HSR
UNIVERSITY OF APPLIED SCIENCES
RAPPERSWIL

COMPUTER SCIENCE

student research project

Open reference implementation of a SCEP v2 client

Ueli Rutishauser
<urut@easc.ch>

Alain Schäfer
<alani@easc.ch>

Advisor
Dr. Andreas F. Müller

March 1, 2002

Typeset by LaTeX

Contents

List of Figures

Preface

by
Dr. Andreas Müller,
Institute of Applied Mathematics,
University of Applied Science Rapperswil, Switzerland

The Simple Certificate Enrollment Protocol (SCEP), originally devised by Cisco and used in IOS based routers to deploy certificates for IPsec VPNs, has proved to be quite successful in many large scale installations. It even became part of other applications, like the Open Settlement Protocol (OSP) used in IP telephony. Its most compelling feature for large scale installations is the possibility to automatically enroll certificates: each certificate requesting entity is authenticated by a password that is used only once, during the enrollment, but is never transmitted in clear. This obviates out of band request verification, normally required to establish the identity of the certificate requestor. Certificate enrollment becomes a much less tedious procedure.

Unfortunately, Internet Browsers, another large group of potential certificate consumers, are not capable of speaking SCEP. Even worse, SCEP in its present form is not capable of transporting a certificate request as generated by say Netscape Navigator for at least two reasons: the special format (SPKAC) this browser is using and the problem that a SCEP client must have access to the private key of the certificate request.

Other enrollment protocols discussed for large scale enrollment suffer from the same or similar difficulties. So why not take the lightweight SCEP and try to adapt it to a wider range of clients? If the proven suitabilty of SCEP for embedded systems (VPN gateways and IP phones as examples) can be conserved while extending its applicability to browsers, traditionally difficult certificate consumers like set-top boxes become prime candidates for SCEP.

Ueli Rutishauser and Alain Schaefer have taken a detailed look at SCEP and have found two extensions which will remove the limitations alluded to in the previous paragraphs. By writing a SCEP client implementation in Java, capable of running within small JVMs like the J2ME, they have verified their proposal (driving the free SCEP implementation of the OpenSCEP project to also include these new features) and opened the possibility to use SCEP in many more applications than currently possible. Besides their readily useable HTTP SCEP proxy to enroll browsers, the command line client could be used on FreeS/WAN based VPN-Gateways to establish the necessary trust relationsships. The fact that their implementation is quite conservative as far as operating system resources are concerned proves that a Java based solution can be an option for embedded systems for an application domain that has traditionally been reserved to “bit-twiddling” languages like C.

Chapter 1

Introduction

The only suitable cryptography scheme for secure communication between an potentially large number of users is public key cryptography. The prime advantage of Public Key Cryptography is that it doesn't require any previous out-of-band communication between two entities that want to communicate securely. If Bob wants to send a secure message to Alice, Bob simply takes the public key of Alice and encrypts the message with it. The problem here is not the encryption algorithm itself but rather the public key. Bob must be able to verify that the public key really belongs to Alice. That is where Certificates and Public Key Infrastructure become part of the game.

Certificates are a collection of a distinguished name (identification of the end entity, for example 'Alice'), a public key and a signature. The signature is the key element with which a user of a public key is able to verify that a public key belongs to a particular distinguished name. The signature of a certificate is being generated by a third party that is trusted by both involved parties. The third party, the so called certificate authority (CA), confirms that a public key really belongs to a particular distinguished name by signing the certificate. Afterwards, whoever wants to verify whether a certificate is valid or not can do so by comparing the signature of a certificate with the one that it has computed itself.

Getting a certificate for a newly created private/public key pair is usually done manually. The process of getting a signed certificate is called certificate enrollment. An entity that whishes to obtain a certificate that is signed by a CA for a newly created private/public key pair has to carry out the following steps:

1. Generate new private/public key pair
2. Generate a certificate request for this key pair
3. Send the certificate request to a trusted third party (CA)
4. Obtain proof of authenticity
5. Get back the signed certificate

Manual certificate enrollment may be suitable for individuals that want to obtain a certificate but it isn't an option for large-scale deployment of public key authentication. There must be a way of obtaining certificates that doesn't involve manual interaction. The Simple Certificate Enrollment Protocol (SCEP) is a proposed protocol for automated certificate enrollment. SCEP offers a standard interface to communicate with a Certificate Authority. It includes several properties that allow it to do an automatic enrollment without any manual interaction. All available implementations of SCEP are based on draft version 5 ([1]) of the SCEP proposal which has been published as an offical Internet Draft. The typical usage scenarios of SCEP is a large network environment with many network devices that require certificates. Cisco uses SCEP as the protocol to automatically get certificates for routers and other network devices.

We will outline why the current version of SCEP, as specified in the draft version 5, isn't usable in a browser based environment where SCEP should be used to obtain certificates from a trusted third party. We will furthermore propose a set of extensions to the current draft specification to enhance SCEP so it can cope with these new requirements. We provide a reference implementation of a SCEP client to provide a basis for evaluating the proposed enhancements.

Chapter 2

Motivation

SCEP ([1]) is a relatively new certificate management protocoll proposed by Cisco as an Internet Standard. It supports certificate life cycle operations such as certificate enrollment, revocation and CRL distribution.

X.509 Certificates are used for providing proof of authentication through the trust in a CA (Certificate Authority). One of the scenarios where X.509 certificates are used is a browser based enviornment where SSL connections or encrypted e-mail require certificates. But there is one step that needs to be done before a browser can use a certificate - the certificate enrollment. Certificate enrollment is usually done by hand. This involves three steps:

1. Generation of a private key and a certificate request
2. Transmission of the certificate request to the CA; Proof of authentity to the CA
3. Installation of the signed certificate

Manually submitting the certificate request of a browser to a CA is no problem if there are just a few clients but this is no option for a large number of users or for users with no knowledge of public key cryptography. Large-scale deployment of certificates (for example in a company where every employee should get a certificate or in a network environment where every network component requires its own certificate) requires automatic certificate enrollment. That is what SCEP (Simple Certificate Enrollment Protocoll) was invented for. SCEP is, at present, most commonly used for automatic certificate enrollment for routers.

No such mechanism as SCEP is currently available for the large-scale deployment of certificates to web browsers. It is the aim of this project to develop a proposal for the extension of the SCEP protocol so that it is capable of dealing with these new requirements. We furthermore aim to develop a reference implementation for the full functionality of the extended SCEP protocol.

It has been proven that SCEP is a good solution for automatic certificate enrollment for IPSec based VPN networks. Our reference implementation should be capable of bringing this benefit to an embedded devices with FreeS/WAN ([20]).

2.1 Requirements

Requirements:

- Public/Private Key generation and enrollment of client certificates via SCEP.
- Automatic import into the certificate databases of Netscape and Internet Explorer

- Easy portable to Windows, MacOS, Linux, Sun OS
- Usable on embedded system
- Automatic enrollment – usable without user interaction or minimal user or operator interaction

Chapter 3

Solutions

This chapter shows possible solutions for the mentioned problems. Advantages and disadvantages of all solutions are described. The chapter ends with a section that shows which of the approaches were chosen to be realized.

3.1 Certificate Storage

After a certificate has sucessfully been obtained, it must be stored on the users harddisk. The new certificate should then easily be accessible for the common webbrowsers.

Both major Browser (Netscape & MSIE) have their own certificate database. Client & Server certificates are stored in this database. Certificates used for the daily web browsing or email (S/MIME) are stored in these certificate databases.

The problem faced is, how to install a new certificate into these certificate databases. This is even more difficult because not only the certificate but also the according private key should be installed into this database.

3.1.1 Browser supported certificate installation

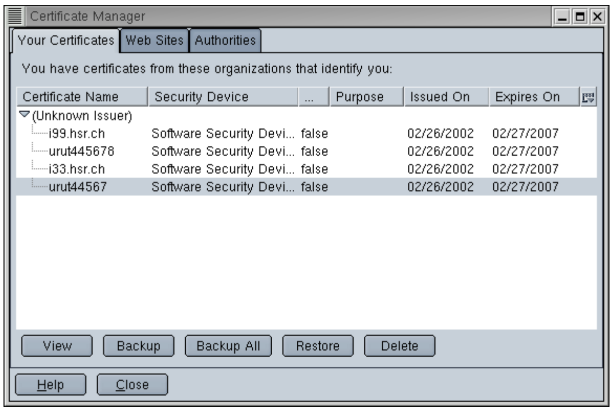
Both major Browser (Netscape & MSIE) have a mechanism to generate client certificate requests. Both can send these certificates embedded in a HTML form via HTTP to a web server. The server is then responsible for signing this certificate and sending it back to the browser. The signed certificate can then be installed with the HTTP response.

The advantage of this approach is the ease of use for endusers. Either they access the remote server via a TCP/IP connection or they access a locally installed server program.

The drawback is that we face again an unsecure communication over http, when using a remote SCEP Proxy. Even when using SSL this is only an option in a secure network.

Netscape Client Certificate Management

Netscape introduced a proprietary HTML-Tag (keygen) with Version 3.0 of its Browser. This keygen tag causes the browser to generate a key pair, and return the public key as a form value. It thus has to be put in a HTML-Form. The KEYGEN tag causes the browser to display a drop down box for the choice of security grade. For an example see figure 3.1.



You have certificates from these organizations that identify you:

Certificate Name	Security Device	...	Purpose	Issued On	Expires On
(Unknown Issuer)					
i99.hsr.ch	Software Security Devi...	false		02/26/2002	02/27/2007
urut445678	Software Security Devi...	false		02/26/2002	02/27/2007
i33.hsr.ch	Software Security Devi...	false		02/26/2002	02/27/2007
urut44567	Software Security Devi...	false		02/26/2002	02/27/2007

Figure 3.1: The Netscape 6.x certificate store

The choices available depend on the version & export type of the browser. The KEYGEN tag is not only supported by Netscape, also Opera and KDE's Konqueror have support for it.

When the KEYGEN-Tag is used, the browser will send the a Certifcate Request in the SPKAC format. The SPKAC Formate is specified as follwed : (siehe developer.netscape.com) noch importieren

Please note that unlike PKCS#10 ([12]) which is used in SCEP, SPKAC ([15]) does not have any distinguished subject name. It does also not contain any optional set of attributes.

After the certificate has been generated on the server, it can be sent back to browser with MIME-Type application/x. A Netscape Browser will recognize this mime type and install the certificate into it certificate database.

Internet Explorer Client Certificate Management

Internet Explorer doesn't come with an embedded engine for client certificate installation like Netscape's Navigator. It uses the CryptoAPI (CAPI) defined by Microsoft. This has the advantage that other applications can access the certificate (see 3.2)

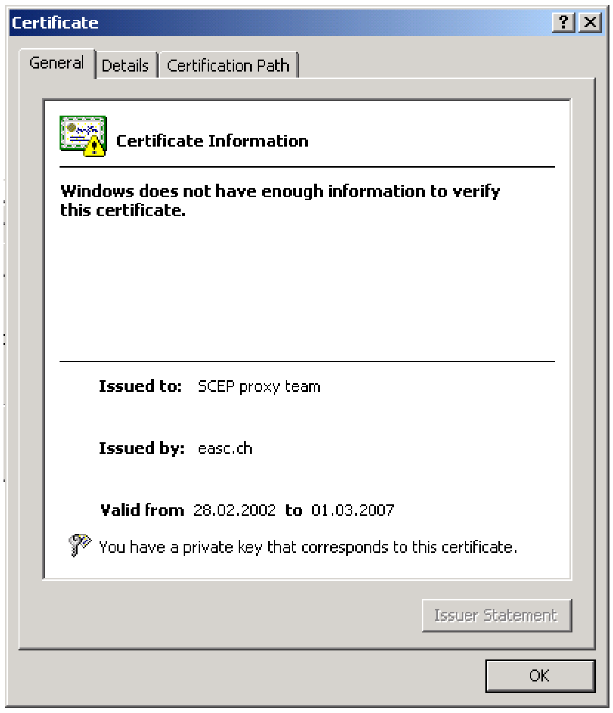
Since CryptoAPI is a system API it can not be accessed from HTML code directly. It has to be accessed with JavaScript or VBScript and an ActiveX Control from within a web page. The result from a call to GenerateKeyPair is a PKCS#10 request which is put in a hidden field in a HTML Formular.

3.2 Client as a Java application

Another approach would be to write a java application which does the certificate enrollment. The same application would also write the certificate into the browsers certificate database.

The difficulty of this approach would be the access to the different certificate databases. Examples and Description about these databases are widley available, see [2], [3], [4], [5]. For a documentation about the Microsoft Crypto API see [6].

Although it would be possible to access both types of certificate databases, it would be very difficult if not impossible to do this in a portable way. It is doubtfull if this could be done with 100% Java code. This makes this solution very hard to install for the end user.



Certificate Information

Windows does not have enough information to verify this certificate.

Issued to: SCEP proxy team

Issued by: easc.ch

Valid from 28.02.2002 to 01.03.2007

You have a private key that corresponds to this certificate.

Figure 3.2: The MS Windows certificate store

3.3 Client as a Java applet

This is approach would be very similiar to one mentioned above. The main advantage of writing the client as a java applet is the ease of installation.

But the implementation as an applet also adds some restrictions for the implementation. A native API as an example can not be accessed from an applet.

3.4 Native Client

Writing a native client would make it easier to access the certificate databases, since native API's for the certificate databases are already shipped with the browsers.

A native client would also rise the burden of making the client portable to different platforms.

3.5 Decision

We have decided to choose the road of browser supported certificate storage. We will write a SCEP Proxy in Java which will act as an HTTP Server to the web browsers. We will initially support Mozilla and Microsoft Internet Explorer. Support for Opera, Netscape 4 and Konqueror support will be added if time permits to do so.

Chapter 4

Architecture

4.1 General Architecture

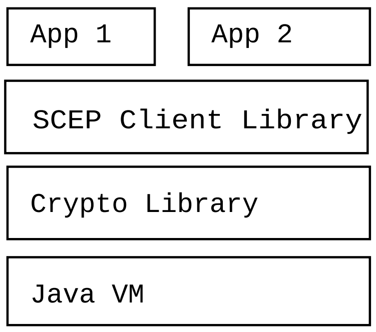


Figure 4.1: Architecture of a SCEP client

The core of every application that wants to act as a SCEP client is the SCEP client library (Figure 4.1). The SCEP client library provides all necessary functions that a typical SCEP client needs. This includes the whole certificate enrollment process (sending certificate request, poll for answer) as well as the CRL handling. The SCEP client library is based on a crypto library that provides the basic cryptography and certificate handling functions like X509, PKCS, ASN.1 (DER and BER encoding) and S/MIME.

4.2 Types of SCEP clients

We consider two types of SCEP clients - browser based clients and embedded system clients. Both types of clients use the same SCEP client library, but they are executed within different runtime environments (J2SE or J2ME). Both type of clients are explained in detail in the next sections.

4.2.1 Browser based clients

Additional to the standard SCEP Client, the standalone client consists of a HTTP server that acts as a proxy. The user can then simply point his browser to the specified URL to access the proxy. This scenario is shown in Figure A.1.

The HTTP server (acting as proxy) serves as a local HTTP server that handles the certificate requests from a browser. There are basically three operations that the HTTP server supports: get new certificate (request),

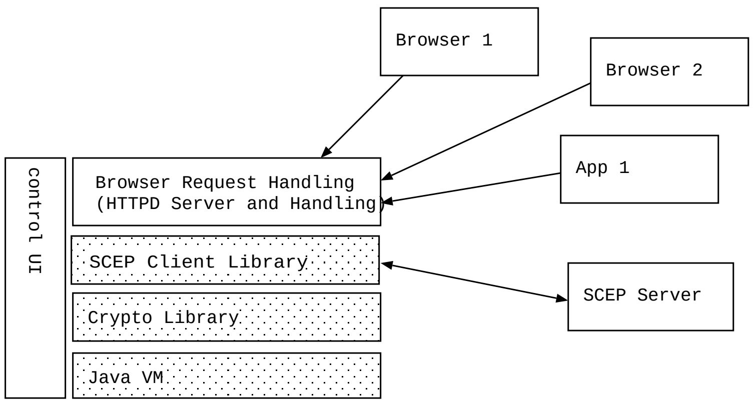


Figure 4.2: Architecture of a system with Browsers as clients

certificate polling (check whether the requested certificate is already available) and get CA certificate. Depending on the type of browser the user is using (Netscape, Mozilla, IE) a different version of the pages is generated. The Netscape/Mozilla/Opera/Konqueror version uses the KEYGEN HTML tag whereas the IE version uses an activeX control that is included by default in MS IE.

The whole package, consisting of the SCEP client library and the embedded HTTP server, is deployed together as a single application. A user who wants to make a certificate request simply starts the proxy server and opens a connection to it with his browser for which he wants to get a certificate. The root certificate of the certificate authority for which the SCEP client is used will typically be included in this package as well. This should establish the necessary trust the browser must have in the SCEP proxy.

4.2.2 Application scenarios for browser based environments

Get Root Certificate

Browsers that support certificates also need to have the ability to get the root certificate of a certificate authority. Most browsers simply download the DER encoded root certificate of the CA via HTTP.

Certificate Request

Browser that support Certificates generate their own private key automatically when the request process is started. How this process is initiated depends on the type of browser, but generally two differing schemes are in use: the KEYGEN HTML tag and client side scripting. After the browser has generated its own private key it generates a certificate request that is sent to the SCEP proxy via HTTP. Then the SCEP client deals with this certificate request and forwards it to a SCEP server. The sequence diagram in Figure 4.3 shows how such a request is transmitted from a Browser to the certificate authority.

CRL Request

CRL (Certificate Revocation Lists) are currently not supported by browsers. Certificate revocation needs manual interaction.

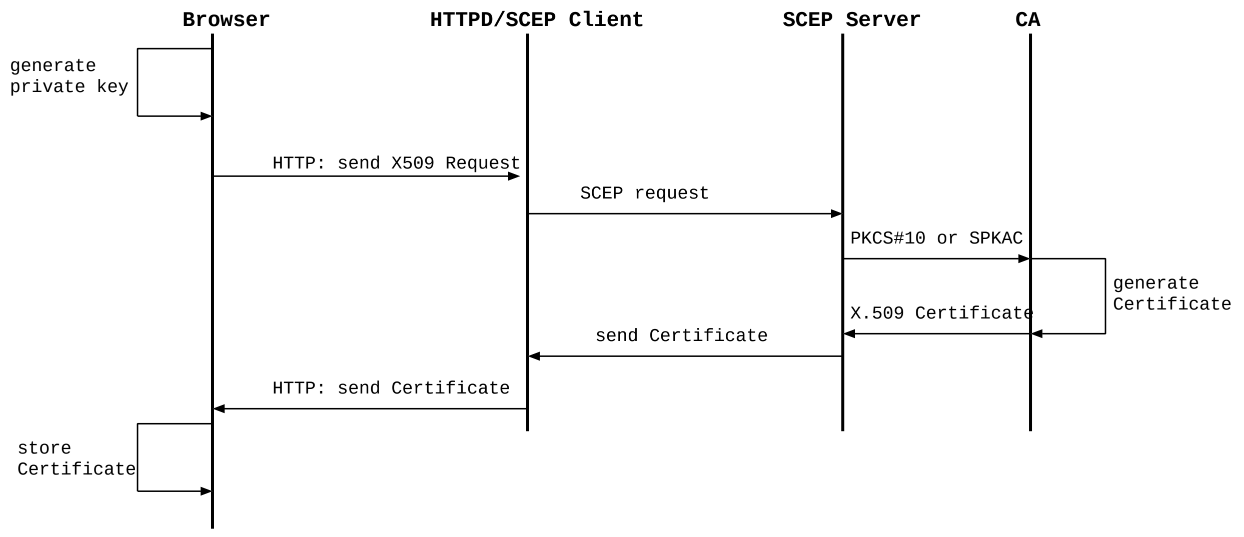


Figure 4.3: Sequence Diagram of a certificate request

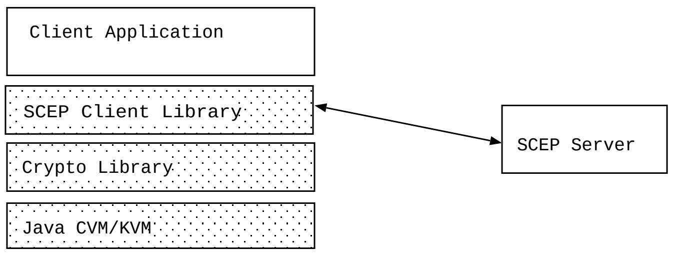


Figure 4.4: Architecture of a embedded system SCEP client

4.2.3 Embedded clients

Requirements of embedded applications

The runtime environment of an application (Figure 4.4) on an embedded system is very different from that of a standard desktop and/or server application. Special emphasis must be given to facts like limited system resources, limited availability of read/write storage or different deployment processes.

Facts that have to be considered when an application is required to be runnable on a embedded system:

- Limited system resources like special processor architectures, slow processors, limited memory
- Limited availability of read/write storage possibilities
- Limited Operating System support, operating system might only support a subset of usual operations
- For Java applications: Limited availability of Java standard libraries, depending on the type of virtual machine that is used
- Execution from read-only memory

SCEP in a embedded system environment

The SCEP client library is also able to run in a J2ME environment. This makes it possible for a developer of a embedded system application to use the SCEP client library for the certificate management.

A possible implementation of SCEP client could be a simple command line client. The developer of this command line client could simply use the SCEP client library to communicate with a SCEP server.

An embedded system client would typically use the CDC (connected device configuration) version of the J2ME.

4.3 Third party products

We use the following third party products.

4.3.1 BouncyCastle Crypto API

The Bouncy Castle Crypto API (See [17]) is a full fledged open source crypto API that contains a lightweight cryptography API, a JCE provider as well as libraries for handling ANS.1, X509 Certificates and partially PKCS.

It is possible to use the BouncyCastle Crypto API with a J2ME runtime environment because the implementation of the SCEP client doesn't require any functionality from the JCE part.

4.3.2 ACME Serve

ACME Serve (See [16]) is a minimal HTTP server that can easily be embedded into other applications. It is written 100% in Java and is compatible with the Java servlet API. It won the 1997 editors choice award of Distinction of BYTE magazine.

4.3.3 JDK1.3 / J2ME

The runtime environment is a standard JDK1.3 (or a similar J2ME environment).

4.3.4 OpenSCEP 0.4.2

OpenSCEP ([18]) is an open source implementation of a SCEP version. Version 0.4.2 implements the full sets of extensions that are proposed in this paper.

Chapter 5

Design and Implementation

5.1 Overview

This implementation is divided into the following parts:

- Core (Protocol stack)
- HTTP Proxy (Gateway HTTP - SCEP)
- Command Line Client

The following parts of this chapter explains every three parts of the system in detail.

5.2 Core

The Core part of the SCEP implementation (Figure 5.1) is where the SCEP protocol itself is implemented. It consists of all classes responsible for decoding and encoding of different SCEP messages.

5.2.1 Overall Design

Every available SCEP command is implemented in its own class inherited from ScepCommand. The class ScepClient acts as facade (Facde pattern, see [7]) for the whole protocol implementation. The execution of the different SCEP commands is done by the execute method of ScepClient. This utilizes the command Pattern ([7]). ScepClient furthermore acts as a data container. It contains all data that is required to execute the different commands. This data are for example keys (private+public), Certificate Authority Certificates or X509-Subjects. ScepClient has the capability to dynamically load the different keys, requests and certificates from arbitrary URLS, in particularly from http and file URL's. All files that are required by ScepClient are assumed to be Base64 encoded and will thus automaticaly be decoded during loading.

5.2.2 Utils - Logging

There are 3 log levels in this implementation - info, warning and error. Clients can choose which messages of which log level should be printed.

The Log class (Figure 5.2) contains static methods callable by the whole system to output log information. There is one method for every log level (Info, Warning,Error).

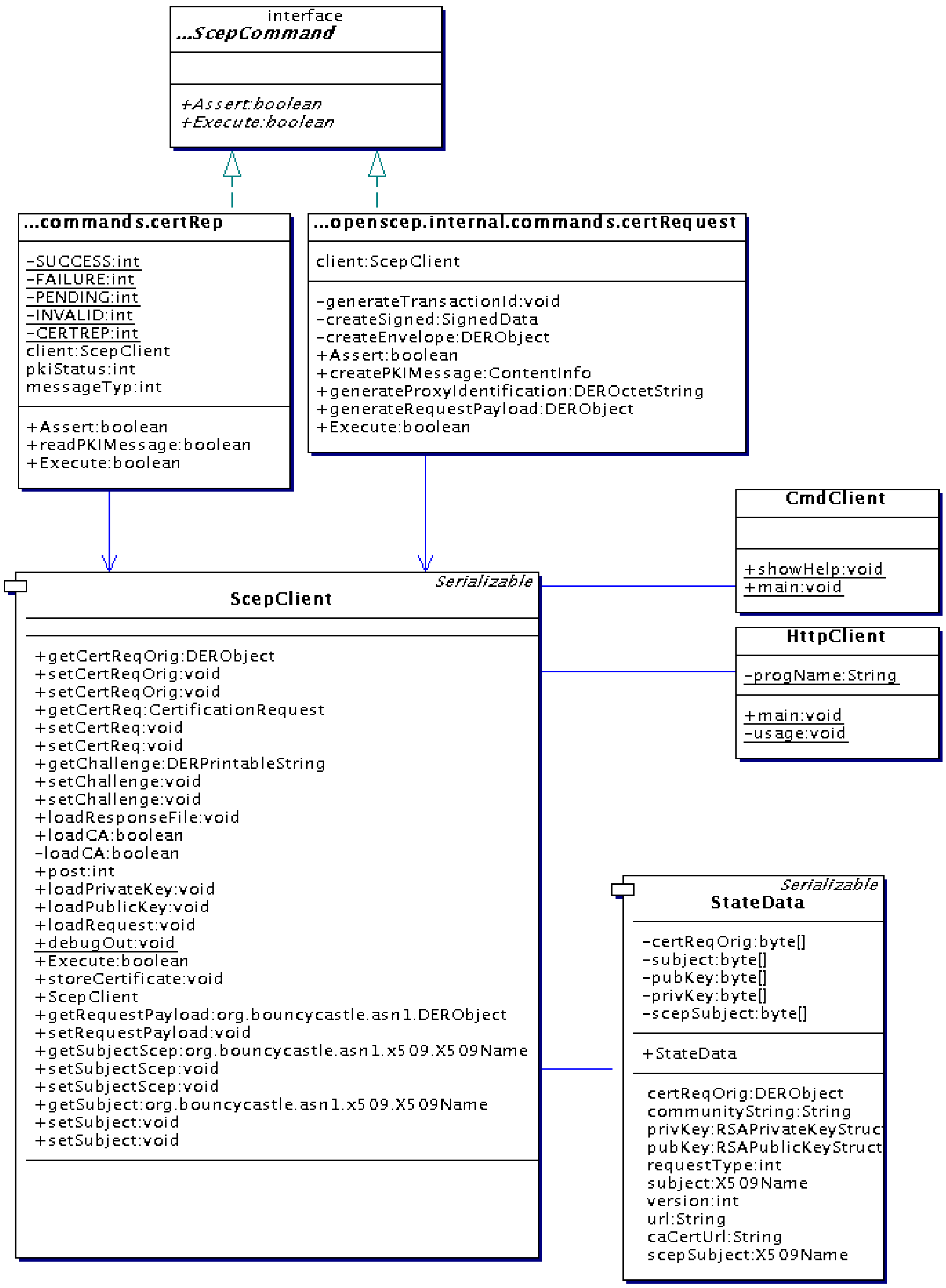


Figure 5.1: Class diagram of the core parts

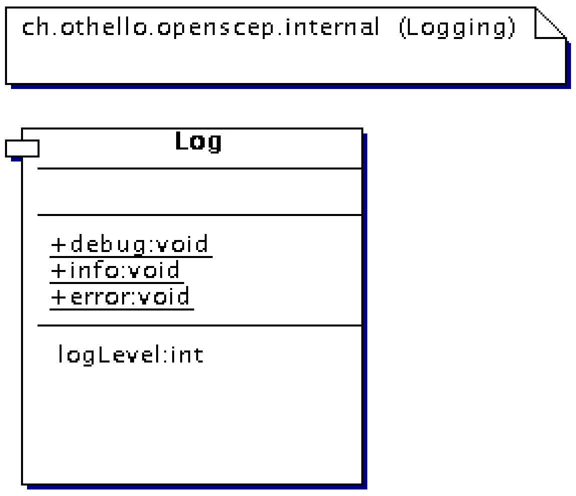


Figure 5.2: Log class

5.2.3 Utils - MD5 Hash and RSA Algorithm

The SCEP protocol requires DES and RSA encryption with special PKCS padding mechanisms. It furthermore requires RSA encrypted MD5 hashes to sign messages. The classes in this part of the system (Figure 5.3) provide this functionality.

SCEPObjectIdentifiers contains static constants for all OID (Object Identifiers) used in the SCEP protocol implementation. Only non-standard attributes like algorithm identifiers and specific SCEP attributes are included in this class. Standard Object Identifiers are directly defined in the crypto library and not specified in here.

CryptUtils provides static methods for dealing with PKCS compliant padding in RSA and DES. The class RSAMD5Digest generates RSA encrypted MD5 signatures and MD5 digests. Utils is a general class that offers methods for dealing with DER encoded data.

5.2.4 Utils - HTTP Client

SCEP uses HTTP/1.0 as transport protocol but Java acts as an HTTP/1.1 client by default. This causes several problems with different encodings. HTTP/1.1 clients have to be prepared to deal with HTTP answers that may contain chunked encoding. The class ScepHTTPGet (Figure 5.4) contains the functionality to read answers from a SCEP server that are chunked encoded. This seems to be a weakness of the native Java HTTP client support (URL Class) – If the HTTP client acts as a 1.1 client, it should also isolate the programmer from the intricacies of 1.1 chunked encoding or it should at least allow the programmer to fall back to 1.0, so that (s)he does't need to deal with it.

5.2.5 Utils - Base64 encoding, conversion of DER Objects

This part (Figure 5.5) of the Utils constists of classes for the handling of Base64 encoded files, strings and streams. It furthermore contains methods to convert DER Objects to bytearrays and vice-versa.

The class PEMReader consists of different methods to decode Base64 encoded data and PEMWriter contains different methods to encode data as Base64.

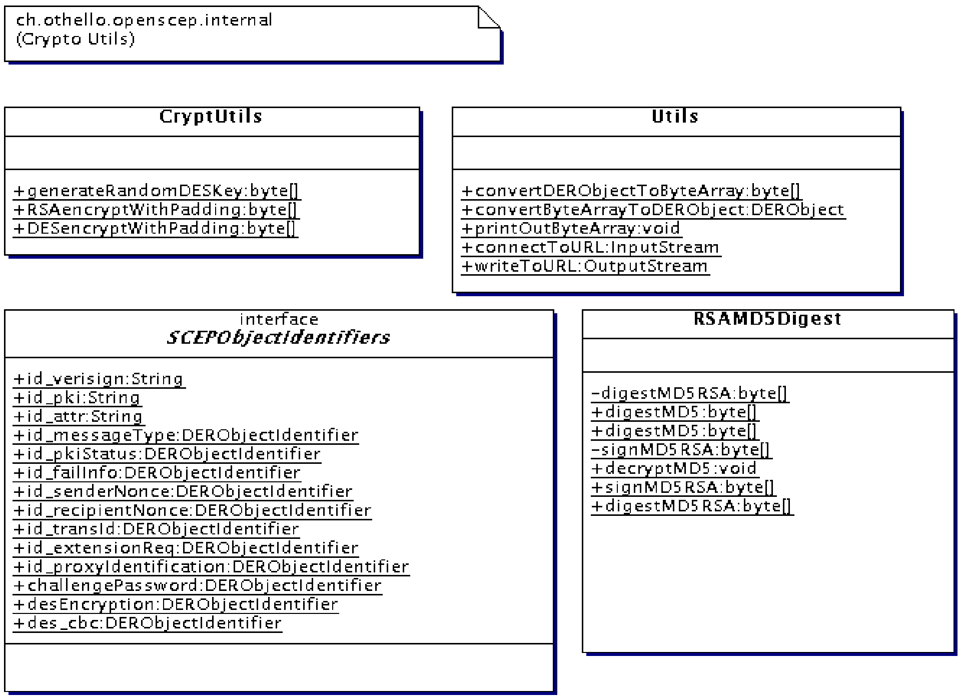

ch.othello.openscep.internal
(Crypto Utils)
CryptUtils
+generateRandomDESKey:byte[]
+RSAencryptWithPadding:byte[]
+DESencryptWithPadding:byte[]
Utils
+convertDERObjectToByteArray:byte[]
+convertByteArrayToDERObject:DERObject
+printOutByteArray:void
+connectToURL:InputStream
+writeToURL:OutputStream
interface
SCEPObjectIdentifiers
+id_verisign:String
+id_pki:String
+id_attr:String
+id_messageType:DERObjectIdentifier
+id_pkiStatus:DERObjectIdentifier
+id_failInfo:DERObjectIdentifier
+id_senderNonce:DERObjectIdentifier
+id_recipientNonce:DERObjectIdentifier
+id_transId:DERObjectIdentifier
+id_extensionReq:DERObjectIdentifier
+id_proxyIdentification:DERObjectIdentifier
+challengePassword:DERObjectIdentifier
+desEncryption:DERObjectIdentifier
+des_cbc:DERObjectIdentifier
RSAMD5Digest
-digestMD5RSA:byte[]
+digestMD5:byte[]
+digestMD5:byte[]
-signMD5RSA:byte[]
+decryptMD5:void
+signMD5RSA:byte[]
+digestMD5RSA:byte[]

Figure 5.3: Cryptography support classes

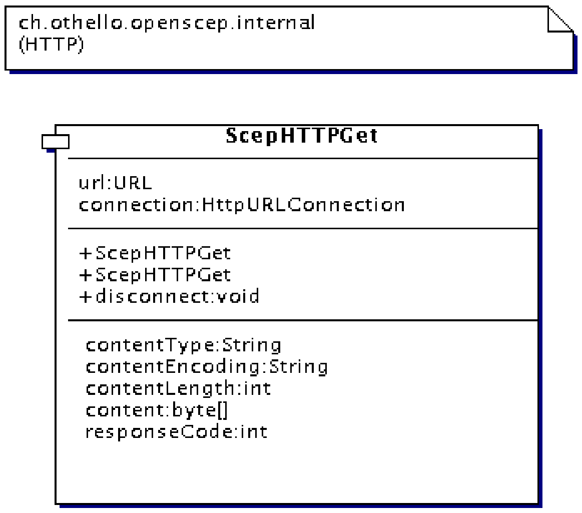


Figure 5.4: HTTP client class

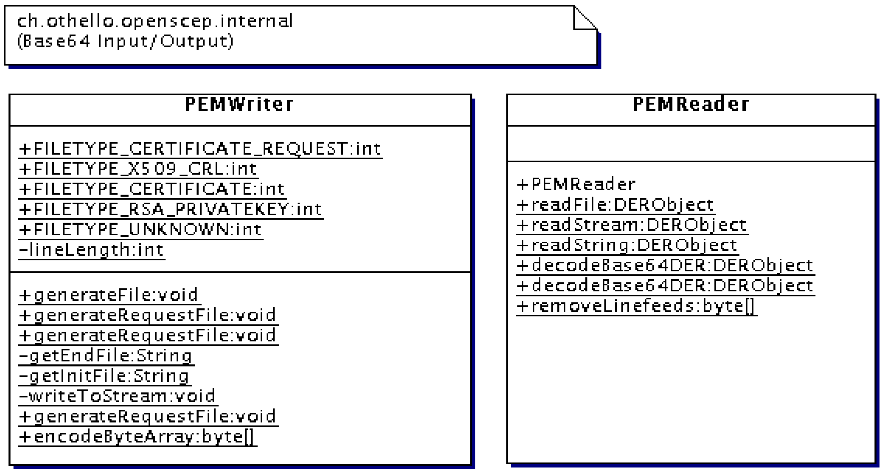


Figure 5.5: Base64 support classes

5.2.6 Protocol - Commands

The relevant SCEP commands for this client implementation are PKCSReq and CertRep. The other commands that are specified in the SCEP protocol specification are not relevant for this implementation.

There is one specific class (Figure 5.6) for handling every type of SCEP command.

certRequest is the class responsible for generating a SCEP request whereas certRep is responsible for decoding an answer from a SCEP server. These two classes contain the core of the SCEP protocol. They include the computation of transactionID's, MD5 digests and signatures as well as RSA and DES encryption/decryption. It furthermore generates the different ASN.1 structures out of this information which is afterwards being sent to a SCEP server.

5.2.7 Protocol - PKCS#7

Figure 5.7 shows the implementation classes of PKCS#7. Only the parts of PKCS#7 that are required by SCEP are implemented. EnvelopedData, SignedData and SignerInfo will become part of BouncyCastle later.

5.3 Proxy

The SCEP proxy for web browsers is designed to make it easy to add support for browsers which are not supported as of today. It strives for easy configuration and deployment on client computers and maximal reuse of code.

The proxys main class is HTTPClient. It creates an instance of ACME-Serve and registers the servlets. All interactions are then handled by the servlets located in the package ch.othello.openscep.servlet (see figure 5.8)

5.3.1 Design

We will look at the different servlets in the sequence of a user interacting with them. As an example we take a user of a Netscape compatible browser enrolling for a certificate at an authority with automatic enrollment (see figure 5.9)

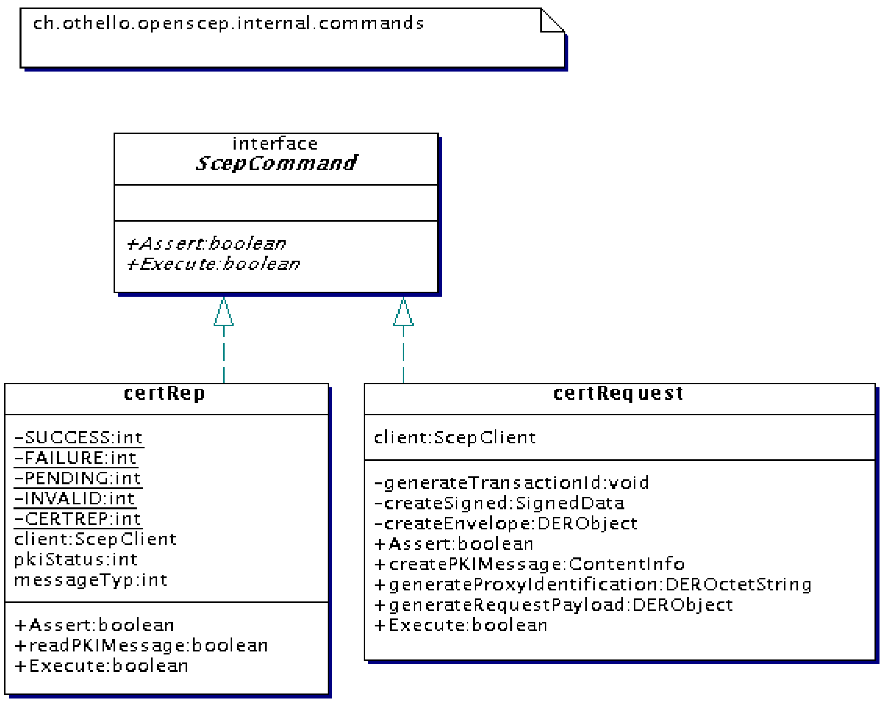


Figure 5.6: Classes for the SCEP commands

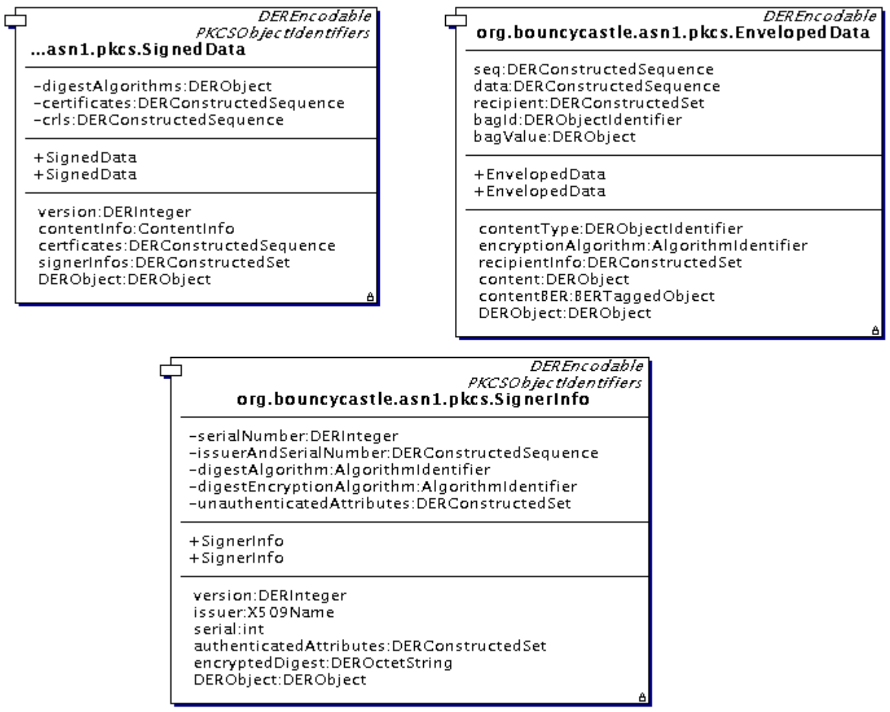

DEREncodable
PKCSObjectIdentifiers
...asn1.pkcs.SignedData
-digestAlgorithms:DERObject
-certificates:DERConstructedSequence
-crls:DERConstructedSequence
+SignedData
+SignedData
version:DERInteger
contentInfo:ContentInfo
certficates:DERConstructedSequence
signerInfos:DERConstructedSet
DERObject:DERObject
DEREncodable
org.bouncycastle.asn1.pkcs.EnvelopedData
seq:DERConstructedSequence
data:DERConstructedSequence
recipient:DERConstructedSet
bagId:DERObjectIdentifier
bagValue:DERObject
+EnvelopedData
+EnvelopedData
contentType:DERObjectIdentifier
encryptionAlgorithm:AlgorithmIdentifier
recipientInfo:DERConstructedSet
content:DERObject
contentBER:BERTaggedObject
DERObject:DERObject
DEREncodable
PKCSObjectIdentifiers
org.bouncycastle.asn1.pkcs.SignerInfo
-serialNumber:DERInteger
-issuerAndSerialNumber:DERConstructedSequence
-digestAlgorithm:AlgorithmIdentifier
-digestEncryptionAlgorithm:AlgorithmIdentifier
-unauthenticatedAttributes:DERConstructedSet
+SignerInfo
+SignerInfo
version:DERInteger
issuer:X509Name
serial:int
authenticatedAttributes:DERConstructedSet
encryptedDigest:DEROctetString
DERObject:DERObject

Figure 5.7: PKCS#7

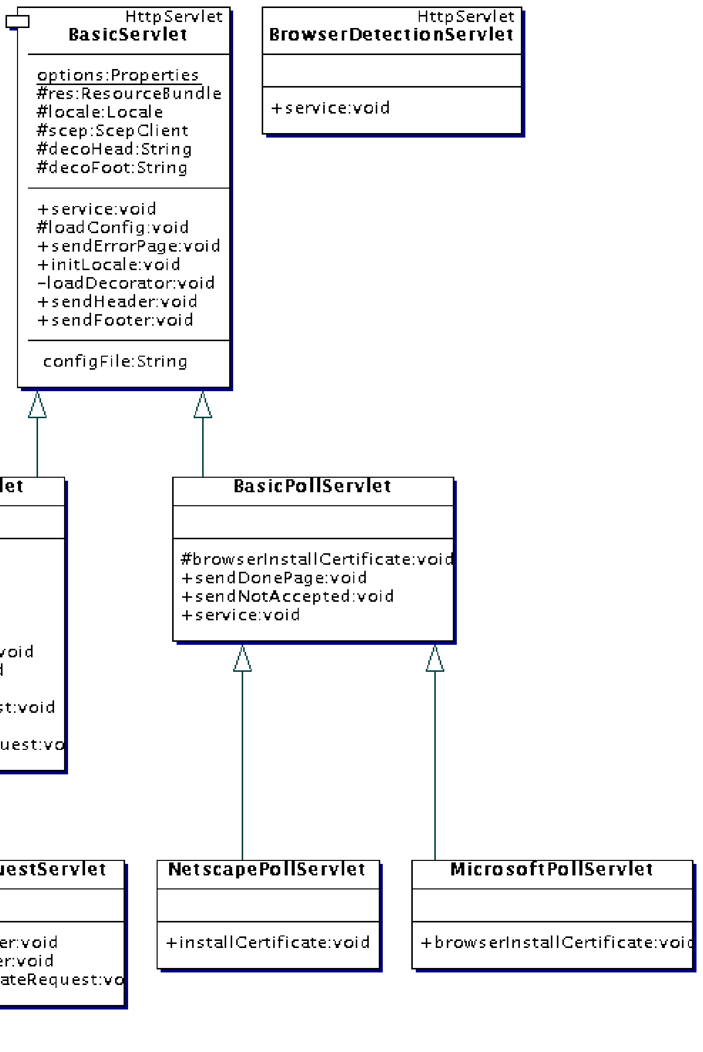


Figure 5.8: HTTP proxy classes

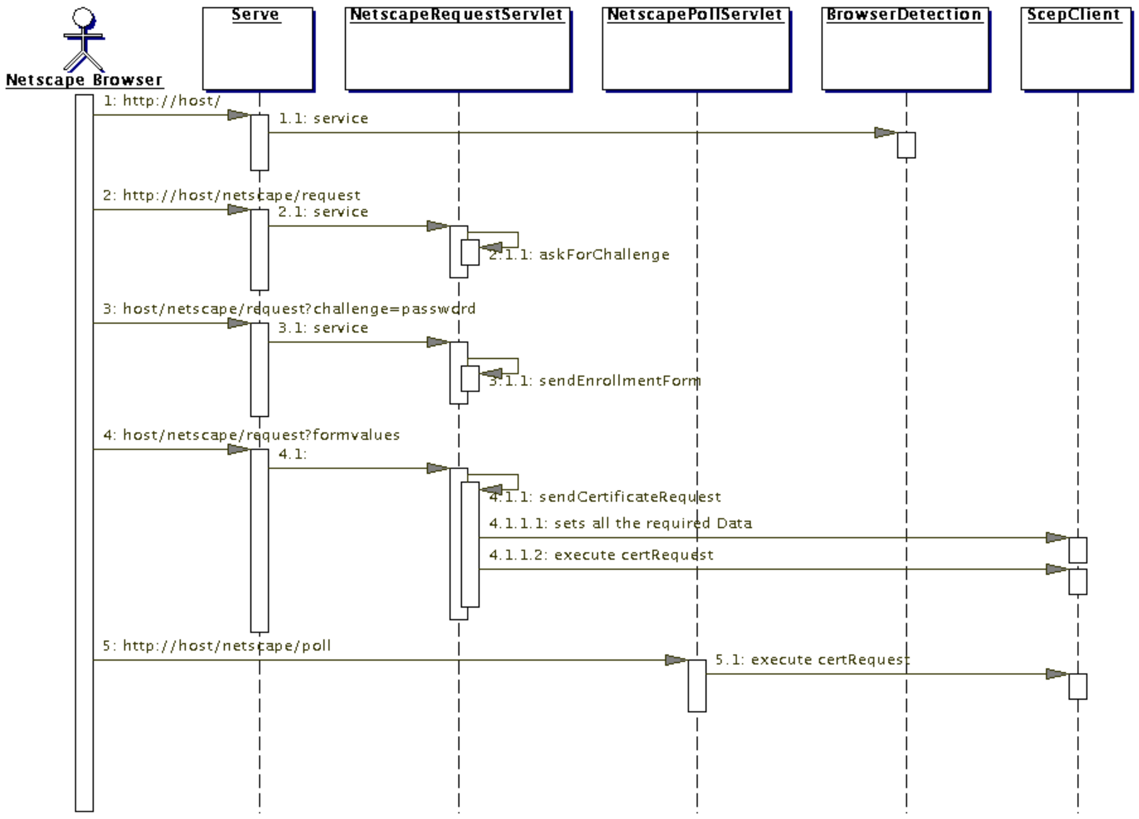

Netscape Browser
Serve
NetscapeRequestServlet
NetscapePollServlet
BrowserDetection
ScepClient
1: http://host/
1.1: service
2: http://host/netscape/request
2.1: service
2.1.1: askForChallenge
3: host/netscape/request?challenge=password
3.1: service
3.1.1: sendEnrollmentForm
4: host/netscape/request?formvalues
4.1:
4.1.1: sendCertificateRequest
4.1.1.1: sets all the required Data
4.1.1.2: execute certRequest
5: http://host/netscape/poll
5.1: execute certRequest

Figure 5.9: Sequence of a User interacting with the Proxy

The BrowserDectectionServlet intercepts the users first request and tries to detect the type of browser by the http header field named user-agent. It forwards the users to the specific servlet for its browser type. In our example this is /netscape/request (1).

The NetscapeRequestServlet answers requests for this path. Depending on the configuration, it first asks the user for a challenge password (2) or directly presents him the certificate generation form (3). The certificate generation form is then submitted again to this servlet, which issues the enrollment request to the SCEP server (4). If the enrollment is successful it sends the certificate back to the browser which installs it in its database.

If the request is pending on the SCEP server, the state of the ScepClient is saved in one or more cookies which are sent to the browser. The users browser is then forwarded to the poll servlet responsible for this type of browser (5), here NetscapePollServlet. The browser then periodically polls for the certificate until the poll servlet sends the issued certificate.

To maximise code resuablility and lower the possiblity of errors, common functionality of the servlet for the two different browsers has been extracted to Baseclasses. BasicServlet implements functions to detect the desired language (see section 5.3.3 on page 22), loading the config file, decorating the output and displaying an ErrorPage to the user.

BasicRequestServlet implements common certificate request functionality. The service method checks if a challenge password has been entered and asks for one if this is required. It proceeds to the certificate request form afterwards.

5.3.2 Configuration

The SCEP Proxy can easily be adapted to specific requirements . A property file (scepProxyConfig), which is located in the package ch.othello.openscep.servlet, is used as configuration file. The config file is put into the java archive of the proxy to make it even easier to install. Should you feel the need to make local changes to your config file and don't want to put it into the java archive you can specify the location of the config file with the command line switch -c configfile.

Configuration parameters

```
url=http://openscep.othello.ch/pkiclient.exe
#specifies the url for the SCEP cgi program

caCertUrl=http://openscep.othello.ch/cacert.pem
#specifies the url for the CA Certificate

privateKey=file:///usr/locale/lib/scep/examples/rsaprivatekey.pem
publicKey=file:///usr/locale/lib/scep/examples/rsapublickey.pem
#these two parameters tell SCEP Proxy where to look for it's own public/private key pair

proxySubject=CN=SCEP Proxy, O=OpenSCEP Ltd., OU=Studenten, L=Rapperswil, ST=SG, C=Switzerland
#this is the distinguished name used by the proxy

communityString=SCEPCommunity
#this is the community string which identifies this proxy to the SCEP server as beeing
#member of a community

forceChallenge=true
#if this parameters value is true, the user will be asked for a challenge String, default
#is false
```

```
challenge=Please enter your challenge Password
#this is the default value set for the challenge password

pollTimeout=10
#this sets the timeout until polling for the certificate will be tried again

#The Following parameters are used for defaultValues for the enrollees distinguished name.
#The Default value is enforced if the value of the coresponding \_Default parameter is true

commonName=
emailAddress=
organization=OpenSCEP Ltd.
organization\_Default=true
orgUnit=Studenten
locality=Rapperswil
locality\_Default=true
state=SG
state\_Default=true
country=Switzerland
country\_Default=true
```

5.3.3 Translation

Since this piece of code was developed in Switzerland, it should be easily translatable to different languages. This is achievd by using Java Resource Bundles. Until now the SCEP Proxy has been written in english and translated to german. The respective files are HTTPResources.properties and HTTPResources.properties_de.

Language detection

The SCEP Proxy tries to detect the user's language by parsing the HTTP-Header accept-language. It accepts the first choice which it finds. This is due to Java's PropertyResourceBundle automatically falling back to the the default language (if the desired language is not supported). This language negotiation can be overridden by adding the parameter lang=XX to the url where XX is the ISO Locale Code (de or de_CH).

5.3.4 Appearance

In its default configuration the SCEP Proxy has a simplistic look. But the proxy implements a derivation of the decorator pattern (see [7]). In the configuration one can specify a decorator file which can add a customized appearance to the SCEP Proxy. This file can be included in the jar file or somewhere on the local filesystem.

The decorator file can contain two different variables.

- {SCEPTITLE} - is replaced by the title of the actual page
- {SCEPCONTENT} - is replaced by the actual content generated by the SCEP Proxy servlets

5.4 Command Line Client

The SCEP command line client is capable of doing version 1 and version 2 requests with either PKCS#10 or SPKAC as embedded certificate request. The certificate request itself has to be supplied as a compulsory command line argument as it isn't automatically generated by the command line client. The command line client loads all external files as an URL which gives the flexibility to load the required files, either per HTTP or from a ordinary file

5.4.1 Implementation

5.4.2 Options

The command line client offers the following options (see the examples in the following section for clarification).

```
-verbose
-logLevel number          1, 2 or 3 (1=minimum, 3=maximum, default=3)
-requestType PKCS|SPKAC   only valid for version 2 (default is PKCS)
-version 1|2              SCEP Version (default 1)
-poll                     poll the server for a certificate
-subject string           subject string, eg. CN=xxxx  . only if type=SPKAC
-subjectProxy string      subject of the proxy, eg. CN=xxxx  . only if type=SPKAC
-community string         Community String (V2->proxy)
-caCertURL url            URL to get the CA cert eg http://..../ or file:///tmp/...
-privatekey filename      URL to get the private key
-pubkey filename          URL to get the public key (is automaticly extracted from the
                          pkcs#10 request (if available)
-certReq filename         URL to get the certification request
-serverURL theURL         URL of the server to which requests are sent eg http://..../
-request filename         File where the request is being stored before it is sent out
-response filename        Manually supply a response from a SCEP server
-certificate filename     Location (URL) where the CA signed end entity certificate is written
                          to (in case of a SUCCESS response)
```

5.4.3 Examples

The following examples assume that the following files are present:

- /tmp/rsaprivatekey.pem Base64 encoded RSA Private Key of the Proxy
- /tmp/rsapublickey.pem Base64 encoded RSA Public Key of the Proxy
- /tmp/rsaprivatekeyScep.pem Base64 encoded RSA Private Key of the Proxy
- /tmp/rsapublickeyScep.pem Base64 encoded RSA Public Key of the Proxy
- /tmp/pkcs10.pem Base64 encoded PKCS#10 Request
- /tmp/spkac.pem Base64 encoded SPKAC Request

It is furthermore assumed that the SCEP Server is reachable at the URL http://localhost/cgi-bin/openscep/pkiclient.exe and that the Certificate Authority Certificate is available at the URL http://localhost/cacert.pem

The answer of a request (in case of a SUCCESS response) is written to /tmp/answer in the following examples.

PKCS#10 with SCEP Version 1

For a version 1 request the supplied private key matches the public key of PKCS#10. Due to this the public key mustn't be specified in case of a version 1 request as it is automatically extraced from the PKCS#10 request.

To make a PKCS#10 request with SCEP Version 1:

```
java -jar scepclient-cmd.jar
-logLevel 3 -caCertURL http://localhost/cacert.pem
-privatekey file:///tmp/rsaprivatekey.pem -certReq file:///tmp/pkcs10.pem
-serverURL http://localhost/cgi-bin/openscep/pkiclient.exe certRequest
```

PKCS#10 and SPKAC with Version 2

Following are two examples of a call to the SCEP command line client that illustrate the major difference between SCEP version 1 and version 2. Version 2, in contrast to version 1, works with two different public/private keypairs. The first keypair is used by the certificate requesting end entity, the second one is used by the SCEP proxy to encrypt the communication between the proxy and the SCEP server. The SCEP proxy just knows the public key of the certificate requesting end entity but not the private key of it(in contrast to version 1).

To make a PKCS#10 request with SCEP Version 2 (SCEP Client is acting as proxy):

```
java -jar scepclient-cmd.jar
-logLevel 2
-requestType PKCS
-version 2
-community SCEPCommunity
-pubkey file:///tmp/rsapublickeyScep.pem
-caCertURL http://localhost/cacert.pem
-privatekey file:///tmp/rsaprivatekeyScep.pem
-certReq file:///tmp/pkcs10.pem
-certificate file:///tmp/answer.pem
-serverURL http://localhost/cgi-bin/openscep/pkiclient.exe certRequest
```

To make a SPKAC request with SCEP version 2 (SCEP client action as proxy):

```
java -jar dist/scepclient-cmd.jar
-logLevel 3
-requestType SPKAC
-subject CN=easc.ch,O=CH
-subjectProxy CN=hinn.ch
-version 2
-community SCEPCommunity
-pubkey file:///tmp/rsapublickeyScep.pem
-caCertURL http://localhost/cacert.pem
-privatekey file:///tmp/rsaprivatekeyScep.pem
-certReq file:///tmp/spkac.pem
-serverURL http://localhost/cgi-bin/openscep/pkiclient.exe certRequest
```

5.5 Packaging

The SCEP client package is divided into the packages

- ch.othello.openscep
  Contains all clients that are using the SCEP client library. It furthermore contains the ScepClient class which acts as a facade to the protocol implementationl
- ch.othello.openscep.internal
  Contains utility classes for reading/writing Base64 encoded files, for MD5 signing, RSA encryption and logging.
- ch.othello.openscep.internal.commands
  Contains the implementation of the SCEP protocol
- ch.othello.servlet
  Contains all servlets that are required by the http proxy implementation of the SCEP client
- ch.easc.tests Contains utility classes for dumping out X509 and ASN.1 structures (for debugging)

5.6 Deployment

5.6.1 Build

All necessary jar files to build this package are included. You only have to start build.sh resp. build.bat. The build systems uses ant (included in the subdirectory lib).

The ant targets are :

- all
  This is the standard target. It builds the two later mentioned jar files
- http-jar
  dist/scepclient-http.jar which contains the http proxy client. To start the proxy type : java -jar dist/scepclient-http.jar
- cmd-jar
  This is a commandline client with minimal memory requirements. It builds the jar file dist/scepclient-cmd.jar. To start the cmd client type : java -jar dist/scepclient-cmd.jar

5.6.2 Configuration

The command line package (scepclient-cmd.jar) doesn't contain any deployment specific files. All necessary options can be supplied as command line arguments. Thus, no customization of scepclient-cmd.jar is necessary.

scepclient-http.jar contains various deployment-specific files:

ch/othello/openscep/servlet/scepProxyConfig Proxy Configuration File

HTTPResources.properties Language specific configuration file; This file contains the default language

HTTPResources_de.properties The same but for german (Language code de). Add more of this files if you want to have other languages

Adjust this files if necessary.

Appendix A

Proposal for the extension of the SCEP Protocol

A.1 Motivation

The SCEP protocol was originally designed to be a simple certificate enrollment protocol. Due to todays increasingly complex network environments, automatic certificate enrollment is one of the most critical issues in deploying big and distributed network environments that could possible involve thousands of different certificates that need to be issued and managed. SCEP gives a simple approach to communicate with a CA. It is, however, limited, as it can only be used by clients that directly implement the SCEP protocol. SCEP Servers can (and do) work as a front-end of a real CA server so that the real CA server doesn't have to implement the SCEP protocol. This is, however, not possible at the client side. Every client that wants to request a certificate has to fulfill two criteria: Firstly, it has to implement the full SCEP protocol and secondly it has to use PKCS#10 as certificate request format. This two preconditions severely limit the applicability of SCEP.

As an example lets look at the situation where the user of an HTML-Browser likes to generate a Certificate and let it be signed by the CA via the SCEP Protocoll. Current Browsers do not implement the SCEP Protocoll. But they are able to generate a Certificate Request and send it to an HTTP Server. This Certificate Request is in PKCS#10 for Microsofts Internet Explorer but in SPKAC for Netscape Navigator and a plethora of other browsers.

The first extension of SCEP is to allow a SCEP client to act as a proxy. This would allow it to have one central SCEP client that acts as a proxy for all the other entities in a network. All the requests that would be received by this SCEP client (which is acting as proxy) would then be forwarded to the respective SCEP server. A similiar problem has arisen with the BOOTP Protocoll. BOOTP introduced the concept of a BOOTP relay agent (RFC 951) to deal with this.

The second extension concerns the limitation to PKCS#10. Restricting the request format to PKCS#10 unnecessarly limits the usage of SCEP, particularly in a browser-based environment. In browser based environments there are two certificate request formats that are most commonly used – PKCS#10 and SPKAC. SPKAC is the format that is used by all browsers of the netscape/mozilla family as well as various other browsers (kde, opera).

The following proposal is divided into two parts: The first part deals with the details of the proxy functionality, the second one deals with the extension of SCEP to allow it to use different certificate request formats. Both sections of the proposal frequently refer to the third section where the complete ASN.1 specification of the proposed protocol extensions is given.

A.2 Extension 1: Proxy

The SCEP protocol draft deals with an end entities key pair. In case of a SCEP client that is acting as proxy it is, however, not always clear which end entity is meant by that – is it the end entity that is actually requesting the certificate or is it the end entity that is dealing with the SCEP protocol (SCEP client)? As long as the end entity that is dealing with the SCEP request is the same as the one that actually wants to request a certificate is the same that doesn't matter. But except for the originally intended use of SCEP, routers, this is usually not the case.

Take the example of a application that wants to request a certificate. This application would typically use a SCEP client library that uses its own private/public key. To make it clear which key has to be used we distinguish between the public/private key of the SCEP client and the public/private key of the certificate requesting end entity. Depending on the application, the public/private key of the SCEP client and the certificate requesting end entity can be the same, but this isn't a requirement anymore.

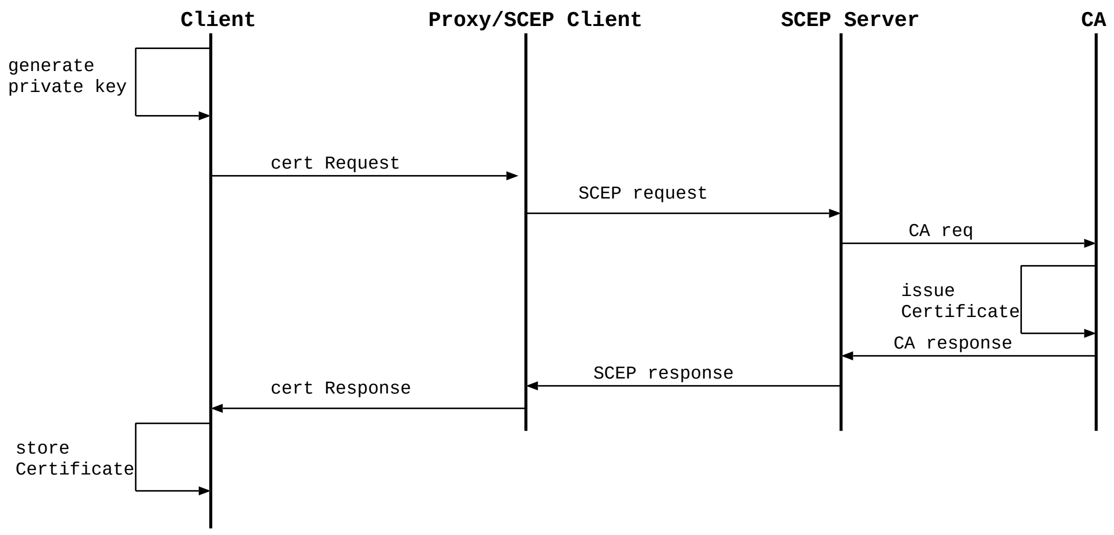


Figure A.1: Sequence Diagram that shows how a certificate request is being forwarded by a SCEP proxy

The proxy functionality introduces a new cryptographic entity, in which trust must be established. While the request itself can be authenticated using attributes in the request (the challengePasswort in the case of a PKCS#10 request, the challenge in a SPKAC request), there is no longer a connection between the SCEP client and the certificate requestor as in the original SCEP protocol. Only proxies that have previously been authorized should be acceptable, so a shared secret (community string) between proxy and server seems appropriate. To establish the connection between the SCEP client and the community, the community string must be part of the signed attributes of the SCEP request. But as the signed attributes are transmitted in clear, the community string must first be hidden in some way.

We should also make sure that the attribute derived from the community string can only be used once: every new client request should lead to a different attribute value. We propose to construct a new proxyIdentification attribute as the MD5 hash of the concatenation of request payload and community string. This makes it impossible for an attacker to snoop a proxyIdentification attribute from the network and use it for a different request without actually knowing the community string. Only a proxy knowing the community string can generate a proxyIdentifcation attribute, and each request leads to a different value.

Requested changes in the SCEP Standard :

All occurrences of "end entity's private key" or "end entity's public key" have to be clarified. They have to be changed either to "certificate requesting end entity's private/public key" or "SCEP client's public/private key".

The attribute proxyIdentification should be added to the list of authenticated Attributes in the SignedData

data structure.

- proxyIdentification Octet String

proxyIdentification contains a MD5 hash of the combination of the Payload and the communityString (see ASN.1 specification for details).

A.3 Extension 2: generic certificate request format

The current version of SCEP implicitly assumes that PKCS#10 is used as certificate request format, but this fact is never explicitly stated. To clearly distinguish between this two format an additonal parameter requestType is part of requestPayload.

The following values for requestType are proposed :

- PKCS#10 (0) – PKCS#10
- SPKAC (1) – SPKAC

Each certificate request format includes other attributes but SCEP only needs the subjectName attribute. As this attribute isn't part of all available certificate request formats there needs to be a possibility to add additional attributes to a request. Because of this, we propose to add a additional Structure called requestPayload that contains the original request and an additional set of attributes where arbitrary PKCS#9 attributes can be added.

```
RequestPayload ::= SEQUENCE {
  requestType INTEGER
 originalRequest BIT STRING -- CertificationRequest or SignedPublicKeyAndChallenge
      attributes ::= SET {    -- DER tagged (unauthenticated),as specified in PKCS#9
         subject   --only required for SPKAC
         -- PKCS#9 attributes, for example SubjectName in the case of SPKAC,
         -- none in the case of PKCS#10
      }
}
```

The calculation of the transactionID needs to be clarified when there are other request formats then PKCS#10 involved. Currently, the transactionID is calculated as the MD5 hash of the public key of the certificate requesting end entity. As other formats like SPKAC don't necessarly allow it to (easily) access the public key of the certificate requesting end entity this is not a practical way to calculate the transactionID. We propose to calculate the transactionID as following: transactionID is the MD5 hash of the combination of the requestPayload and the SCEP proxy public key. This allows it to clearly identify requests that are coming from the same end entity and the same proxy.

A.4 ASN.1 specification of the proposal

```
-- PKCSReq information portion
pkcsCertReq CertificationRequest ::= {   -- PKCS#10
    version 0
    subject  "the certificate requesting end entity's subject name"
    subjectPublicKeyInfo {
        algorithm {pkcs-1 1}  -- rsa encryption
        subjectPublicKey "DER encoding of the certificate requesting end entity's public key"
    }
    attributes {
        challengePassword {{pkcs-9 7} "password string" }
        extensions
    }
    signatureAlgorithm {pkcs-1 4} -- MD5WithRSAEncryption
    signature "bit string which is created by signing inner content
              of the defined pkcsCertReq using certificate requesting end entity's private
              key, corresponding to the public key included in
              subjectPublicKeyInfo."
}

--SPKAC, as specified by netscape
PublicKeyAndChallenge ::= SEQUENCE {
  spki SubjectPublicKeyInfo,
  challenge IA5STRING
}
SignedPublicKeyAndChallenge ::= SEQUENCE {
  publicKeyAndChallenge PublicKeyAndChallenge,
  signatureAlgorithm AlgorithmIdentifier,
  signature BIT STRING
}

RequestPayload ::= SEQUENCE {
      requesttype INTEGER
      originalRequest BIT STRING -- original PKCS#10 (CertificationRequest)
                                 -- or SPKAC request (SignedPublicKeyAndChallenge)
      attributes ::= SET {    -- DER tagged (unauthenticated),as specified in PKCS#9
 subject    -- only required for SPKAC
         -- PKCS#9 attributes, for example SubjectName in the case of SPKAC,
         -- none in the case of PKCS#10
      }
}

-- Enveloped information portion
pkcsCertReqEnvelope EnvelopeData ::= {   -- PKCS#7
    version 0
    recipientInfo {
        version 0
        issuerAndSerialNumber {
            issuer  "the CA issuer name"
            serialNumber  "the CA certificate serial number"
        }
        keyEncryptionAlgorithm  {pkcs-1 1}  -- rsa encryption
        encryptedKey "content-encryption key
                      encrypted by CA public key"
    }
    encryptedContentInfo {
        contentType {pkcs-7 1}  -- data content
        contentEncryptionAlgorithm  "object identifier
                                     for DES encryption"
        encryptedContent  "encrypted requestPayload using
                   the content encryption key"
    }
}
```

```
-- Signed PKCSReq
pkcsCertReqSigned SignedData ::= { -- PKCS#7
    version 1
    digestAlgorithm {iso(1) member-body(2) US(840) rsadsi(113549)
                     digestAlgorithm(2) 5}
    contentInfo {
        contentType {pkcs-7 1} -- data content identifier
        content  pkcsCertReqEnvelope
    }
    certificate {   -- the SCEP client's self-signed certificate
        version 3
        serialNumber  "the transaction id associated with enrollment"
        signature {pkcs-1 4}  -- md5WithRSAEncryption
      issuer " the SCEP client's subject name"
        validity {
            notBefore "a UTC time"
            notAfter  "a UTC time"
        }
        subject  "the SCEP client's subject name"
        subjectPublicKeyInfo {
            algorithm {pkcs-1 1}
            subjectPublicKey "DER encoding of SCEP client's public key"
        }
        signatureAlgorithm {pkcs-1 4}
        signature "the signature generated by using the SCEP client's
                   private key corresponding to the public key in
                   this certificate."
    }
    signerInfo  {
        version 1
        issuerAndSerialNumber {
            issuer "the SCEP client's subject name"
            serialNumber "the transaction id associated
                          with the enrollment"
        }
        digestAlgorithm {iso(0) member-body(2) US(840) rsadsi(113549)
                         digestAlgorithm(2) 5}
        authenticateAttributes {
            contentType  {{pkcs-9 3} {pkcs-7 1}}
            messageDigest {{pkcs-9 4} "an octet string"}
            transaction-id {{id-attributes transId(7)} "printable
                                                        string"}
                           -- this transaction id will be used
                           -- together with the subject name (of the certificate request) as
                           -- the identifier of the certificate requesting end entity's key
                           -- pair during enrollment
            messageType {{id-attributes messageType(2)} "PKCSReq"} --DERPrintableString

        proxyIdentification
{{id-attributes proxyIdentification(id-proxyIdentification)} "hash"} -- see definition

            senderNonce {{id-attributes senderNonce(5)}
                         "a random number encoded as a string"}
        }
        digestEncryptionAlgorithm {pkcs-1 1} -- rsa encryption
        encryptedDigest "encrypted digest of the authenticated
                         attributes using SCEP client's private key"
    }
}
pkcsReq PKIMessage ::= {
    contentType {pkcs-7 2}
    content pkcsCertRepSigned
}

id-proxyIdentification  OBJECT_IDENTIFIER ::= {1 3 6 1 4 1 4263 5 5}    --othello namespace
```

The following attribute is encoded as an authenticated attribute.

- proxyIdentification Octet String

proxyIdentification is the MD5 hash of requestPayload (whole structure) combined with the communityString. This makes sure that the proxy who sent this request knows the communityString. To prove that the proxy knows the communityString the communityString itself doesn't have to be transmitted over the network - combining the requestPayload and the communityString to calculate a MD5 hash of it is sufficient. Please not that the communityString is a raw string that isn't encoded as any form of DER String before the MD5 hash is being calculated.

The attribute requestType in requestPayload can have the following values :

- PKCS#10 (0) – PKCS#10
- SPKAC (1) – SPKAC

The attribute messageTyp of a SCEP request originally had the value "19" (DERPrintableString). To identify a SCEP request of a newer version (e.g. with a requestPayload structure and/or different keys because a proxy is sending the request) as early as possible, new values for messageType are used for this extension. The are defined as following:

- 19 normal SCEP request as specified in the offical proposal
- 17 SCEP request that contains a requestPayload structure that is directly being sent by an end entity
- 18 SCEP request that contains a requestPayload that is being sent from a proxy

RequestPayload contains the original certificate request, which is either PKCS#10 or SPKAC (at present, can be extended). Depending on the type of request, additional attributes can be added to the payload. This is, for example, used to add the subjectName attribute, which is not part of a SPKAC request.

A.5 SPKAC example

Following is a ASN.1 dump of a SPKAC request as it is generated by a browser (Netscape).

```
  0:d=0  hl=4 l= 586 cons: SEQUENCE
  4:d=1  hl=4 l= 306 cons: SEQUENCE
  8:d=2  hl=4 l= 290 cons: SEQUENCE
 12:d=3  hl=2 l=  13 cons: SEQUENCE
 14:d=4  hl=2 l=   9 prim: OBJECT            :rsaEncryption
 25:d=4  hl=2 l=   0 prim: NULL
 27:d=3  hl=4 l= 271 prim: BIT STRING
302:d=2  hl=2 l=  10 prim: IA5STRING         :1234567890
314:d=1  hl=2 l=  13 cons: SEQUENCE
316:d=2  hl=2 l=   9 prim: OBJECT            :md5WithRSAEncryption
327:d=2  hl=2 l=   0 prim: NULL
329:d=1  hl=4 l= 257 prim: BIT STRING
```

A.6 PKCS#10 example

Following is an example of a PKCS#10 request as it is generated by a browser (Internet Explorer in this case).

```
  0:d=0  hl=4 l= 783 cons: SEQUENCE
  4:d=1  hl=4 l= 632 cons:  SEQUENCE
  8:d=2  hl=2 l=   1 prim:   INTEGER           :00
 11:d=2  hl=3 l= 140 cons:   SEQUENCE
 14:d=3  hl=2 l=  20 cons:    SET
 16:d=4  hl=2 l=  18 cons:     SEQUENCE
 18:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
 23:d=5  hl=2 l=  11 prim:      PRINTABLESTRING   :Switzerland
 36:d=3  hl=2 l=  11 cons:    SET
 38:d=4  hl=2 l=   9 cons:     SEQUENCE
 40:d=5  hl=2 l=   3 prim:      OBJECT            :stateOrProvinceName
 45:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :SG
 49:d=3  hl=2 l=  19 cons:    SET
 51:d=4  hl=2 l=  17 cons:     SEQUENCE
 53:d=5  hl=2 l=   3 prim:      OBJECT            :localityName
 58:d=5  hl=2 l=  10 prim:      PRINTABLESTRING   :Rapperswil
 70:d=3  hl=2 l=  22 cons:    SET
 72:d=4  hl=2 l=  20 cons:     SEQUENCE
 74:d=5  hl=2 l=   3 prim:      OBJECT            :organizationName
 79:d=5  hl=2 l=  13 prim:      PRINTABLESTRING   :OpenScep Ltd.
 94:d=3  hl=2 l=  18 cons:    SET
 96:d=4  hl=2 l=  16 cons:     SEQUENCE
 98:d=5  hl=2 l=   3 prim:      OBJECT            :organizationalUnitName
103:d=5  hl=2 l=   9 prim:      PRINTABLESTRING   :Studenten
114:d=3  hl=2 l=  13 cons:    SET
116:d=4  hl=2 l=  11 cons:     SEQUENCE
118:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
123:d=5  hl=2 l=   4 prim:      PRINTABLESTRING   :sdfg
129:d=3  hl=2 l=  23 cons:    SET
131:d=4  hl=2 l=  21 cons:     SEQUENCE
133:d=5  hl=2 l=   9 prim:      OBJECT            :emailAddress
144:d=5  hl=2 l=   8 prim:      IA5STRING         :asdfasdf
154:d=2  hl=3 l= 159 cons:   SEQUENCE
157:d=3  hl=2 l=  13 cons:    SEQUENCE
159:d=4  hl=2 l=   9 prim:     OBJECT            :rsaEncryption
170:d=4  hl=2 l=   0 prim:     NULL
172:d=3  hl=3 l= 141 prim:    BIT STRING
    0001 - <SPACES/NULS>
316:d=2  hl=4 l= 320 cons:   cont [ 0 ]
320:d=3  hl=2 l=  26 cons:    SEQUENCE
322:d=4  hl=2 l=  10 prim:     OBJECT            :1.3.6.1.4.1.311.13.2.3
334:d=4  hl=2 l=  12 cons:     SET
```

```
336:d=5  hl=2 l=  10 prim:      IA5STRING         :5.0.2195.2
348:d=3  hl=2 l=  32 cons:    SEQUENCE
350:d=4  hl=2 l=  10 prim:     OBJECT            :Microsoft Extension Request
362:d=4  hl=2 l=  18 cons:     SET
364:d=5  hl=2 l=  16 cons:      SEQUENCE
366:d=6  hl=2 l=  14 cons:       SEQUENCE
368:d=7  hl=2 l=   3 prim:        OBJECT            :X509v3 Key Usage
373:d=7  hl=2 l=   1 prim:        BOOLEAN           :255
376:d=7  hl=2 l=   4 prim:        OCTET STRING
    0000 - 03                                              .
382:d=3  hl=3 l= 255 cons:    SEQUENCE
385:d=4  hl=2 l=  10 prim:     OBJECT            :1.3.6.1.4.1.311.13.2.2
397:d=4  hl=3 l= 240 cons:     SET
400:d=5  hl=3 l= 237 cons:      SEQUENCE
403:d=6  hl=2 l=   1 prim:       INTEGER           :01
406:d=6  hl=2 l=  92 prim:       BMPSTRING
500:d=6  hl=3 l= 137 prim:       BIT STRING
    0001 - <SPACES/NULS>
640:d=1  hl=2 l=  13 cons:  SEQUENCE
642:d=2  hl=2 l=   9 prim:   OBJECT            :md5WithRSAEncryption
653:d=2  hl=2 l=   0 prim:   NULL
655:d=1  hl=3 l= 129 prim:  BIT STRING
    0001 - <SPACES/NULS>
```

Glossary

CA	Certificate Authority
CRL	Certificate Revocation List
DES	Data encryption standard
DN	Distinguished Name
JCE	Java Cryptography Extension
KEYGEN	Netscape proprietary HTML Tag that is also supported by other browsers like Opera
MD5	Hashing algorithm
PKCS	Public Key Cryptography Standard
RA	Registration Authority
RSA	Public key encryption algorithm - named after the three inventors Ron Rivest, Adi Shamir and Leonard Adleman
SPKAC	Netscape signed public key and challenge (ASN.1 structure)

Bibliography

[1] Internet Draft, Cisco Systems' Simple Certificate Enrollment Protocol SCEP, draft-nourse-scep-05

[2] Netscape Certificate Database Information
Available at : http://www.drh-consultancy.demon.co.uk/cert7.html

[3] Netscape Communicator Key Database Format
Available at : http://www.drh-consultancy.demon.co.uk/key3.html

[4] Mozzila.org - Using the Certificate Database Tool
Available at : http://www.mozilla.org/projects/security/pki/nss/tools/certutil.html

[5] Netscape DB KeyStore Research Edition 1.0
Available at : http://agora.sei.cmu.edu/ndbs10/

[6] The Cryptography API, or How to Keep a Secret
Availabale at :
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncapi/html/msdn_cryptapi.asp

[7] Erich Gamma et al; Design Patterns, Elements of Reusable Object-Oriented Software.

[8] Schneier, Bruce, Applied Cryptography, 2nd Edition, Wiley, 1996.

[9] RSA Laboratories; Public Key Cryptography Standard #1 (RSA Cryptography standard). Available at: http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html

[10] RSA Laboratories; Public Key Cryptography Standard #7 (Cryptographic Message Syntax Standard). Available at: http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/index.html

[11] RSA Laboratories; Public Key Cryptography Standard #9 (Selected Attribute Types). Available at: http://www.rsasecurity.com/rsalabs/pkcs/pkcs-9/index.html

[12] RSA Laboratories; Public Key Cryptography Standard #9 (Certification Request Syntax Standard). Available at: http://www.rsasecurity.com/rsalabs/pkcs/pkcs-10/index.html

[13] RSA Laboratories; Public Key Cryptography Standard #9 (Personal Information Exchange Syntax Standard). Available at: http://www.rsasecurity.com/rsalabs/pkcs/pkcs-12/index.html

[14] Kaliski, Burton S., Jr., "An Overview of the PKCS Standards", An RSA Laboratories Technical Note, revised November 1, 1993. Available at: http://www.rsa.com/rsalabs/pubs/PKCS/

[15] Netscape Certificate Specifications, available at http://www.netscape.com/eng/security/certs.html

[16] ACME Sere, a HTTP server. Available at http://http://www.acme.com/java/software/Acme.Serve.Serve.html/

[17] Bouncy Castle Crypto API, a lightweight, opensoure, Javacryptography API. Available at: http://www.bouncycastle.org

[18] OpenSCEP, an opensource server implementation of SCEP. Available at:
http://openscep.othello.ch

[19] Apache ANT, a java based build tool. Available at: http://jakarata.apache.org/ant/index.html

[20] The Linux FreeS/WAN project. Available at: http://www.freeswan.org