponse(CACertResponse);
    }
    PKIMessage repPKIMessage =
      new PKIMessage(myPKIHeader,
                     new PKIBody(myCertRepMessage , 3));

    // Write response
    DEROutputStream sos =
      new DEROutputStream(response.getOutputStream());
    response.setContentType("application/pkixcmp");
    sos.writeObject(repPKIMessage);
    sos.close();

    }
    catch (Exception e) {
      e.printStackTrace();
    }
}
else {
```

```
    PrintWriter out = response.getWriter();
    out.println("Move along, nothing to see here.");
    out.close();
  }
}

public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
                   throws ServletException, IOException
{
  doGet(request, response);
}

private DERBitString doDSASignature(
  DSAPrivateKeyParameters privateKey,
                          byte[] message)
{
  // Run message through SHA-1
  SHA1Digest mySHA1Digest = new SHA1Digest();
  mySHA1Digest.update(message , 0 , message.length);
  byte[] digestOut = new byte[mySHA1Digest.getDigestSize()];
  mySHA1Digest.doFinal(digestOut , 0);

  // Sign message digest and return signature
  BigInteger[] myRS =
    signDSASigner.generateSignature(digestOut);
  DERInteger[] myDERIntegerArray =
    {new DERInteger(myRS[0]),
     new DERInteger(myRS[1])};

  return new DERBitString(new DERSequence(myDERIntegerArray));
}

private DERBitString doRSASignature(
  PSSSigner myRSASigner,
  RSAPrivateCrtKeyParameters privateKeyRSA,
  byte[] message)
  throws CryptoException
{
  myRSASigner.init(true, privateKeyRSA);
  myRSASigner.update(message , 0 , message.length);

  return new DERBitString(myRSASigner.generateSignature());
}

private boolean checkSignature(
  PSSSigner myRSASigner,
  DSASigner myDSASigner,
  X509CertificateStructure inCert,
```

```
  byte[] sig,
  byte[] message,
  AlgorithmIdentifier sigAlgId)
  throws CryptoException , IOException
{

  SubjectPublicKeyInfo clientSubjectPublicKeyInfo =
    inCert.getSubjectPublicKeyInfo();
  AlgorithmIdentifier pubKeyAlgId =
    clientSubjectPublicKeyInfo.getAlgorithmId();

  // OID 1.2.840.113549.1.1.5 = sha-1WithRSAEncryption
  // OID 1.2.840.113549.1.1.1 = RSAEncryption (RSA public key)
  if(sigAlgId.getObjectId().getId().equals(
    "1.2.840.113549.1.1.5"))
  {
    DERSequence RSADERSequence =
      (DERSequence)clientSubjectPublicKeyInfo.getPublicKey();

    myRSASigner.init(
      false,
      new RSAKeyParameters(
        false,
        ((DERInteger)RSADERSequence.getObjectAt(0)).getValue(),
        ((DERInteger)RSADERSequence.getObjectAt(1)).getValue()
      )
    );
    myRSASigner.update(message , 0 , message.length);
    return myRSASigner.verifySignature(sig);
  }
  else{
    DERSequence sigDERSequence, verDERSequence;
    DERInteger sigR, sigS, DERP, DERQ, DERR;
    ASN1InputStream ais = new ASN1InputStream(sig);
    sigDERSequence = (DERSequence)ais.readObject();
    sigR = (DERInteger)sigDERSequence.getObjectAt(0);
    sigS = (DERInteger)sigDERSequence.getObjectAt(1);
    // this sequence consists of 3 Integers: p, q and g.
    verDERSequence = (DERSequence)sigAlgId.getParameters();
    DERP = (DERInteger)verDERSequence.getObjectAt(0);
    DERQ = (DERInteger)verDERSequence.getObjectAt(1);
    DERR = (DERInteger)verDERSequence.getObjectAt(2);
    DSAPublicKeyParameters publicKey =
      new DSAPublicKeyParameters(
        ((DERInteger)clientSubjectPublicKeyInfo.
        getPublicKey()).getValue(),
        new DSAParameters(
          DERP.getValue(),
          DERQ.getValue(),
```

```
            DERR.getValue()
          )
        );

      SHA1Digest mySHA1Digest = new SHA1Digest();
      mySHA1Digest.update(message , 0 , message.length);
      byte[] digestOut = new byte[mySHA1Digest.getDigestSize()];
      mySHA1Digest.doFinal(digestOut , 0);
      myDSASigner.init(false , publicKey);
      return myDSASigner.verifySignature(digestOut,
                                         sigR.getValue(),
                                         sigS.getValue());
    }
  }// end method checkSignature
}// end class server
```

Appendix I

Client implementation

```
// SPEKE/CMP on-the-fly SSO implementation
//
// Client implementation
// Author: Martin Eian

package no.ntnu.item.ttm4900;

import com.novosec.pkix.asn1.cmp.*;
import com.novosec.pkix.asn1.crmf.*;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStream;

import java.math.BigInteger;

import java.net.URLConnection;
import java.net.URL;

import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.Provider;
import java.security.SecureRandom;
import java.security.Security;

import java.security.spec.DSAPrivateKeySpec;
import java.security.spec.RSAPrivateCrtKeySpec;

import java.util.Date;
import java.util.Properties;

import javax.crypto.Cipher;
```

```
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

import org.bouncycastle.asn1.ASN1InputStream;
import org.bouncycastle.asn1.DERBitString;
import org.bouncycastle.asn1.DERGeneralizedTime;
import org.bouncycastle.asn1.DERInteger;
import org.bouncycastle.asn1.DERNull;
import org.bouncycastle.asn1.DERObject;
import org.bouncycastle.asn1.DERObjectIdentifier;
import org.bouncycastle.asn1.DEROctetString;
import org.bouncycastle.asn1.DEROutputStream;
import org.bouncycastle.asn1.DERSequence;
import org.bouncycastle.asn1.util.ASN1Dump;
import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.asn1.x509.GeneralName;
import org.bouncycastle.asn1.x509.SubjectPublicKeyInfo;
import org.bouncycastle.asn1.x509.Time;
import org.bouncycastle.asn1.x509.X509CertificateStructure;
import org.bouncycastle.asn1.x509.X509Name;

import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
import org.bouncycastle.crypto.CryptoException;
import org.bouncycastle.crypto.digests.SHA1Digest;
import org.bouncycastle.crypto.engines.RSAEngine;
import org.bouncycastle.crypto.generators.DSAKeyPairGenerator;
import org.bouncycastle.crypto.generators.DSAParametersGenerator;
import org.bouncycastle.crypto.generators.RSAKeyPairGenerator;
import org.bouncycastle.crypto.params.DSAKeyGenerationParameters;
import org.bouncycastle.crypto.params.DSAParameters;
import org.bouncycastle.crypto.params.DSAPrivateKeyParameters;
import org.bouncycastle.crypto.params.DSAPublicKeyParameters;
import org.bouncycastle.crypto.params.RSAKeyGenerationParameters;
import org.bouncycastle.crypto.params.RSAKeyParameters;
import org.bouncycastle.crypto.params.RSAPrivateCrtKeyParameters;
import org.bouncycastle.crypto.signers.DSASigner;
import org.bouncycastle.crypto.signers.PSSSigner;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.jce.provider.X509CertificateObject;

import org.bouncycastle.openssl.PEMWriter;

class client
{
  static SecureRandom secureRandom;

  static final String propertiesFile = "client.properties";
```

```
// certainty for prime generation,
// 2^-certainty probability of error
static final int certainty = 100;

static boolean debug = false;
static boolean timing = false;
static boolean generateDSAParameters = false;
static boolean generateRSAParameters = false;
static boolean renewCertificate = false;
static boolean rsa = false;

static BigInteger ZERO = BigInteger.valueOf(0);
static BigInteger ONE = BigInteger.valueOf(1);
static BigInteger TWO = BigInteger.valueOf(2);

static BigInteger DSAP, DSAQ, DSAG;
static String modulus;

public static void main(String[] args)
{
  try {

    // Start loading configuration file
    Properties myProperties = new Properties();
    myProperties.load(new FileInputStream(propertiesFile));
    String serverUrl = myProperties.getProperty("serverUrl");
    // the p in Z_p used for SPEKE, all calculations
    // are performed mod p
    int modulusLength =
      Integer.parseInt(
        myProperties.getProperty("modulusLength"));
    // bit length of random parameters (A , B) in SPEKE
    int randlength =
      Integer.parseInt(myProperties.getProperty("randLength"));
    // bit length of modulus for digital signatures
    int strength =
      Integer.parseInt(myProperties.getProperty("strength"));

    String certfile = myProperties.getProperty("certfile");
    String keyfile = myProperties.getProperty("keyfile");
    String pemcertfile = myProperties.getProperty("pemcertfile");
    String pemkeyfile = myProperties.getProperty("pemkeyfile");
    String messageFile = myProperties.getProperty("messageFile");
    String CAcertfile = myProperties.getProperty("CAcertfile");
    String domain = myProperties.getProperty("domain");
    String debugString = myProperties.getProperty("debug");
    String timingString = myProperties.getProperty("timing");
    String rsaString = myProperties.getProperty("rsa");
    String generateDSAParametersString =
```

```
  myProperties.getProperty("generateDSAParameters");
String generateRSAParametersString =
  myProperties.getProperty("generateRSAParameters");
String renewCertificateString =
  myProperties.getProperty("renewCertificate");
long renewInterval =
  Long.parseLong(myProperties.getProperty("renewInterval"));
int renewTimes =
  Integer.parseInt(myProperties.getProperty("renewTimes"));
if (modulusLength == 2048) {
  modulus = myProperties.getProperty("2048modulus");
}
else if (modulusLength == 1024) {
  modulus = myProperties.getProperty("1024modulus");
}
else {
  modulus = myProperties.getProperty("768modulus");
}

// Pre-generated DSA parameters
if (strength == 2048) {
  DSAP = new BigInteger(myProperties.getProperty("2048p"));
  DSAQ = new BigInteger(myProperties.getProperty("2048q"));
  DSAG = new BigInteger(myProperties.getProperty("2048g"));
}
else if (strength == 1024) {
  DSAP = new BigInteger(myProperties.getProperty("1024p"));
  DSAQ = new BigInteger(myProperties.getProperty("1024q"));
  DSAG = new BigInteger(myProperties.getProperty("1024g"));
}
else {
  DSAP = new BigInteger(myProperties.getProperty("768p"));
  DSAQ = new BigInteger(myProperties.getProperty("768q"));
  DSAG = new BigInteger(myProperties.getProperty("768g"));
}
if (debugString.equalsIgnoreCase("true")) {
  debug = true;
}
if (timingString.equalsIgnoreCase("true")) {
  timing = true;
}
if (generateDSAParametersString.equalsIgnoreCase("true")) {
  generateDSAParameters = true;
}
if (generateRSAParametersString.equalsIgnoreCase("true")) {
  generateRSAParameters = true;
}
if (renewCertificateString.equalsIgnoreCase("true")) {
  renewCertificate = true;
```

```
}
if (rsaString.equalsIgnoreCase("true")) {
  rsa = true;
}
// Done loading configuration file

Date startTime = new Date();
Date stopTime = new Date();;

AlgorithmIdentifier keyAlgorithmIdentifier;
AlgorithmIdentifier sigAlgorithmIdentifier;
SubjectPublicKeyInfo mySubjectPublicKeyInfo;
Provider bcProvider = new BouncyCastleProvider();
Security.addProvider(bcProvider);
secureRandom = new SecureRandom();

String username = (args.length > 0 ? args[0] : "test");
String pwd = (args.length > 1 ? args[1] : "test");
BigInteger p = new BigInteger(modulus);

// run the password through SHA-1
SHA1Digest mySHA1Digest = new SHA1Digest();
byte[] digestIn = pwd.getBytes("US-ASCII");
byte[] digestOut;
BigInteger P;
mySHA1Digest.update(digestIn , 0 , digestIn.length);
digestOut = new byte[mySHA1Digest.getDigestSize()];
mySHA1Digest.doFinal(digestOut , 0);
P = new BigInteger(1 , digestOut);
if(debug) {System.out.println("H(P) = " + P.toString());}
if(debug) {System.out.println(
  "H(P)^2 = " + P.modPow(TWO , p).toString());}

// Generate key pair pk_A, sk_A

DSAKeyPairGenerator keyGen = new DSAKeyPairGenerator();
DSAKeyGenerationParameters kgp = null;
DSAParametersGenerator dpg = null;
AsymmetricCipherKeyPair keyPair = null;
DSAPrivateKeyParameters privateKey = null;
DSAPublicKeyParameters publicKey = null;
DERInteger[] dssPublicParameters = new DERInteger[3];
RSAKeyGenerationParameters rkgp = null;;
RSAKeyPairGenerator rpg = null;
AsymmetricCipherKeyPair RSAKeyPair = null;
RSAPrivateCrtKeyParameters privateKeyRSA = null;
RSAKeyParameters publicKeyRSA = null;
DERInteger[] rsaPublicParameters = new DERInteger[2];
DERSequence myDERSequence = null;
```

```
DSASigner myDSASigner = new DSASigner();
// 20 = length of salt. 20 bytes = 160 bits, the
// length of the output from SHA-1
PSSSigner myRSASigner =
  new PSSSigner( new RSAEngine() , new SHA1Digest() , 20);
if(rsa) {
  // Client will use RSA
  if(debug) {System.out.print(
    "Generating RSA key pair...");}
  if(timing) {startTime = new Date();}
  rkgp = new RSAKeyGenerationParameters(
    new BigInteger("65537"),
    secureRandom,
    strength,
    certainty) ;
  rpg = new RSAKeyPairGenerator();
  rpg.init(rkgp);
  RSAKeyPair = rpg.generateKeyPair();
  if(timing) {stopTime = new Date();}
  if(debug) {System.out.println("OK");}
  if(timing) {System.out.println(
    "RSA key generation took " +
    (stopTime.getTime() - startTime.getTime()) +
    " milliseconds.");}
  privateKeyRSA =
    (RSAPrivateCrtKeyParameters)RSAKeyPair.getPrivate();
  publicKeyRSA = (RSAKeyParameters)RSAKeyPair.getPublic();
  rsaPublicParameters[0] =
    new DERInteger(publicKeyRSA.getModulus());
  rsaPublicParameters[1] =
    new DERInteger(publicKeyRSA.getExponent());
  myDERSequence = new DERSequence(rsaPublicParameters);

  // rsaEncryption
  keyAlgorithmIdentifier =
    new AlgorithmIdentifier(
      new DERObjectIdentifier("1.2.840.113549.1.1.1"),
      new DERNull()
    );

  // sha-1WithRSAEncryption
  sigAlgorithmIdentifier =
    new AlgorithmIdentifier(
      new DERObjectIdentifier("1.2.840.113549.1.1.5"),
      new DERNull()
    );
  mySubjectPublicKeyInfo =
    new SubjectPublicKeyInfo(
      keyAlgorithmIdentifier,
```

```
      myDERSequence
    );
}
else{
  // Client will use DSA
  if(generateDSAParameters) {
    // Generate DSA parameters instead of
    // using those in the properties file
    // This takes a long time if we are
    // using a large modulus
    if(debug) {System.out.print(
      "Generating DSA parameters...");}
    if(timing) {startTime = new Date();}
    dpg = new DSAParametersGenerator();
    dpg.init(strength , certainty , secureRandom);
    kgp = new DSAKeyGenerationParameters(
      secureRandom, dpg.generateParameters());
    if(timing) {stopTime = new Date();}
    if(debug) {System.out.println("OK");}
    if(timing) {System.out.println(
      "DSA parameter generation took " +
      (stopTime.getTime() - startTime.getTime()) +
      " milliseconds.");}
    if (debug) {System.out.println(
      strength + "-bit P: " +
      kgp.getParameters().getP().toString());}
    if (debug) {System.out.println(
      "Q: " + kgp.getParameters().getQ().toString());}
    if (debug) {System.out.println(
      "G: " + kgp.getParameters().getG().toString());}
  }
  else {
    // Load key generation parameters from file
    kgp = new DSAKeyGenerationParameters(
      secureRandom , new DSAParameters(DSAP, DSAQ, DSAG));
  }
  if(debug) {System.out.print(
    "Generating DSA key pair...");}
  if(timing) {startTime = new Date();}
  keyGen.init(kgp);
  keyPair = keyGen.generateKeyPair();
  if(timing) {stopTime = new Date();}
  privateKey =
    (DSAPrivateKeyParameters)keyPair.getPrivate();
  publicKey =
    (DSAPublicKeyParameters)keyPair.getPublic();
  if(debug) {System.out.println("OK");}
  if(timing) {System.out.println(
    "DSA key generation took " +
```

```
    (stopTime.getTime() - startTime.getTime()) +
    " milliseconds.");}
  // Include p,q,g as per 7.3.3 in rfc2459.
  dssPublicParameters[0] =
    new DERInteger(publicKey.getParameters().getP());
  dssPublicParameters[1] =
    new DERInteger(publicKey.getParameters().getQ());
  dssPublicParameters[2] =
    new DERInteger(publicKey.getParameters().getG());
  // dsa
  keyAlgorithmIdentifier =
    new AlgorithmIdentifier(
      new DERObjectIdentifier("1.2.840.10040.4.1"),
      new DERSequence(dssPublicParameters)
    );
  // dsa-with-sha1
  sigAlgorithmIdentifier =
    new AlgorithmIdentifier(
      new DERObjectIdentifier("1.2.840.10040.4.3"),
      new DERSequence(dssPublicParameters)
    );
  // use publicKey.getY() to retrieve the public key
  // as a BigInteger
  mySubjectPublicKeyInfo =
    new SubjectPublicKeyInfo(
      keyAlgorithmIdentifier,
      new DERInteger(publicKey.getY())
    );
}

// Generate random value RA

BigInteger RA = new BigInteger(randlength , secureRandom);

// Compute SPEKE verifier H(password)^{2*A}

BigInteger ha = P.modPow(RA.multiply(TWO) , p);
if(debug) {System.out.println(
  "H(P)^(2RA) = " + ha.toString());}

// Construct CertTemplate
X509Name myX509Name =
  new X509Name(domain+",uid="+username);

CertTemplate myCertTemplate = new CertTemplate();
// version=2 -> X.509v3
myCertTemplate.setVersion(new DERInteger(2));
myCertTemplate.setSubject(myX509Name);
myCertTemplate.setPublicKey(mySubjectPublicKeyInfo);
```

```
CertRequest myCertRequest =
  new CertRequest(
    new DERInteger(new BigInteger(128 , secureRandom)),
    myCertTemplate
  );

// SPEKE public key

// Construct POP
// IMPORTANT: we use POP for authentication, using SPEKE.
// The certificate will be encrypted with the key
// derived from the SPEKE

// POPOPrivKey(DERInteger(0), 1)
// 1 = subsequentMessage, 0 = encrCert
POPOPrivKey myPOPOPrivKey =
  new POPOPrivKey(new DERInteger(0) , 1);
// 2 = keyEncipherment
ProofOfPossession myProofOfPossession =
  new ProofOfPossession(myPOPOPrivKey , 2);

// Construct CertReqMessage

CertReqMsg myCertReqMsg = new CertReqMsg(myCertRequest);
myCertReqMsg.setPop(myProofOfPossession);

// Add CertReqMessage to CertReqMessages

CertReqMessages myCertReqMessages =
  new CertReqMessages(myCertReqMsg);

// Construct PKIHeader (with SPEKE parameters)

BigInteger myNonce = new BigInteger(128 , secureRandom);
PKIHeader myPKIHeader =
  new PKIHeader(
    new DERInteger(1),
    new GeneralName(myX509Name),
    new GeneralName(new X509Name("dc=no,dc=ntnu,cn=ca"))
  );
// We use SenderKID to store the SPEKE public key
myPKIHeader.setSenderKID(
  new DEROctetString(ha.toByteArray()));
myPKIHeader.setMessageTime(
  new DERGeneralizedTime(new Date()));
myPKIHeader.setSenderNonce(
  new DEROctetString(myNonce.toByteArray()));
```

```
// Construct PKIMessage
// 2 = Certification Request
PKIBody myPKIBody = new PKIBody(myCertReqMessages, 2);
PKIMessage myPKIMessage =
  new PKIMessage(myPKIHeader, myPKIBody);

// Write the message to file, to enable stress test
DEROutputStream mFile =
  new DEROutputStream(new FileOutputStream(messageFile));
mFile.writeObject(myPKIMessage);
mFile.close();

URL serverURL = new URL(serverUrl);
URLConnection serverConnection =
  serverURL.openConnection();
serverConnection.setRequestProperty(
  "Content-Type", "application/pkixcmp");
serverConnection.setUseCaches(false);
serverConnection.setDoOutput(true);
serverConnection.setDoInput(true);
serverConnection.connect();

// Send PKIMessage to server, receive CertRep or Error
DEROutputStream dRequestData =
  new DEROutputStream(serverConnection.getOutputStream());
dRequestData.writeObject(myPKIMessage);
dRequestData.close();

myPKIMessage =
  PKIMessage.getInstance(
    new ASN1InputStream(
      serverConnection.getInputStream()
    ).readObject()
  );


BigInteger hb =
  new BigInteger(
    myPKIMessage.getHeader().getSenderKID().getOctets()
  );
// SPEKE constraint
if (hb.compareTo(ZERO) == 0)
  {throw new Exception("g^R_B is 0, exiting.");}

if(debug) {System.out.println(
  "H(P)^(2RB) = " + hb.toString());}
// Compute K, verify that K != 1
BigInteger KA = hb.modPow(RA.multiply(TWO) , p);
if(debug) {System.out.println(
```

```
  "H(P)^(4RA*RB) = " + KA.toString());}
// SPEKE constraint
if (KA.compareTo(ONE) == 0)
  {throw new Exception("K is 1, exiting.");}

// K_S, the shared session key, is a hash of K,
// to provide forward secrecy
mySHA1Digest = new SHA1Digest();
mySHA1Digest.update(KA.toByteArray(),
                    0,
                    KA.toByteArray().length);
byte[] K_S = new byte[mySHA1Digest.getDigestSize()];
mySHA1Digest.doFinal(K_S , 0);

// Decrypt certificate with K_S
SecretKeySpec myAESKeySpec =
  new SecretKeySpec(K_S , 0 , 16 , "AES");
Cipher myCipher =
  Cipher.getInstance("AES/CBC/PKCS7Padding" , bcProvider);

byte[] myAESIV =
  myPKIMessage.getBody().getCp().getResponse(0).
  getCertifiedKeyPair().getCertOrEncCert().
  getEncryptedCert().getEncSymmKey().getBytes();

myCipher.init(Cipher.DECRYPT_MODE,
              myAESKeySpec,
              new IvParameterSpec(myAESIV));

byte[] myDecryptedCert =
  myCipher.doFinal(
    myPKIMessage.getBody().getCp().getResponse(0).
    getCertifiedKeyPair().getCertOrEncCert().
    getEncryptedCert().getEncValue().getBytes()
  );

X509CertificateStructure myCert =
  X509CertificateStructure.getInstance(
    new ASN1InputStream(myDecryptedCert).readObject()
  );

// Try to read CA root certificate from file. If that fails,
// decrypt the CA root certificate from the Certification Response
X509CertificateStructure caCert;
try{
  caCert =
    X509CertificateStructure.getInstance(
      new ASN1InputStream(
        new FileInputStream(CAcertfile)
```

```
      ).readObject()
    );
}
catch(Exception e) {
  if(debug) {System.out.println(
    "Using encrypted CA root certificate" +
    "from Certification Response.");}

  myAESIV =
    myPKIMessage.getBody().getCp().getResponse(1).
    getCertifiedKeyPair().getCertOrEncCert().
    getEncryptedCert().getEncSymmKey().getBytes();

  myCipher.init(Cipher.DECRYPT_MODE,
                myAESKeySpec,
                new IvParameterSpec(myAESIV));

  myDecryptedCert =
    myCipher.doFinal(
      myPKIMessage.getBody().getCp().getResponse(1).
      getCertifiedKeyPair().getCertOrEncCert().
      getEncryptedCert().getEncValue().getBytes()
    );

  caCert =
    X509CertificateStructure.getInstance(
      new ASN1InputStream(myDecryptedCert).readObject()
    );
}

// Check CA signature on CA certificate
if (checkSignature(myRSASigner,
                   myDSASigner,
                   caCert,
                   caCert.getSignature().getBytes(),
                   caCert.getTBSCertificate().getEncoded(),
                   caCert.getSignatureAlgorithm()))
{
  if(debug) {System.out.println(
    "CA self-signature verified!");}
} else {
  if(debug) {System.out.println(
    "CA self-signature verification FAILED!");}
  throw new Exception("CA self-signature invalid.");
}

// Check CA signature on certificate
if (checkSignature(myRSASigner,
                   myDSASigner,
```

```
                    caCert,
                    myCert.getSignature().getBytes(),
                    myCert.getTBSCertificate().getEncoded(),
                    myCert.getSignatureAlgorithm()))
{
  if(debug) {System.out.println(
    "CA signature verified!");}
} else {
  if(debug) {System.out.println(
    "CA signature verification FAILED!");}
  throw new Exception("CA signature invalid.");
}

// Must check contents of certificate!
// An attacker could have modified the Certification Request
// Crude check, but the most important thing is to check
// the public key
// TODO: check exponent as well
if(rsa){
  if(myCert.getSubject().toString().equals(
       myX509Name.toString()
     )
     &&
     ((DERInteger)
       ((DERSequence)myCert.getSubjectPublicKeyInfo().
         getPublicKey()
       ).getObjectAt(0)
     ).getValue().compareTo(publicKeyRSA.getModulus()) == 0)
  {
    if(debug) {System.out.println(
      "Certificate contains correct RSA public " +
      "key and name.");}
  }
  else {
    throw new Exception("Certificate has invalid data!");
  }

}
else{
  if(myCert.getSubject().toString().equals(
       myX509Name.toString()
     )
     &&
     ((DERInteger)
       myCert.getSubjectPublicKeyInfo().getPublicKey()
     ).getValue().compareTo(publicKey.getY()) == 0)
  {
    if(debug) {System.out.println(
      "Certificate contains correct DSA public " +
```

```
      "key and name.");}
  }
  else {
    throw new Exception("Certificate has invalid data!");
  }
}
// Store private key + cert
PEMWriter pemCertFile =
  new PEMWriter(new FileWriter(pemcertfile));
pemCertFile.writeObject(new X509CertificateObject(myCert));
pemCertFile.close();

DEROutputStream certFile =
  new DEROutputStream(new FileOutputStream(certfile));
certFile.writeObject(myCert);
certFile.close();


KeyFactory myKeyFactory;
PrivateKey myPemPrivateKey;
DEROutputStream keyFile;
if(rsa){
  myKeyFactory = KeyFactory.getInstance("RSA");
  myPemPrivateKey =
    myKeyFactory.generatePrivate(
      new RSAPrivateCrtKeySpec(
        privateKeyRSA.getModulus(),
        privateKeyRSA.getPublicExponent(),
        privateKeyRSA.getExponent(),
        privateKeyRSA.getP(),
        privateKeyRSA.getQ(),
        privateKeyRSA.getDP(),
        privateKeyRSA.getDQ(),
        privateKeyRSA.getQInv()
      )
    );

} else {
  myKeyFactory = KeyFactory.getInstance("DSA");
  myPemPrivateKey =
    myKeyFactory.generatePrivate(
      new DSAPrivateKeySpec(
        privateKey.getX(),
        privateKey.getParameters().getP(),
        privateKey.getParameters().getQ(),
        privateKey.getParameters().getG()
      )
    );
```

```
  keyFile =
    new DEROutputStream(new FileOutputStream(keyfile));
  keyFile.writeObject(new DERInteger(privateKey.getX()));
  keyFile.close();
}

PEMWriter pemKeyFile =
  new PEMWriter(new FileWriter(pemkeyfile));
pemKeyFile.writeObject(myPemPrivateKey);
pemKeyFile.close();


// Renew cert
if(renewCertificate) {
  BigInteger[] mySignature = new BigInteger[2];
  DERBitString myDERSignature;
  DERInteger[] myDERIntegerArray = new DERInteger[2];
  DSAPrivateKeyParameters newPrivateKey = null;
  DSAPublicKeyParameters newPublicKey = null;
  X509CertificateStructure myNewCert;
  PSSSigner myPSSSigner;
  RSAPrivateCrtKeyParameters newPrivateKeyRSA = null;
  RSAKeyParameters newPublicKeyRSA = null;

  for (int i = 1 ; i <= renewTimes ; i++) {

    Thread.sleep(renewInterval);
    if(debug) {System.out.println(
      "Renewing certificate...");}

    // Generate new key pair
    if(rsa) {
      if(debug) {System.out.print(
        "Generating RSA key pair...");}
      if(timing) {startTime = new Date();}
      RSAKeyPair = rpg.generateKeyPair();
      if(timing) {stopTime = new Date();}
      if(debug) {System.out.println("OK");}
      if(timing) {System.out.println(
        "RSA key generation took " +
        (stopTime.getTime() - startTime.getTime()) +
        " milliseconds.");}
      newPrivateKeyRSA =
        (RSAPrivateCrtKeyParameters)RSAKeyPair.getPrivate();
      newPublicKeyRSA =
        (RSAKeyParameters)RSAKeyPair.getPublic();
      rsaPublicParameters[0] =
        new DERInteger(newPublicKeyRSA.getModulus());
      rsaPublicParameters[1] =
```

```
    new DERInteger(newPublicKeyRSA.getExponent());
  myDERSequence =
    new DERSequence(rsaPublicParameters);

  mySubjectPublicKeyInfo =
    new SubjectPublicKeyInfo(keyAlgorithmIdentifier,
                             myDERSequence);

}
else{
  if(debug) {System.out.print(
    "Generating key pair...");}
  if(timing) {startTime = new Date();}
  keyPair = keyGen.generateKeyPair();
  if(timing) {stopTime = new Date();}
  // The old key pair is privateKey, publicKey
  newPrivateKey =
    (DSAPrivateKeyParameters)keyPair.getPrivate();
  newPublicKey =
    (DSAPublicKeyParameters)keyPair.getPublic();
  if(debug) {System.out.println("OK");}
  if(timing) {System.out.println(
    "Key generation took " +
    (stopTime.getTime() - startTime.getTime()) +
    " milliseconds.");}
  dssPublicParameters[0] =
    new DERInteger(
      newPublicKey.getParameters().getP());
  dssPublicParameters[1] =
    new DERInteger(
      newPublicKey.getParameters().getQ());
  dssPublicParameters[2] =
    new DERInteger(
      newPublicKey.getParameters().getG());

  mySubjectPublicKeyInfo =
    new SubjectPublicKeyInfo(
      keyAlgorithmIdentifier,
      new DERInteger(newPublicKey.getY())
    );
}

// Construct new CertTemplate
myCertTemplate = new CertTemplate();
// version=2 -> X.509v3
myCertTemplate.setVersion(new DERInteger(2));
myCertTemplate.setSubject(
  new X509Name(domain+",uid="+username));
myCertTemplate.setPublicKey(mySubjectPublicKeyInfo);
```

```
myCertRequest =
  new CertRequest(
    new DERInteger(new BigInteger(128 , secureRandom)),
    myCertTemplate
  );

myCertReqMsg = new CertReqMsg(myCertRequest);

// Add CertReqMessage to CertReqMessages

myCertReqMessages = new CertReqMessages(myCertReqMsg);

myNonce = new BigInteger(128 , secureRandom);

myPKIHeader =
  new PKIHeader(new DERInteger(1),
                new GeneralName(
                  new X509Name(domain+",cn="+username)
                ),
                new GeneralName(
                  new X509Name("dc=no,dc=ntnu,cn=ca")
                ));
myPKIHeader.setMessageTime(
  new DERGeneralizedTime(new Date()));
myPKIHeader.setSenderNonce(
  new DEROctetString(myNonce.toByteArray()));

myPKIHeader.setProtectionAlg(sigAlgorithmIdentifier);

// Construct PKIMessage
// 2 = Certification Request
myPKIBody = new PKIBody(myCertReqMessages, 2);
myPKIMessage = new PKIMessage(myPKIHeader, myPKIBody);

// Add old certificate as extraCert
myPKIMessage.addExtraCert(myCert);

// Add protection to PKIMessage

digestIn = myPKIMessage.getProtectedBytes();
if(rsa) {
  myPSSSigner =
    new PSSSigner(new RSAEngine(),
                  new SHA1Digest(),
                  20);
  myPSSSigner.init(true, privateKeyRSA);
  myPSSSigner.update(digestIn , 0 , digestIn.length);
  myDERSignature =
```

```
    new DERBitString(myPSSSigner.generateSignature());
}
else{
  mySHA1Digest = new SHA1Digest();
  mySHA1Digest.update(digestIn , 0 , digestIn.length);
  digestOut = new byte[mySHA1Digest.getDigestSize()];
  mySHA1Digest.doFinal(digestOut , 0);

  myDSASigner.init(true , privateKey);
  mySignature =
    myDSASigner.generateSignature(digestOut);
  myDERIntegerArray[0] = new DERInteger(mySignature[0]);
  myDERIntegerArray[1] = new DERInteger(mySignature[1]);

  myDERSignature =
    new DERBitString(
      new DERSequence(myDERIntegerArray));
}
myPKIMessage.setProtection(myDERSignature);

// Write the message to file, to enable stress test
mFile =
  new DEROutputStream(
    new FileOutputStream(messageFile));
mFile.writeObject(myPKIMessage);
mFile.close();

serverConnection = serverURL.openConnection();
serverConnection.setRequestProperty(
  "Content-Type", "application/pkixcmp");
serverConnection.setUseCaches(false);
serverConnection.setDoOutput(true);
serverConnection.setDoInput(true);
serverConnection.connect();

// Send PKIMessage to server,
// receive CertRep or Error
dRequestData =
  new DEROutputStream(
    serverConnection.getOutputStream());
dRequestData.writeObject(myPKIMessage);
dRequestData.close();

myPKIMessage =
  PKIMessage.getInstance(
    new ASN1InputStream(
      serverConnection.getInputStream()
    ).readObject()
  );
```

```
myNewCert =
  myPKIMessage.getBody().getCp().getResponse(0).
  getCertifiedKeyPair().getCertOrEncCert().
  getCertificate();

// Check CA signature on certificate
if (checkSignature(myRSASigner,
          myDSASigner,
          caCert,
          myNewCert.getSignature().getBytes(),
          myNewCert.getTBSCertificate().getEncoded(),
          myNewCert.getSignatureAlgorithm()))
{
  if(debug) {System.out.println(
    "CA signature verified!");}
} else {
  if(debug) {System.out.println(
    "CA signature verification FAILED!");}
  throw new Exception("CA signature invalid.");
}


// We should check the certificate contents
// here as well, but it is not quite as critical
// due to the message protection of the Certification
// Request
if(rsa){
  if(myNewCert.getSubject().toString().equals(
       myX509Name.toString()
     )
     &&
     ((DERInteger)
       ((DERSequence)myNewCert.
       getSubjectPublicKeyInfo().getPublicKey()).
       getObjectAt(0)
     ).getValue().compareTo(
       newPublicKeyRSA.getModulus()) == 0)
  {
    if(debug) {System.out.println(
      "Certificate contains correct RSA public " +
      "key and name.");}
  }
  else {
    throw new Exception(
      "Certificate has invalid data!");
  }

}
else{
```

```
  if(myNewCert.getSubject().toString().equals(
       myX509Name.toString()
     )
     &&
     ((DERInteger)
       myNewCert.getSubjectPublicKeyInfo().
       getPublicKey()
     ).getValue().compareTo(
       newPublicKey.getY()) == 0)
  {
    if(debug) {System.out.println(
      "Certificate contains correct DSA public " +
      "key and name.");}
  }
  else {
    throw new Exception(
      "Certificate has invalid data!");
  }
}

// If we got this far, then the update went well.
// Start using the new certificate.
if(rsa){
  privateKeyRSA = newPrivateKeyRSA;
  publicKeyRSA = newPublicKeyRSA;
}
else {
  privateKey = newPrivateKey;
  publicKey = newPublicKey;
}
myCert = myNewCert;

// Store private key + cert
pemCertFile =
  new PEMWriter(new FileWriter(pemcertfile));
pemCertFile.writeObject(
  new X509CertificateObject(myCert));
pemCertFile.close();

certFile =
  new DEROutputStream(new FileOutputStream(certfile));
certFile.writeObject(myCert);
certFile.close();

if(rsa){
  myPemPrivateKey =
    myKeyFactory.generatePrivate(
      new RSAPrivateCrtKeySpec(
        privateKeyRSA.getModulus(),
```

```
                privateKeyRSA.getPublicExponent(),
                privateKeyRSA.getExponent(),
                privateKeyRSA.getP(),
                privateKeyRSA.getQ(),
                privateKeyRSA.getDP(),
                privateKeyRSA.getDQ(),
                privateKeyRSA.getQInv()
              )
            );

        } else {
          myPemPrivateKey =
            myKeyFactory.generatePrivate(
              new DSAPrivateKeySpec(
                privateKey.getX(),
                privateKey.getParameters().getP(),
                privateKey.getParameters().getQ(),
                privateKey.getParameters().getG()
              )
            );

          keyFile =
            new DEROutputStream(new FileOutputStream(keyfile));
          keyFile.writeObject(
            new DERInteger(privateKey.getX()));
          keyFile.close();
        }
        pemKeyFile =
          new PEMWriter(new FileWriter(pemkeyfile));
        pemKeyFile.writeObject(myPemPrivateKey);
        pemKeyFile.close();

      }
    } // end if(renewCertificate)
  } // end try

  catch (Exception e) {
    e.printStackTrace();
  } // end catch


} // end main

private static boolean checkSignature(
  PSSSigner myRSASigner,
  DSASigner myDSASigner,
  X509CertificateStructure inCert,
  byte[] sig,
  byte[] message,
```

```
  AlgorithmIdentifier sigAlgId)
  throws CryptoException , IOException
{

  SubjectPublicKeyInfo clientSubjectPublicKeyInfo =
    inCert.getSubjectPublicKeyInfo();
  AlgorithmIdentifier pubKeyAlgId =
    clientSubjectPublicKeyInfo.getAlgorithmId();

  // OID 1.2.840.113549.1.1.5 = sha-1WithRSAEncryption
  if(sigAlgId.getObjectId().getId().equals(
    "1.2.840.113549.1.1.5"))
  {
    if(debug) {System.out.println(
      "Checking RSA signature...");}
    DERSequence RSADERSequence =
      (DERSequence)clientSubjectPublicKeyInfo.getPublicKey();

    myRSASigner.init(false,
      new RSAKeyParameters(
        false,
        ((DERInteger)RSADERSequence.getObjectAt(0)).getValue(),
        ((DERInteger)RSADERSequence.getObjectAt(1)).getValue()
      )
    );
    myRSASigner.update(message , 0 , message.length);
    return myRSASigner.verifySignature(sig);
  }
  else{
    if(debug) {System.out.println(
      "Checking DSA signature...");}
    DERSequence sigDERSequence, verDERSequence;
    DERInteger sigR, sigS, DERP, DERQ, DERR;
    ASN1InputStream ais = new ASN1InputStream(sig);
    sigDERSequence = (DERSequence)ais.readObject();
    sigR = (DERInteger)sigDERSequence.getObjectAt(0);
    sigS = (DERInteger)sigDERSequence.getObjectAt(1);
    // this sequence consists of 3 Integers: p, q and g.
    verDERSequence = (DERSequence)sigAlgId.getParameters();
    DERP = (DERInteger)verDERSequence.getObjectAt(0);
    DERQ = (DERInteger)verDERSequence.getObjectAt(1);
    DERR = (DERInteger)verDERSequence.getObjectAt(2);

    DSAPublicKeyParameters publicKey =
      new DSAPublicKeyParameters(
        ((DERInteger)clientSubjectPublicKeyInfo.
          getPublicKey()).getValue(),
        new DSAParameters(DERP.getValue(),
                          DERQ.getValue(),
```

```
                          DERR.getValue())
        );

      SHA1Digest mySHA1Digest = new SHA1Digest();
      mySHA1Digest.update(message , 0 , message.length);
      byte[] digestOut = new byte[mySHA1Digest.getDigestSize()];
      mySHA1Digest.doFinal(digestOut , 0);
      myDSASigner.init(false , publicKey);
      return myDSASigner.verifySignature(digestOut,
                                         sigR.getValue(),
                                         sigS.getValue());

    }
  }// end method checkSignature
} // end class client
```

Appendix J

AS/CA configuration file

```
# Bit length of SPEKE modulus
#modulusLength=768
modulusLength=1024
#modulusLength=2048

# Length of random parameters R_C and R_CA
#randLength=256
randLength=336
#randLength=336

# Bit length of modulus for CA signatures
#strength=768
strength=1024
#strength=2048

# Client certificate lifetime in milliseconds
validityPeriod=3600000

# Validity of CA certificate, 1 year
CAvalidityPeriod=31536000000

# Should we use RSA instead of DSA for CA signatures?
rsa=false

domain=dc=no,dc=ntnu,ou=people
debug=true
timing=true

# 768 bit SPEKE modulus
768modulus=10219182700471792528074903142209730986025741205156098642305686012359098230342508321545017947423837486501347485903017716296524640521478880949455713848752851363189893414165280240385170707274920304827309332202930258505165707826361219870
```

```
# 1024 bit SPEKE modulus
1024modulus=13922312061313283396673007316069242814270243018093605148097769751041479404686564484319758024870299937171858242909991917509954854454739964165543404935059901895283081432044525907948952672855519722025153264922554730100664527930501168640253722520088121514677203680367236993485555038241086227016959510508139988940 3

# 2048 bit SPEKE modulus
2048modulus=28394547994823715564799156911720245722292026507913208372892676049909852921998124767262631291053515789947269933732483532726274257711141341842806259014448879345475422203139439087892831618773015556344600056119245090741319734087440439422266778074411549698891036722466262048294270259734016594455919646908239667679758638472769761124911495062483656256482325620463800183739615646865943609400410423394064921157047924899616378771656408403850942044361012198741695055017089756995337800182105874874642885756944569723590974491758952236377080209922111882907646342183348651486796045896416705675891978704211218928300281879580628474393
```

Appendix K

Client configuration file

```
# URL to server
serverUrl=http://127.0.0.1:8080/sso

# Bit length of SPEKE modulus p
#modulusLength=768
modulusLength=1024
#modulusLength=2048

# Bit length of random parameters R_C and R_CA
#randLength=256
randLength=336

# Bit length of modulus for client signatures
#strength=768
strength=1024
#strength=2048

# Should we generate DSA parameters?
generateDSAParameters=false

# Should we use RSA instead of DSA?
rsa=true

# Should we renew the certificate?
renewCertificate=true

# How often should we renew the certificate (milliseconds)?
renewInterval=1800000

# How many times should we renew the certificate?
renewTimes=10

# Hardcoded DSA parameters
```

```
# 768 bit modulus DSA parameters
768p=123675228593721293574237947732471722740289097240703569274339690250766568418263320309556459296330636016210387606740415097066797837494247763424448964225639516271742488784650146649255784185945366900807460298520877105371219926625297689l

768q=1456260988372919293152728381233525829989072028861

768g=258250327898488641613356475695325815411600655252492035833256747936409018053338546572036872717813522979378340537349496140965216450290540558646793109886209015016130727657495269746618284801929597613696835114322467425544950483511605630

# 1024 bit modulus DSA parameters
1024p=100605512138451092793937890202301370956887271863319499037004804742992234725416797556454844267254878766116695700643012828523475874230713873425486842563918613396951140442092893940789848428573528095351693736627556556567278107612595031489803773899152425678753482400601170432480044402279696734231955659569557232265 7

1024q=974919424217843684803141841243633253920568455477

1024g=209174719312300984079660269399880155742384657795411567305321826810928169251548722459259276171211112243607102663985586792510118829018669911990986043024306266279144818307889539059428222718656947570030344862178740599775246659932609954018069959678489239778714617794030906382853683199517853999572112929137747151837

# 2048 bit modulus DSA parameters
2048p=285240074584526653713893987977601501647297087620068032929403687379870525204290355776523816976070999155008978425007891860274113317090495765566378837783604667069380381484998149435876653241000573363494191187489789710177348938765822806828106868782455975896888565311232948246439501088789018304935092979710509748610554278715333764652162673194364029958946239990924968935325054520346419291247454515966999431458030074348102727649675101785155595767594056131570317707536166002299911237974408820907078029711079650329530947752874087710183368781087955974826066627691241943356909161783072280840946657913259479512776151322066830578691

2048q=864616035198890466891554873473834366078831290793

2048g=953472940613863680743088133214405131161116890074892311928699922313332648783818534248869350327508074386493265626186177924006221274743832946534916755739661845756576229012444298731272197699094275555660966483402263537097528125895263192915051015427314237652507018608590624928892015205147233035409843223380117710877446734509833959632519576648919561452953136208246330391440696715933240111357421378719290090948374402402723753172424046320991015944415416940125476942
```

```
107237921138499273917672818589970549314673688435334446028881145909
968192838675116526518163689695445110864034901034810275070070415373
021375944174662675821262382 8

# Files used to save keys and certificates
certfile=client.crt
keyfile=client.key
pemcertfile=client_crt.pem
pemkeyfile=client_key.pem

# Uncomment one of these to use a CA certificate distributed
# out-of-band
#CAcertfile=ca_768.crt
#CAcertfile=ca_1024.crt
#CAcertfile=ca_2048.crt

# Filename for PKIMessage used for stress tests
messageFile=certreq.der

domain=dc=no,dc=ntnu,ou=people
debug=true
timing=true

# 768 bit SPEKE modulus
768modulus=102191827004717925280749031422097309860257412051560986 4
230568601235909823034250832154501794742383748650134748590301771629
652464052147888094945571384875285136318989341416528024038517070727
492030482730933220293025850516570782636121987

# 1024 bit SPEKE modulus
1024modulus=139223120613132833966730073160692428142702430180936051
480977697510414794046865644843197580248702999371718582429099919175
099548544547399641655434049350599018952830814320445259079489526728
555197220251532649225547301006645279305011686402537225200881215146
772036803672369934855550382410862270169595105081399889403

# 2048 bit SPEKE modulus
2048modulus=283945479948237155647991569117202457222920265079132083
728926760499098529219981247672626312910535157899472699337324835327
262742577111413418428062590144488793454754222031394390878928316187
730177556344600056119245090741319734087440439422266778074411549698
891036722466262048294270259734016594455919646908239667679758638472
769761124911495062483656256482325620463800183739615646865943609400
410423394064921157047924899616378771656408403850942044361012198741
695055017089756995337800182105874874642885756944569723590974491758
952236377080209922111882907646342183348651486796045896416705675891
978704211218928300281879580628474 39
```

Appendix L

Stress test shell script

```
#!/bin/bash
JAVA_HOME=/usr/local/jdk1.5.0_02
PATH=$JAVA_HOME/bin:$PATH
CLASSPATH=novosec-bc-ext/novosec_cmp:\
          crypto-127/jars/bcprov-jdk15-127.jar:\
          crypto-127/jars/bctest-jdk15-127.jar:\
          crypto-127/jars/bctsp-jdk15-127.jar:\
          classes
STARTTIME=‘date +"%s%N"‘
LOGFILE=performance/new/stresstest_$STARTTIME.log
touch $LOGFILE
echo "First, generate new PKIMessage..."
java -classpath $CLASSPATH no.ntnu.item.ttm4900.client eian test123
echo "Starting stress test..."
while true
do
if test ‘ps -ef|grep no.ntnu.item.ttm4900.stresstest|grep java|wc -l‘ \
-lt $1
  then java -classpath $CLASSPATH no.ntnu.item.ttm4900.stresstest \
eian test123 10000 >> $LOGFILE & else java -classpath $CLASSPATH \
no.ntnu.item.ttm4900.stresstest eian test123 1 >> $LOGFILE ; fi
date >> $LOGFILE
sleep 60
done
```

Appendix M

Java stress test classes

stresstest.java is used for performance testing of the initial issuance of certificated using SPEKE authentication:

```
// SPEKE/CMP on-the-fly SSO implementation
//
// Author: Martin Eian
// This class is used to stress test initial issuance by the server.
// The client class must be run once first, to write a PKIMessage
// to file that can be loaded by the stresstest.

package no.ntnu.item.ttm4900;

import java.io.FileInputStream;
import java.io.OutputStream;

import java.math.BigInteger;

import java.net.URLConnection;
import java.net.URL;

import java.security.Provider;
import java.security.SecureRandom;
import java.security.Security;

import java.util.Date;
import java.util.Properties;

import org.bouncycastle.asn1.ASN1InputStream;
import org.bouncycastle.asn1.DEROctetString;
import org.bouncycastle.asn1.DEROutputStream;

import org.bouncycastle.jce.provider.BouncyCastleProvider;

import com.novosec.pkix.asn1.cmp.*;
```

```
import com.novosec.pkix.asn1.crmf.*;

class stresstest
{
  static final boolean debug = false;
  static final boolean timing = true;
  static final String propertiesFile = "client.properties";
  // SHA-1 of the password 'test123'
  static BigInteger P = new
  BigInteger("65387856594671371314962910427547810457186772780 4");
  static BigInteger TWO = BigInteger.valueOf(2);

  public static void main(String[] args)
  {
    try {
      Properties myProperties = new Properties();
      myProperties.load(new FileInputStream(propertiesFile));
      String serverUrl = myProperties.getProperty("serverUrl");
      // the p in Z_p used for SPEKE, all calculations are
      // performed mod p
      int modulusLength =
        Integer.parseInt(myProperties.getProperty("modulusLength"));
      // bit length of random parameters (A , B) in SPEKE
      int randlength =
        Integer.parseInt(myProperties.getProperty("randLength"));
      String modulus;
      if (modulusLength == 2048) {
        modulus = myProperties.getProperty("2048modulus");
      }
      else if (modulusLength == 1024) {
        modulus = myProperties.getProperty("1024modulus");
      }
      else {
        modulus = myProperties.getProperty("768modulus");
      }
      BigInteger p = new BigInteger(modulus);

      Provider bcProvider = new BouncyCastleProvider();
      Security.addProvider(bcProvider);
      SecureRandom secureRandom = new SecureRandom();

      String username = (args.length > 0 ? args[0] : "test");
      String pwd = (args.length > 1 ? args[1] : "test");
      // How many requests to make during stress test
      String requests = (args.length > 1 ? args[2] : "1");
      int numberRequests = Integer.parseInt(requests);

      Date startTime, stopTime;
      PKIMessage myPKIMessage =
```

```
  PKIMessage.getInstance(
    new ASN1InputStream(
      new FileInputStream("certreq.der")).readObject());

PKIHeader myPKIHeader = myPKIMessage.getHeader();
PKIBody myPKIBody = myPKIMessage.getBody();

BigInteger RA, ha;

URL serverURL = new URL(serverUrl);
URLConnection serverConnection = serverURL.openConnection();
if(timing) {startTime = new Date();}

int errors = 0;
for ( int i = 1 ; i <= numberRequests ; i++ ) {
  try{

    // Generate new SPEKE public key for each request to
    // avoid caching on the server
    RA = new BigInteger(randlength , secureRandom);
    ha = P.modPow(RA.multiply(TWO) , p);
    myPKIHeader.setSenderKID(new DEROctetString(ha.toByteArray()));
    myPKIMessage = new PKIMessage(myPKIHeader, myPKIBody);

    serverConnection = serverURL.openConnection();
    serverConnection.setRequestProperty("Content-Type",
      "application/pkixcmp");
    serverConnection.setUseCaches(false);
    serverConnection.setDoOutput(true);
    serverConnection.setDoInput(true);
    serverConnection.connect();

    // Send PKIMessage to server, receive CertRep or Error
    DEROutputStream dRequestData = new DEROutputStream(
    serverConnection.getOutputStream());
    dRequestData.writeObject(myPKIMessage);
    //dRequestData.flush();
    dRequestData.close();
    PKIMessage repPKIMessage = PKIMessage.getInstance(
      new ASN1InputStream(
        serverConnection.getInputStream()).readObject());
  }
  catch (Exception e) {
    errors++;
    e.printStackTrace();
  }
}
if(timing) {stopTime = new Date();}
if(timing) {System.out.println(requests + " requests took " +
```

```
        (stopTime.getTime() - startTime.getTime()) +
        " milliseconds, number of errors: " + errors);}

    } // end try

    catch (Exception e) {
        e.printStackTrace();
    } // end catch

  } // end main

} // end class stresstest
```

stresstest_renew.java is used for performance testing of renewal of certificates using digital signatures for authentication:

```
// SPEKE/CMP on-the-fly SSO implementation
//
// Author: Martin Eian
// This class is used to stress test certificate renewal by the server.
// The client class must be run once first, with at least on renewal,
// to write a PKIMessage to file that can be loaded by the stress test.
package no.ntnu.item.ttm4900;

import java.io.FileInputStream;
import java.io.FileReader;
import java.io.OutputStream;

import java.math.BigInteger;

import java.net.URLConnection;
import java.net.URL;

import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.Provider;
import java.security.SecureRandom;
import java.security.Security;

import java.util.Date;
import java.util.Properties;

import org.bouncycastle.asn1.ASN1InputStream;
import org.bouncycastle.asn1.DERBitString;
import org.bouncycastle.asn1.DERInteger;
import org.bouncycastle.asn1.DEROctetString;
import org.bouncycastle.asn1.DEROutputStream;
import org.bouncycastle.asn1.DERSequence;
import org.bouncycastle.asn1.x509.X509CertificateStructure;
import org.bouncycastle.crypto.digests.SHA1Digest;
```

```
import org.bouncycastle.crypto.engines.RSAEngine;
import org.bouncycastle.crypto.params.DSAParameters;
import org.bouncycastle.crypto.params.DSAPrivateKeyParameters;
import org.bouncycastle.crypto.params.RSAPrivateCrtKeyParameters;
import org.bouncycastle.crypto.signers.DSASigner;
import org.bouncycastle.crypto.signers.PSSSigner;
import org.bouncycastle.openssl.PEMReader;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.jce.provider.JCERSAPrivateKey;
import org.bouncycastle.jce.provider.JCERSAPrivateCrtKey;
import org.bouncycastle.jce.provider.RSAUtil;

import com.novosec.pkix.asn1.cmp.*;
import com.novosec.pkix.asn1.crmf.*;

class stresstest_renew
{
  static final boolean debug = false;
  static final boolean timing = true;
  static boolean rsa = false;
  static final String propertiesFile = "client.properties";
  // SHA-1 of the password 'test123'
  static BigInteger P = new
  BigInteger("65387856594671371314962910427547810457186772780 4");
  static BigInteger TWO = BigInteger.valueOf(2);

  public static void main(String[] args)
  {
    try {
      Properties myProperties = new Properties();
      myProperties.load(new FileInputStream(propertiesFile));
      String serverUrl = myProperties.getProperty("serverUrl");

      Provider bcProvider = new BouncyCastleProvider();
      Security.addProvider(bcProvider);
      SecureRandom secureRandom = new SecureRandom();

      String username = (args.length > 0 ? args[0] : "test");
      String pwd = (args.length > 1 ? args[1] : "test");
      // How many requests to make during stress test
      String requests = (args.length > 1 ? args[2] : "1");
      int numberRequests = Integer.parseInt(requests);
      DERInteger myDERP, myDERQ, myDERG, myDERX;
      DSAPrivateKeyParameters privateKey;
      DERSequence myDERSequence;
      SHA1Digest mySHA1Digest = new SHA1Digest();
      DSASigner myDSASigner = new DSASigner();
      PSSSigner myRSASigner =
```

```
  new PSSSigner(new RSAEngine() , mySHA1Digest , 20);
byte[] digestIn, digestOut;
RSAPrivateCrtKeyParameters privateKeyRSA = null;
JCERSAPrivateCrtKey privateKeyTmpRSA = null;
KeyPair keyPairRSA;

Date startTime, stopTime;
PKIMessage myPKIMessage =
  PKIMessage.getInstance(
    new ASN1InputStream(
      new FileInputStream("certreq.der")).readObject());

PKIHeader myPKIHeader = myPKIMessage.getHeader();
PKIBody myPKIBody = myPKIMessage.getBody();

X509CertificateStructure myCert =
  X509CertificateStructure.getInstance(
    new ASN1InputStream(
      new FileInputStream("client.crt")).readObject());

PEMReader myPEMReader =
  new PEMReader(new FileReader("client_key.pem"));

if(myCert.getSubjectPublicKeyInfo().getAlgorithmId().
          getObjectId().getId().
          equals("1.2.840.113549.1.1.1"))
{
  // We are using RSA, load RSA private key
  rsa = true;
  keyPairRSA = (KeyPair)myPEMReader.readObject();
  privateKeyRSA =
    (RSAPrivateCrtKeyParameters)
      RSAUtil.generatePrivateKeyParameter(
        (JCERSAPrivateKey)keyPairRSA.getPrivate());
  myRSASigner.init(true, privateKeyRSA);
}
else {
  // We are using DSA, load DSA private key
  myDERX = DERInteger.getInstance(new ASN1InputStream(
  new FileInputStream("client.key")).readObject());
  myDERSequence =
    (DERSequence)
      myCert.getSubjectPublicKeyInfo().
             getAlgorithmId().getParameters();
  myDERP = (DERInteger)myDERSequence.getObjectAt(0);
  myDERQ = (DERInteger)myDERSequence.getObjectAt(1);
  myDERG = (DERInteger)myDERSequence.getObjectAt(2);

  privateKey =
```

```
        new DSAPrivateKeyParameters(
          myDERX.getValue(),
          new DSAParameters(myDERP.getValue(),
                            myDERQ.getValue(),
                            myDERG.getValue()));
      myDSASigner.init(true , privateKey);
    }

    BigInteger myNonce;

    DERInteger[] myDERIntegerArray = new DERInteger[2];
    DERBitString myDERSignature;
    BigInteger[] mySignature = new BigInteger[2];
    URL serverURL = new URL(serverUrl);
    URLConnection serverConnection = serverURL.openConnection();
    if(timing) {startTime = new Date();}

    int errors = 0;
    for ( int i = 1 ; i <= numberRequests ; i++ ) {
      try{

        // Make each PKIMessage different, so that
        // the server cannot cache the signature
        // validation data.
        myNonce = new BigInteger(128 , secureRandom);
        myPKIHeader.setSenderNonce(new DEROctetString(
          myNonce.toByteArray()));

        myPKIMessage = new PKIMessage(myPKIHeader, myPKIBody);

        // Add old certificate as extraCert
        myPKIMessage.addExtraCert(myCert);

        // The message has changed, so we need to sign
        // it again
        // Add protection to PKIMessage
        digestIn = myPKIMessage.getProtectedBytes();
if(rsa){
          myRSASigner.update(digestIn , 0 , digestIn.length);
          myDERSignature =
            new DERBitString(myRSASigner.generateSignature());
}
else{
          mySHA1Digest.update(digestIn , 0 , digestIn.length);
          digestOut = new byte[mySHA1Digest.getDigestSize()];
          mySHA1Digest.doFinal(digestOut , 0);

          // Then sign it
          mySignature = myDSASigner.generateSignature(digestOut);
```

```
            myDERIntegerArray[0] = new DERInteger(mySignature[0]);
            myDERIntegerArray[1] = new DERInteger(mySignature[1]);

            myDERSignature =
              new DERBitString(new DERSequence(myDERIntegerArray));
  }

          myPKIMessage.setProtection(myDERSignature);

          serverConnection = serverURL.openConnection();
          serverConnection.setRequestProperty("Content-Type",
          "application/pkixcmp");
          serverConnection.setUseCaches(false);
          serverConnection.setDoOutput(true);
          serverConnection.setDoInput(true);
          serverConnection.connect();

          // Send PKIMessage to server, receive CertRep or Error
          DEROutputStream dRequestData = new DEROutputStream(
          serverConnection.getOutputStream());
          dRequestData.writeObject(myPKIMessage);
          //dRequestData.flush();
          dRequestData.close();
          PKIMessage repPKIMessage = PKIMessage.getInstance(
          new ASN1InputStream(
          serverConnection.getInputStream()).readObject());
        }
        catch (Exception e) {
          errors++;
          e.printStackTrace();
        }
      }
      if(timing) {stopTime = new Date();}
      if(timing) {System.out.println(requests + " requests took " +
      (stopTime.getTime() - startTime.getTime()) +
      " milliseconds, number of errors: " + errors);}

    } // end try

    catch (Exception e) {
      e.printStackTrace();
    } // end catch

  } // end main

} // end class stresstest_renew
```

Appendix N

Key generation class

```
// SPEKE/CMP on-the-fly SSO implementation
//
// This class is used to test key generation
// performance
// Author: Martin Eian

package no.ntnu.item.ttm4900;

import java.io.FileInputStream;

import java.math.BigInteger;

import java.security.Provider;
import java.security.SecureRandom;
import java.security.Security;

import java.util.Date;
import java.util.Properties;

import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
import org.bouncycastle.crypto.generators.DSAKeyPairGenerator;
import org.bouncycastle.crypto.generators.DSAParametersGenerator;
import org.bouncycastle.crypto.generators.RSAKeyPairGenerator;
import org.bouncycastle.crypto.params.DSAKeyGenerationParameters;
import org.bouncycastle.crypto.params.DSAParameters;
import org.bouncycastle.crypto.params.RSAKeyGenerationParameters;

import org.bouncycastle.jce.provider.BouncyCastleProvider;

class keygen
{
  static final boolean debug = true;
  static final boolean timing = true;
  static final String propertiesFile = "client.properties";
```

```
static BigInteger DSAP, DSAQ, DSAG;
static int certainty = 100;

public static void main(String[] args)
{
  try {
    String timesString = (args.length > 0 ? args[0] : "1");
    int times = Integer.parseInt(timesString);
    String strengthString = (args.length > 1 ? args[1] : "768");
    int strength = Integer.parseInt(strengthString);
    Date startTime, stopTime;
    Properties myProperties = new Properties();
    myProperties.load(new FileInputStream(propertiesFile));
    // Pre-generated DSA parameters
    if (strength == 2048) {
      DSAP = new BigInteger(myProperties.getProperty("2048p"));
      DSAQ = new BigInteger(myProperties.getProperty("2048q"));
      DSAG = new BigInteger(myProperties.getProperty("2048g"));
    }
    else if (strength == 1024) {
      DSAP = new BigInteger(myProperties.getProperty("1024p"));
      DSAQ = new BigInteger(myProperties.getProperty("1024q"));
      DSAG = new BigInteger(myProperties.getProperty("1024g"));
    }
    else {
      DSAP = new BigInteger(myProperties.getProperty("768p"));
      DSAQ = new BigInteger(myProperties.getProperty("768q"));
      DSAG = new BigInteger(myProperties.getProperty("768g"));
    }

    Provider bcProvider = new BouncyCastleProvider();
    Security.addProvider(bcProvider);
    SecureRandom secureRandom = new SecureRandom();
    AsymmetricCipherKeyPair keyPair, myRSAKeyPair;
    RSAKeyGenerationParameters rkgp =
      new RSAKeyGenerationParameters(
        new BigInteger("65537"),
        secureRandom,
        strength,
        certainty) ;
      RSAKeyPairGenerator rpg = new RSAKeyPairGenerator();
      rpg.init(rkgp);

      DSAKeyGenerationParameters kgp = new DSAKeyGenerationParameters(
        secureRandom , new DSAParameters(DSAP, DSAQ, DSAG));
      DSAKeyPairGenerator keyGen = new DSAKeyPairGenerator();
      keyGen.init(kgp);
      keyPair = keyGen.generateKeyPair();
```

```
      for ( int i = 1 ; i <= times ; i++ ) {
        if(timing) {startTime = new Date();}
        myRSAKeyPair = rpg.generateKeyPair();
        if(timing) {stopTime = new Date();}
        if(timing) {System.out.println(
          "RSA|" + strength +
          "|" +
          (stopTime.getTime() - startTime.getTime()) +
          "|milliseconds.");}
        if(timing) {startTime = new Date();}
        keyPair = keyGen.generateKeyPair();
        if(timing) {stopTime = new Date();}
        if(timing) {System.out.println(
          "DSA|" + strength +
          "|" +
          (stopTime.getTime() - startTime.getTime()) +
          "|milliseconds.");}
      }

    } // end try

    catch (Exception e) {
        e.printStackTrace();
    } // end catch

  } // end main

} // end class keygen
```

Appendix O

DSA parameter generation class

```
// SPEKE/CMP on-the-fly SSO implementation
//
// This class is used to test DSA parameter
// generation performance
// Author: Martin Eian

package no.ntnu.item.ttm4900;

import java.math.BigInteger;

import java.security.Provider;
import java.security.SecureRandom;
import java.security.Security;

import java.util.Date;

import org.bouncycastle.crypto.generators.DSAParametersGenerator;
import org.bouncycastle.crypto.params.DSAParameters;

import org.bouncycastle.jce.provider.BouncyCastleProvider;

class paramgen
{
  static final boolean debug = true;
  static final boolean timing = true;
  static int certainty = 100;

  public static void main(String[] args)
  {
    try {
      String timesString = (args.length > 0 ? args[0] : "1");
```

```
      int times = Integer.parseInt(timesString);
      String strengthString = (args.length > 1 ? args[1] : "768");
      int strength = Integer.parseInt(strengthString);
      Date startTime, stopTime;

      Provider bcProvider = new BouncyCastleProvider();
      Security.addProvider(bcProvider);
      SecureRandom secureRandom = new SecureRandom();

      DSAParametersGenerator dpg = new DSAParametersGenerator();
      dpg.init(strength , certainty , secureRandom);
      DSAParameters dp;

      for ( int i = 1 ; i <= times ; i++ ) {
        if(timing) {startTime = new Date();}
        dp = dpg.generateParameters();
        if(timing) {stopTime = new Date();}
        if(timing) {System.out.println(
          "DSAparam|" + strength +
          "|" +
          (stopTime.getTime() - startTime.getTime()) +
          "|milliseconds.");}
      }

    } // end try

    catch (Exception e) {
        e.printStackTrace();
    } // end catch

  } // end main

} // end class paramgen
```

Appendix P

Hardware info

OS (uname -a) and CPU information (/proc/cpuinfo) for Dell PE 2650 w/dual Intel Xeon 2.8 GHz. The server had 2 GB of physical RAM.

```
Linux puma 2.6.5-7.151-smp #1 SMP Fri Mar 18 11:31:21 UTC 2005 i686
i686 i386 GNU/Linux

processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Xeon(TM) CPU 2.80GHz
stepping        : 9
cpu MHz         : 2791.424
cache size      : 512 KB
physical id     : 0
siblings        : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe cid
bogomips        : 5505.02

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Xeon(TM) CPU 2.80GHz
```

```
stepping        : 9
cpu MHz         : 2791.424
cache size      : 512 KB
physical id     : 0
siblings        : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe cid
bogomips        : 5570.56

processor       : 2
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Xeon(TM) CPU 2.80GHz
stepping        : 9
cpu MHz         : 2791.424
cache size      : 512 KB
physical id     : 3
siblings        : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe cid
bogomips        : 5570.56

processor       : 3
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Xeon(TM) CPU 2.80GHz
stepping        : 9
cpu MHz         : 2791.424
cache size      : 512 KB
physical id     : 3
siblings        : 2
```

```
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe cid
bogomips        : 5570.56
```