# TLS Extensions

**iSaSiLk** Version 6

# Contents

# 1 Introduction

This manual describes how to use TLS extensions with the iSaSiLk TLS library. The reader should have some knowledge about the basic principles of the Transport Layer Security ([TLS]) protocol and may have already used iSaSiLk for writing some simple client/server applications with standard cipher suites.

This manual is organized as followed: the second chapter gives a short introduction into the idea of TLS extensions; for a detailed description refer to the TLS extension specification ([RFC 4366]) of the TLS working group. The third chapter provides a short installation guide of the iSaSiLk library. The final fourth chapter gives a detailed survey about the extensions supported by iSaSiLk, based on many source code samples. A client-server demo is included in the `src/demo/extensions` directory of your iSaSiLk distribution.

# 2 TLS extensions

TLS extensions are defined in [RFC 4366]). They provide a general mechanism for extending the TLS protocol about security features that have to be negotiated at the beginning of the TLS handshake.

Each extension is identified by a unique type number. If a client wants to use some particular extension(s) it starts the handshake by sending an extended *ClientHello* message containing the numbers (and maybe additional information) of all extensions it wants to use.

The server, after having parsed the extension list received from the client, responds with an extended *ServerHello* message containing all those extensions the server is willing to accept. The extension list of the server anytime has to be a subset of the extension list received from the client. It may contain all extensions of the client list, but it cannot contain more extensions than those got from the client; i.e. the server is not allowed to enable extensions that have not been suggested by the client. By exchanging the extended *Hello* messages both client and server have to agree on the same set of extensions.

RFC 4366 itself specifies six standard extensions which are all supported by iSaSiLk[1]:

- Server Name Indication (type no. 0)
- Maximum Fragment Length Negotiation (type no. 1)
- Client Certificate URLs (type no. 2)
- Trusted CA Indication (type no. 3)
- Truncated HMAC (type no. 4)
- Certificate Status Request (type no. 5)

The *Server Name Indication* extension allows the client to specify which server name it expects in the certificate of the server. The *Maximum Fragment Length* extension may be negotiated to reduce the maximum TLS record fragment length for saving bandwidth. Instead of sending his full certificate chain, the client may provide a list of *Client Certificate URLs* from which the server can download the client certificate(s). The *Truncated HMAC* extension may be used to shorten the output size of the MAC calculation. And finally, by sending a *Certificate Status Request* extension, the client may request that the server provides a *CertificateStatus* message containing revocation status information about the server certificate.

Each of the extensions above will influence the TLS handshake in a different way. Please refer to the TLS extension specification ([RFC 4366]) for a detailed description of all extensions. Their usage with iSaSiLk is discussed in chapter 4 (*TLS extensions and iSaSiLk*).

[1] Note that this manual does not cover extensions that are not specified in RFC 4366. Examples are *RenegotiationInfo* (RFC 5746), *SupportedEllipticCurves* and *SupportedPointFormats* (RFC 4492), *SignatureAlgorithms* (RFC 5246, TLS 1.2) or TLS 1.3 extensions like KeyShare (RFC 8446). See the iSaSiLk Doc/JavaDoc and demo samples for more information.

# 3 Installation

There are no specific installation requirements other than when using iSaSiLk in common way without TLS extensions. All you need is a Java$^{TM}$ ([JAVA]) runtime environment (for instance JRE 1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x or 1.7.x).

After having downloaded (or received by CD) and unpacked the iSaSiLk distribution file (iSaSiLk<version>.zip) you will find the following folder hierarchy in your local iSaSiLk installation directory:

- `docs`: the Javadoc output
- `lib`: contains the iSaSiLk library files, `iaik_ssl.jar` + `iaik_ssl_demo.jar` (the demo classes), `w3c_http.jar` (W3C Jigsaw based https library), and the IAIK-JCE library files (`iaik_jce.jar`, signed and unsigned version[2])
- `manuals:` contains additional manuals like the one you are currently reading
- `demo/src`: Source-Code of the iSaSiLk demo programs
- `demo/lib`: library jar files used by the demo programs
- `demo/cmd:` Windows batch files for running the iSaSiLk demos
- `demo/sh:` shell scripts for running the iSaSiLk demos on Linux
- `images`: logos, ...

For running iSaSiLk you will have to put the `iaik_ssl.jar` file into your classpath. For the required cryptographic functionalities you will need a proper JCA/JCE ([JCA], [JCE]) provider. Most appropriate you will use the IAIK-JCE ([IAIK-JCE]) provider which is included in the iSaSiLk license. If not already done, get IAIK-JCE from the SIC/IAIK web site at http://jce.iaik.tugraz.at (please see the IAIK-JCE installation notes for specific JCA/JCE related requirements like provider registration/signing or key strength checking).

If you now want to try the TLS extension client/server demo included in the `iaik_ssl_demo.jar` file your classpath has to look like (assuming that all required SIC/IAIK library files are included in a lib sub-folder of your current working directory):

```
> set cp=lib/iaik_jce.jar;lib/iaik_jce_demo.jar;lib/iaik_ssl.jar;
```

First you will have to start the TLS extension demo server contained in package `demo.extensions` and then start the TLS extension client included in the same package:

[2] Because of patent reasons `iaik_jce.jar` does not contain implementations of the IDEA, ESDH, RC5 and RC6 algorithms. You can download the full IAIK-JCE version (`iaik_jce_full.jar`) from the SIC/IAIK web site http://jce.iaik.tugraz.at

```
> java –cp %cp% demo.extensions.TLSExtensionsServer
> java –cp %cp% demo.extensions.TLSExtensionsClient
```

You also can go to the `cmd/extensions` or `sh/extensions` directory and run the `runTlsExtensionsServer.bat` and `runTlsExtensionsClient.bat` batch scripts, or the `runTlsExtensionsServer.sh` and `runTlsExtensionsClient.sh` shell scripts, respectively.

# 4 TLS extensions and iSaSiLk

After having installed iSaSiLk you are now ready to extend your iSaSiLk based client/server applications about support for TLS extensions. This chapter will help you to select which extensions may be suitable for your environment and how to configure and use iSaSiLk with TLS extensions.

First we will describe the general extension framework of iSaSiLk and then we will discuss each of the supported extensions in detail.

## *4.1 The iSaSiLk extension framework*

The base class of all extensions is the abstract class `Extension` of package `iaik.security.ssl`:

```
public abstract class Extension;
```

Any class that implements some particular TLS extension is derived from class `Extension`. It defines the following basic properties that are available for each Extension object:

- The *type* of the extension
- The (type) *name* of the extension
- The *critical* specification of the extension

The *type* represents the type number that uniquely identifies some particular extension (see section 2 – *TLS Extensions*). The *name* is a short name of the extension according to the TLS extension specification ([RFC 4366]):

| | |
|---|---|
| o Server Name Indication: | `server_name` |
| o Maximum Fragment Length Negotiation: | `max_fragment_length` |
| o Client Certificate URLs: | `client_certificate_url` |
| o Trusted CA Indication: | `trusted_ca_keys` |
| o Truncated HMAC: | `truncated_hmac` |
| o Certificate Status Request: | `status_request` |

Extension type and name can be got from an `Extension` object by calling method `getExtensionType()` (which wraps type and name), or immediately by calling `getType()` or `getName()`, respectively:

```
Extension extension = ...;
int type = extension.getType();
String name = extension.getName();
```

**Listing 4-1: Querying an Extension object for type and name**

The *critical* specification of an Extension is not a standard extension property. It is defined by iSaSiLk to decide whether a TLS extension has to be treated as *critical* or *not critical*. If an `Extension` object is marked as being *critical*, iSaSiLk will abort the handshake if the peer does not include an extension of the same type into its extension list. For instance, if you have configured your iSaSiLk client to send a `server_name` extension, but the server does not respond with a `server_name` extension, the handshake will only continue if you have marked your local (client-side) `server_name` extension as *not critical*, but will be aborted if you have classified it as being *critical*:

```
Extension extension = ...;
extension.setCritical(true);
```

**Listing 4-2: Marking an Extension object as being critical**

In similar way, if you have configured your iSaSiLk server to support the `server_name` extension, but the client does not send a `server_name` extension, the handshake will be aborted if you have marked your local (server-side) `server_name` extension as *critical*.
Note that by default client-side extensions are classified as being critical (if you send a particular extension to the server you may want that the server responds with an extension of the same type), but server-side extensions are treated as being not-critical (you tell the server that it shall support some particular extension, but you may not insist on using this extension if it is not suggested by the client). Use method `setCritical` as shown in Listing 4-2 for changing the *critical* property of your extension.
Note also that the *critical* property of some particular extension may have some additional meaning depending on the definition of the extension in mind (see the extension survey of chapter 4.2 below).
So far we have learned that extensions are implemented as descendants of class `Extension` and that we can specify if some particular extension has to be treated as *critical* or *non critical*. However, how can we tell an iSaSiLk client or server which extensions it shall use? As you already might know an iSaSiLk client is configured by its `SSLClientContext`, and an iSaSiLk server is configured by its `SSLServerContext`. `SSLClient`- and `SSLServerContext`, respectively, contain all the TLS specific properties (like keys, certificates, supported ciphersuites, etc.) that are required to go into a TLS session with the peer. For using TLS extensions you must configure your `SSLClient`- or `SSLServerContext` with the specific extensions your client or server wants to support. Extensions can be set for an `SSLClient`- or `SSLServerContext` ob-

jects by calling its `setExtensions` method and thereby specifying an `ExtensionList` object containing all extensions that shall be enabled:

```
// create an ExtensionList object
ExtensionList extensionList = new ExtensionList();
...
// add any extensions
Extension extension = …;
extensionList.addExtension(extension);
...
// create SSLClientContext
SSLClientContext context = new SSLClientContext();
...
// extensions are only defined for TLS; set version to TLS1.0
context.setAllowedProtocolVersions(SSLContext.VERSION_TLS10,
                                   SSLContext.VERSION_TLS10);
...
// set extensions
context.setExtensions(extensionList);
...
// set keys, certificates, ciphersuites,…
...
// server name and port to connect to
String serverName = "...";
int serverPort = 443;
// create SSLSocket to connect to server
SSLSocket sslSocket = new SSLSocket(serverName, serverPort, context);
// proceed as usual
```

**Listing 4-3: Adding extensions to an SSLClientContext**

As you can see in the example of Listing 4-3 we have set the client-side protocol version to TLS1.0. This is required since extensions are defined for the TLS protocol only (but not for its SSL predecessor[3]).

[3] If you have configured your SSLContext to send extensions, iSaSiLk will also send them if SSLv3 is used (SSLv3 does not define extensions but allows to use it; therefor iSaSiLk will also parse extensions when SSLv3 is used)

Setting extensions on the server side is quite similar. However, use an `SSLServerContext` instead of an `SSLClientContext` and an `SSLServerSocket` instead of an `SSLSocket`:

```
// create an ExtensionList object
ExtensionList extensionList = new ExtensionList();
...
// add any extensions
Extension extension = ...;
extensionList.addExtension(extension);
...
// create SSLServerContext
SSLServerContext serverContext = new SSLServerContext();
// set extensions
serverContext.setExtensions(extensionList);
...
// set keys, certificates, ciphersuites,…
...
// the port to listen on client connections
int port = 443;
// create SSLServerSocket to listen on client connections
SSLServerSocket serverSocket =
  new SSLServerSocket(port, serverContext);
// proceed as usual
...
```

**Listing 4-4: Adding extensions to an SSLServerContext**

Note that at the time you add the extensions to an `SSLClient-` or `SSLServerContext` you already must have finished the extension configuration. Any changes that you make later to your `Extension` objects, are not adopted by the `Extension` objects that have been added to the SSLContext.
Class `ExtensionList` also provides methods for querying for or removing particular extensions (see iSaSiLk Javadoc). However, in general you only will have to add extensions and not to search for extensions or get information from extensions.

Now that we know how to add extensions to an iSaSiLk client- or server context, it is time to have a deeper look at the extensions supported by iSaSiLk.

## 4.2 *iSaSiLk extension survey*

As already mentioned in chapter 2 iSaSiLk supports all standard extensions defined by RFC 2546:

| | |
|---|---|
| o Server Name Indication: | `server_name` |
| o Maximum Fragment Length Negotiation: | `max_fragment_length` |
| o Client Certificate URLs: | `client_certificate_url` |
| o Trusted CA Indication: | `trusted_ca_keys` |
| o Truncated HMAC: | `truncated_hmac` |
| o Certificate Status Request: | `status_request` |

Depending on the iSaSiLk version you have got, it may also support additional extensions (like the *SessionTicket* extension specified in [RFC 4507]). However, this manual only explains the standard extensions specified by the TLS extension specification ([RFC 4366]) itself. Other extensions may be described in different manuals.
For each extension, the following discussion is grouped into three sections. First we give a brief introduction into the syntax and meaning of the extension itself. Subsequently we describe how the extension has to be configured and used on the client side, and finally we explain how it has to be used on the server side.
Note that this is mainly a user manual that shall help you to quickly get familiar with the iSaSiLk extension library. It does not describe all features and options that are provided by particular extension implementations (please refer to the iSaSiLk Javadoc for a detailed discussion), however, it covers the most common use cases.
For some extensions it only may be necessary to add the representing `Extension` objects to the `ExtensionList`. However, other extensions may require some additional configuration settings as described in the extension survey below.

### 4.2.1 Server Name Indication

Often it might be necessary to run multiple virtual servers at the same machine or ip address. Each virtual server may be addressed by a different name, and depending on to which server (name) the client actually is connecting to, a different certificate may have to be presented within the *Certificate* handshake message. When name-based virtual hosting is used several servers share the same ip address. It is the responsible of the application protocol (e.g. http) to direct the client request to the right virtual server (for instance, by looking at the value of the "Host"-header field of the http client request).

However, server authentication already has to take place during the TLS handshake and without having access to any application data. So far the TLS handshake protocol did not provide any functionality to let the server know which server name is expected to be contained into its certificate. With the *Server Name Indication* (`server_name`) extension now the client is able to provide server name information already within the *ClientHello* message. The `server_name` extension contains a list of server names that may be used by the server to select a proper certificate. If the server receives a `server_name` extension he includes an empty `server_name` extension into his extended *ServerHello* message indicating that he has recognized the `server_name` extension sent by the client.

#### 4.2.1.1 Client Side

The *Server Name Indication* is implemented by iSaSiLk class `ServerNameList`. For each server name you want announce to the server you will have to create a `ServerName` object and add it to your `ServerNameList` extension object. Subsequently add the `ServerNameList` to the extension list you set for your `SSLClientContext`. For instance, if you want to tell the server that you will accept the two server names "jce.iaik.tugraz.at" and "jce.iaik.at" add two `ServerName` objects to your server name list:

```
// create ServerNameList
ServerName[] serverNames = { new ServerName("jce.iaik.tugraz.at"),
                             new ServerName("jce.iaik.at") };
ServerNameList serverNameList = new ServerNameList(serverNames);
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(serverNameList);
...
// set extensions for the SSLClientContext configuration:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setExtensions(extensions);
...
```

**Listing 4-5: Client-side server name configuration**

However, it is not necessary to explicitly specify the server name(s) when creating the server name list. If you specify an empty `ServerNameList`, iSaSiLk tries to get the server name from the underlying transport layer. For instance, in the following example where you connect to server "jce.iaik.tugraz.at", iSaSiLk gets the server name ("jce.iaik.tugraz.at") from the SSLSocket transport object and sends it within its server name list in the extended *ClientHello* message:

```
// create empty ServerNameList
ServerNameList serverNameList = new ServerNameList();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(serverNameList);
...
// set extensions for the SSLClientContext configuration:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setExtensions(extensions);
...
// the host name of the server to connect to
String hostName = "jce.iaik.tugraz.at";
// the server port
int port = 443;
// create Socket
SSLSocket socket = new SSLSocket(hostName, port, clientContext);
...
```

**Listing 4-6: Let iSaSiLk get the server name from the transport layer**

As you already know all extensions by default are marked as *critical* on the client side. This means that if you send a *Server Name Indication* extension to the server, but the server does not send back a *Server Name Indication* extension or sends an "unrecognized_name" warning alert, iSaSiLk will abort the handshake and send an `internal_error` alert to the server. It also means that your iSaSiLk client will reject the server certificate if it does not contain any of the suggested server names, provided that you don’t have disabled certificate checking by disabling the `ChainVerifier` or overriding the `ChainVerifier` method `verifyServer` in a way to do not check the certificate server name(s) against the *Server Name Indication* extension.
If the server sends an "unrecognized_name" fatal alert, the handshake will be aborted in any case, regardless if the *Server Name Indication* extension is configured as critical or not critical.

#### 4.2.1.2 Server side

On the server side it is only necessary to configure iSaSiLk to support the *Server Name Indication* extension. Just add an empty `ServerNameList` extension object to the extension list of your SSLServerContext:

```
// create empty ServerNameList
ServerNameList serverNameList = new ServerNameList();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(serverNameList);
...
// set extensions for the SSLServerContext configuration:
SSLServerContext serverContext = new SSLServerContext();
...
serverContext.setExtensions(extensions);
```

**Listing 4-7: Enabling Server Name Indication extension support on the server side**

Server-side extensions are *not critical* by default. Setting your `ServerNameList` extension object to critical would mean that iSaSiLk would abort the handshake if the client does not send a server name list.
When getting a server name list from the client the iSaSiLk server will search its server credentials for a proper certificate. Thereby iSaSiLk tries to calculate the server names from the *SubjectAltName* extension, Netscape *SSLServerName* extension or subject DN common name of its server certificates. You may help iSaSiLk to determine the server name(s) associated with a certificate by explicitly specifying them for your server `KeyAndCerts`, e.g.:

```
// the cert chain of the server running at "jce.iaik.tugraz.at"
X509Certificate[] certChain = ...;
// the private key of the server
PrivateKey key = ...;
// create server credentials
KeyAndCert serverCredentials = new KeyAndCert(certChain, key);
// explicitly set server name(s)
ServerName[] serverNames = { new ServerName("jce.iaik.tugraz.at") };
serverCredentials.setTLSServerNames(serverNames);
// add credentials to your SSLServerContext
SSLServerContext serverContext = new SSLServerContext();
```

**Listing 4-8: Explicitly specifying the TLS server name(s) of your server credentials**

### 4.2.2 Maximum Fragment Length Negotiation

In constrained environments it may be desirable to use a shorter maximum fragment length than the TLS default value of $2^{14}$ (16384) bytes. If the client plans to use a smaller fragment length it includes a `max_fragment_length` extension into the extended *ClientHello* message. The `max_fragment_length` extension has to announce the new maximum fragment length value (either $2^9$ (512), $2^{10}$ (1024), $2^{11}$ (2048) or $2^{12}$ (4096) bytes) the client wants to use. If the server agrees to use a shorter maximum fragment length he sends back an empty `max_fragment_length` extension.

#### 4.2.2.1 Client side

The `max_fragment_length` extension is implemented by iSaSiLk class `MaxFragmentLength`. When creating a `MaxFragmentLength` extension object on the client side you have to specify the new maximum fragment value you want to use by referring one of the following static variables:

- `MaxFragmentLength.L_512` for using $2^9$ (512) bytes
- `MaxFragmentLength.L_1024` for using $2^{10}$ (1024) bytes
- `MaxFragmentLength.L_2048` for using $2^{11}$ (2048) bytes
- `MaxFragmentLength.L_4096` for using $2^{12}$ (4096) bytes

In the following example the client suggests to use a new maximum fragment length value of $2^{10}$ (1024) bytes:

```
// we want to use a max fragment length of 2^10 (1024)
int maxLen = MaxFragmentLength.L_1024;
// create MaxFragmentLength
MaxFragmentLength maxFragmentLength = new MaxFragmentLength(maxLen);
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(maxFragmentLength);
...
// set extensions for the SSLClientContext configuration:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setExtensions(extensions);
...
```

**Listing 4-9: The MaxFragmentLength extension has to contain the suggested max fragment value**

Again the client-side `MaxFragmentLength` extension is *critical* by default meaning that the handshake will be aborted if the server does not send back a `MaxFragmentLength` extension. However, using a non-critical `MaxFragmentLength` extension does not make any sense. A client that suggests to use a shorter maximum fragment length will run in a constrained environment and therefore may fail if the server ignores the extension and continues to use the default TLS maximum fragment length.
Regardless of having a *critical* or *non-critical* `MaxFragmentLength` extension, the handshake will be aborted with an `illegal_parameter` alert if the server sends back a different maximum fragment length value than that suggested by the client.

### 4.2.2.2 Server side

On the server side it is only necessary to configure iSaSiLk to support the `max_fragment_length` extension. Just add an empty `MaxFragmentLength` extension object to the extension list of your SSLServerContext:

```
// create empty MaxFragmentLength
MaxFragmentLength maxFragmentLength = new MaxFragmentLength();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(maxFragmentLength);
...
// set extensions for the SSLServerContext configuration:
SSLServerContext serverContext = new SSLServerContext();
...
serverContext.setExtensions(extensions);
...
```

**Listing 4-10: Enabling MaxFragmentLength extension support on the server side**

Server-side extensions are *not critical* by default. Setting your `MaxFragmentLength` extension object to *critical* would mean that iSaSiLk would abort the handshake if the client does not send `MaxFragmentLength` extension.
Regardless of having a *critical* or *non-critical* `MaxFragmentLength` extension, the handshake will be aborted with an `illegal_parameter` alert if the client suggests an illegal maximum fragment length value (not $2^9$ (512), $2^{10}$ (1024), $2^{11}$ (2048) or $2^{12}$ (4096)).

### 4.2.3 Client Certificate URLs

In a constrained environment a client may not be able to store the certificates it has to use if the server insists on client authentication. By sending an (empty) `client_certificate_url` extension the client indicates that it will not provide a *Certificate* message if the server requests that the client has to authenticate himself.

If the server agrees on using client certificate URLs the client will send a *CertificateURL* message (in place of the *Certificate* message) containing a list of URLs from where the server can download the client certificates. More precisely the *CertificateURL* message will contain a list of one or more of so-called `UrlAndOptionalHash` objects (see [RFC 4366]). Whether one or more `UrlAndOptionalHash` objects are included depends on the *certChainType* of the *CertificateURL* message, which may be `pkipath` or `individual_certs.` A pki path collects all client certificates together into an ASN.1 object and therefore requires only one single `UrlAndOptionalHash` object that points to the location from where the pki path can be obtained. When using cert chain type `individual_certs` you will need as many `UrlAndOptionalHash` objects as certificates are included in the client certificate chain. In this case each of the `UrlAndOptionalHash` objects will refer to one different certificate of the client certificate chain.

Regardless of using cert chain type `pkipath` or `individual_certs` the top level certificate of the client certificate chain may not be included in the `UrlAndOptionalHash` reference(s). The server may get it by other means to use it as trust anchor for the client certificate chain.

Note also that the certificates contained in a pki path are ordered in a way that the client (end) certificate is located at index [n-1] (where n is the number of certificates included in the pki path). This is reverse to the certificate chain order expected by TLS where the client certificate has to be located at index [0]. The `URLAndOptionalHash` objects of a *CertificateMessage* of type `individual_certs`, however, refer the certificates in TLS compliant way with the client certificate at index [0].

The optional hash of an `URLAndOptionalHash` object is a SHA-1 hash calculated over a DER encoded individual certificate or over the DER encoding of the entire pki path, respectively. If included, the server has to compare the hash value to one calculated over the DER encoded certificate or pki path got from the referenced URL. This check will prove that the certificates downloaded from the given URL(s) actually match to those indicated by the included hash value(s).

#### 4.2.3.1 Client Side

The client announces the usage of certificate URLs by sending an empty `client_certificate_url` extension. Thus use the empty default constructor for creating a `ClientCertificateURL` extension object on the client side:

```
// create empty ClientCertificateURL
ClientCertificateURL clientCertURL = new ClientCertificateURL();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(clientCertURL);
...
// set extensions for the SSLClientContext configuration:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setExtensions(extensions);
...
```

**Listing 4-11: The client sends an empty ClientCertificateURL extension**

By default the `ClientCertificateURL` extension is *critical* on the client side. Thus the handshake will be aborted if the server does not respond with a `ClientCertificateURL` extension if it has one received from the client.

When adding the `ClientCertificateURL` extension to the extension list of your SSLClientContext iSaSiLk knows that it shall send client certificate URLs instead of certificates. However, iSaSiLk does not know which URLs (or `URLAndOptionalHash` list) should be sent within the *CertificateURL* handshake message.
Usually you add client credentials as `KeyAndCert` objects to your SSLClientContext to tell iSaSiLk which keys and certificates shall be used for client authentication:

```
// the cert chain of the client
X509Certificate[] certChain = ...;
// the private key of the client
PrivateKey key = ...;
// create client credentials
KeyAndCert clientCredentials = new KeyAndCert(certChain, key);
// add credentials to your SSLClientContext
SSLClientContext clientContext = new SSLClientContext();
clientContext.addClientCredentials(clientCredentials);
```

**Listing 4-12: Client credentials are usually provided as `KeyAndCert` objects**

If you want to use client certificate urls you must specify your client credentials as instances of class `KeyAndCertURL`. Like a `KeyAndCert` object a `KeyAndCertURL` object

also will hold a private key of the client. However, instead of containing the client certificate chain, it only will refer it by means of `URLAndOptionalHash` object(s).
For convenience class `KeyAndCertURL` provides a constructor that can be featured with certificate and URL values to automatically calculate the required `URLAndOptionalHash` objects. However, client certificate urls are intended for use on constrained clients where the certificates may not be available. Thus you first will have to create the `URLAndOptionalHash` object(s) from the URL(s) that refer to the client certificate(s). If you want to include the hash value(s) of the referred individual certificates or pki path, you already will have calculated it on another, not constrained platform. When finally creating the `KeyAndCertURL` object from the `URLAndOptionalHash` list also provide the private key of the client and the cert chain type (`pki_path` or `individual_certs`) of the *CertificateURL* message to be sent. Furthermore you will have to specify if the client certificate shall be used for RSA signing, DSA signing, or RSA or DSA based DH key exchange (see Javadoc). This information is later used to check if the `KeyAndCertURL` is appropriate for the certificate types the server may sent within its *CertificateRequest* message.

```
// URLAndOptionalHash referring a pki path with the client certs
String pkiPathUrl = ...;
URLAndOptionalHash pkiPathUrlAndHash =
  new URLAndOptionalHash(pkiPathUrl);
// optional hash calculated over the DER encoded pki path
byte[] pkiPathHash = ...;
pkiPathUrlAndHash.setHash(pkiPathHash);
// create URLAndOptionalHash list containing one element:
URLAndOptionalHash[] urlAndOptionalHashList = { pkiPathUrlAndHash };
// create client credentials (certificate type is RSA_SIGN)
int certType = SSLContext.CERTTYPE_RSA_SIGN;
// the private key of the client:
PrivateKey privateKey ...;
// cert chain type is pki_path:
int certChainType = KeyAndCertURL.CHT_PKI_PATH;
// create KeyAndCertURL
KeyAndCertURL clientCredentials =
  new KeyAndCertURL(certType, privateKey, certChainType,
                    urlAndOptionalHashList);
// add credentials to your SSLClientContext
SSLClientContext clientContext = new SSLClientContext();
clientContext.addClientCredentials(clientCredentials);
```

**Listing 4-13: Creating Client credentials as KeyAndCertURL of cert chain type pki_path**

The example of Listing 4-13 creates a `KeyAndCertURL` object with chain type `pki_path` containing for an RSA signing key and referring to an URL from where the client certificate chain can be obtained.
However, if each certificate of the client chain is referenced by a different URL you will have to use cert chain type `individual_certs` and create one separate `URLAndOptionalHash` object for each certificate you have to refer to. For instance, if the client certificate chain consists of three certificates, you will have to create an `URLAndOptionalHash` list that consists of two elements pointing to the client end certificate and the intermediate CA certificate of the chain (the self-signed top level CA certificate may not be referenced since it has to be obtained by the server by other means):

```
// URLAndOptionalHash for client cert url
String clientCertUrl = ...;
URLAndOptionalHash clientUrlAndHash =
  new URLAndOptionalHash(clientCertUrl);
// optional hash calculated over the DER encoded client cert
byte[] clientCertHash = ...;
clientUrlAndHash.setHash(clientCertHash);
// URLAndOptionalHash for intermediate ca cert url
String caCertUrl = ...;
URLAndOptionalHash caUrlAndHash =
  new URLAndOptionalHash(caCertUrl);
// optional hash calculated over the DER encoded ca cert
byte[] caCertHash = ...;
caUrlAndHash.setHash(caCertHash);
// create URLAndOptionalHash list containing two elements:
URLAndOptionalHash[] urlAndOptionalHashList =
  { clientUrlAndHash, caUrlAndHash };
// create client credentials (certificate type is RSA_SIGN)
int certType = SSLContext.CERTTYPE_RSA_SIGN;
// the private key of the client:
PrivateKey privateKey ...;
// cert chain type is individual_certs:
int certChainType = KeyAndCertURL.CHT_INDIVIDUAL_CERTS;
// create KeyAndCertURL
KeyAndCertURL clientCredentials =
  new KeyAndCertURL(certType, privateKey, certChainType,
                    urlAndOptionalHashList);
...
```

**Listing 4-14: Creating Client credentials as KeyAndCertURL of cert chain type individual_certs**

#### 4.2.3.2 Server Side

On the server side it is only necessary to configure iSaSiLk to support the `client_certificate_url` extension. Just add an empty `ClientCertificateURL` extension object to the extension list of your SSLServerContext:

```
// create empty ClientCertificateURL
ClientCertificateURL clientCertURL = new ClientCertificateURL();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(clientCertURL);
...
// set extensions for the SSLServerContext configuration:
SSLServerContext serverContext = new SSLServerContext();
...
serverContext.setExtensions(extensions);
...
```

**Listing 4-15: Enabling ClientCertificateURL extension support on the server side**

Server-side extensions are *not critical* by default. Setting your `ClientCertificateURL` extension object to *critical* would mean that iSaSiLk would abort the handshake if the client does not send a `ClientCertificateURL` extension.
If the server is not able to download the client certificates from the referenced URLs it will send a `certificate_unobtainable` alert; if the any of the hash comparison checks will fail the server will abort the handshake and send a `bad_certificate_hash_value` alert (see [RFC 4366]).

### 4.2.4 Trusted CA Indication

The *Trusted CA Indication* (`trusted_ca_keys`) extension gives the client the possibility to tell the server which ca keys the client is willing to trust. The server may use this information to select a proper certificate when authenticating itself to the client. This practice may reduce the number of handshake trials and maybe useful for constrained clients that do not want to maintain a large number of trust anchor certificates.

Within its extended *ClientHello* message the client sends a list of trusted authorities to the server. The trusted authorities list may reference the ca keys by a prior agreement (in this case no further information is required), by a SHA-1 hash of the ca public key, by the DER encoded distinguished name of the ca, or by a SHA-1 hash of the DER encoded CA certificate. The server confirms to use the information provided by the client by sending back an empty trusted authorities list.

#### 4.2.4.1 Client Side

The `trusted_ca_keys` extension is implemented by class `TrustedAuthorities` which represents the main structure of the `trusted_ca_keys` extension (see [RFC 4366]). When creating the `TrustedAuthorities` list to be sent to the server, the client has to decide which identifier type it wants to use:

- `TrustedAuthority.ID_PRE_AGREED`: prior agreement between client and server
- `TrustedAuthority.ID_KEY_SHA1_HASH`: SHA-1 hash of the CA public key
- `TrustedAuthority.ID_X509_NAME`: DER encoded distinguished name of the CA
- `TrustedAuthority.ID_CERT_SHA1_HASH`: SHA-1 hash of the DER encoded CA certificate

You may explicitly create a `TrustedAuthority` object for any trusted CA you want to reference in the `TrustedAuthorities` list. Or – more simply – you may only specify the identifier type to be used and let iSaSiLk itself calculate the required `TrustedAuthority` elements from the trust anchor certificates you have set for your client ChainVerifier:

```
// the identifier type to be used:
int idType = TrustedAuthority.ID_CERT_SHA1_HASH
// create TrustedAuthorities list for the requestd identifier type
TrustedAuthorities trustedAuthorities =
  new TrustedAuthorities(idType);
// add to ExtensionList:
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(trustedAuthorities);
...
// create SSLClientContext:
SSLClientContext clientContext = new SSLClientContext();
...
// add any trust anchors
X509Certificate trustedCert = ...;
clientContext.addTrustedCertificate(trustedCert);
...
// set extensions
clientContext.setExtensions(extensions);
...
```

**Listing 4-16: Client side TrustedAuthorities extension configuration**

In the example of Listing 4-16 we use the `TrustedAuthority` identifier type `cert_sha1_hash`. This means that iSaSiLk will calculate a SHA-1 hash of the DER encoding of any of the trusted CA certificates and adds a `TrustedAuthority` object containing this hash to the `TrustedAuthorities` list. Since `cert_sha1_hash` is the default identifier type used by iSaSiLk you will get the same result when creating your `TrustedAuthorities` list by means of its default constructor. Note that iSaSiLk will calculate the trusted authorities when you add the extension list to your SSLClientContext. Thus any trust anchors must have been already set at this time. If no trusted CA certificates are set iSaSiLk does not send a TrustedAuthorities list to the server.
As you may know iSaSiLk tries to be crypto provider-independent where possible. However, calculating TrustedAuthority identifiers in provider independent way can only be done for the identifier types `pre_agreed`, `cert_sha1_hash` and (partially) `key_sha1_hash`. Distinguished names cannot be handled generally and a `key_sha1_hash` identifier can be only calculated for RSA keys (because some ASN.1 processing is required for calculating the ids of other key types). However, this all is no problem if you are using iSaSiLk with the IAIK-JCE provider which is enabled by default and supports all identifier types. If you are using a different crypto provider you may customize the iSaSiLk `SecurityProvider` by overriding its method `calculateTrustedAuthorityIdentifier` (see the Javadoc and `SecurityProvider` documentation included in the iSaSiLk distribution).

As all client-side extensions, the `TrustedAuthorities` extension is critical on the client side by default. This means that the handshake will be aborted if the server does not send back a `TrustedAuthorities` extension.

### 4.2.4.2 Server Side

On the server side it is only necessary to configure iSaSiLk to support the `trusted_ca_keys` extension. Just add an empty `TrustedAuthorities` object to the extension list of your SSLServerContext:

```
// create empty TrustedAuthorities
TrustedAuthorities authorities = new TrustedAuthorities();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
extensions.addExtension(authorities);
...
// set extensions for the SSLServerContext configuration:
SSLServerContext serverContext = new SSLServerContext();
...
serverContext.setExtensions(extensions);
```

**Listing 4-17: Enabling TrustedAuthorities extension support on the server side**

Server-side extensions are *not critical* by default. Setting your `TrustedAuthorities` extension object to *critical* would mean that iSaSiLk would abort the handshake if the client does not send a `TrustedAuthorities` extension.

### 4.2.5 Truncated HMAC

The *Truncated HMAC* extension allows to save bandwidth by truncating the output size of the MAC calculation to 80 bits. For using truncated HMACs client and server have to exchange empty Truncated HMAC extensions within their *ClientHello* and *ServerHello* messages, respectively.

#### 4.2.5.1 Client Side

A client that wants to use truncated HMACs includes an empty `TruncatedHMAC` extension into the extension list of his extended *ClientHello* message:

```
// create empty TruncatedHMAC
TruncatedHMAC truncatedHMAC = new TruncatedHMAC();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(truncatedHMAC);
...
// set extensions for the SSLClientContext configuration:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setExtensions(extensions);
```

**Listing 4-18: Adding a TruncatedHMAC extension to your client extension list**

By default a client-side `TruncatedHMAC` extension is considered as critical. This means that the handshake will be aborted if the client sends a `TruncatedHMAC` extension but the server does not respond with a `TruncatedHMAC` extension.

#### 4.2.5.2 Server Side

On the server side enable support for Truncated HMACs by adding an empty `TruncatedHMAC` extension to your extension list:

```
// create empty TruncatedHMAC
TruncatedHMAC truncatedHMAC = new TruncatedHMAC();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
extensions.addExtension(truncatedHMAC);
...
// set extensions for the SSLServerContext configuration:
SSLServerContext serverContext = new SSLServerContext();
...
serverContext.setExtensions(extensions);
```

**Listing 4-19: Enabling TruncatedHMAC extension support on the server side**

Server-side extensions are *not critical* by default. Setting your `TruncatedHMAC` extension object to *critical* would mean that iSaSiLk would abort the handshake if the client does not send a `TruncatedHMAC` extension.

### 4.2.6 Certificate Status Request

A client may send a *Certificate Status Request* extension to save resources when getting status information about the server certificate. When the server receives a certificate status request from the client (and confirms on using it) he may send a *CertificateStatus* message immediately after his *Certificate* message. The *CertificateStatus* message contains status information about the server certificate. This relieves the client from the task to obtain the server certificate status information from some other source.

The *Certificate Status Reques*t extension is not bound to a particular status type. However, most commonly the Online Certificate Status Protocol ([OCSP]) will be used for providing revocation status information about the server certificate. OCSP currently is also the only predefined status type specified for the *Certificate Status Request* extension (see [RFC 4366]).

When the client sends an OCSP *Certificate Status Request* extension to the server it may include the IDs of accepted OCSP responders, and any number of request extensions (like for instance a *Nonce* extension to protect against replay attacks). The format of responder IDs and request extensions are specified by the [OCSP] standard.

If responder IDs are included in the OCSP *Certificate Status Request* extension, they may guide the server when selecting an OCSP responder for getting certificate revocation information about his certificate. However, most usually the server may only contact one OCSP responder for getting status information, thus including responder IDs in the *Certificate Status Request* extension may not be required. Any request extensions contained in an O*CSP Certificate Status Request* extension will be passed further to the OCSP responder.

The following client-server discussion assumes that you want to send an OCSP certificate status request and that you are using IAIK-JCE as underlying cryptographic provider. IAIK-JCE is enabled by default. iSaSiLk allows to use the Certificate Status Request extension with other crypto providers, too. However, in this case you must implement and plug-in the required status request and status response en/decoding and processing classes since OCSP cannot be handled in crypto provider independent way.

#### 4.2.6.1 Client Side

The simplest way for instructing iSaSiLk to send an OCSP *Certificate Status Request* extension is to create and add an empty `CertificateStatusRequest` extension object to the extension list of your client:

```
// create empty CertificateStatusRequest extension
CertificateStatusRequest certStatusRequest =
  new CertificateStatusRequest();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(certStatusRequest);
...
// set extensions for the SSLClientContext configuration:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setExtensions(extensions);
...
```

**Listing 4-20: Instructing iSaSiLk to request OCSP certificate status information from the server**

By adding an empty `CertificateStatusRequest` extension object to the extension list of your client you tell iSaSiLk to send a *Certificate Status Message* of type OCSP to the server. To protect against replay attacks iSaSiLk automatically will add a *Nonce* request extension to the status request. If you want to include any further (or other) request extensions or responder IDs into the request message, you will have to explicitly create and set an `OCSPStatusRequest` when creating the `CertificateStatusRequest` extension object (see the iSaSiLk Javadoc for detailed information):

```
// the responder IDs to be announced to the server
ResponderID[] responderIDs = ...;
// request extensions
OCSPExtensions requestExtensions = ...;
// create OCSPStatusRequest
OCSPStatusRequest ocspStatusRequest = new
  OCSPStatusRequest(responderIDs, requestExtensions);
// create CertificateStatusRequest extension of type ocsp:
CertificateStatusRequest certStatusRequest =
  new CertificateStatusRequest(OCSPStatusRequest.STATUS_TYPE,
                               ocspStatusRequest.getEncoded());
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(certStatusRequest);
...
// set extensions for the SSLClientContext configuration:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setExtensions(extensions);
...
```

**Listing 4-21: Explicitly creating an OCSP certificate status request**

By default a client-side `CertificateStatusRequest` extension is marked as being *critical*. This means that the handshake will be aborted if the server does not send back a `CertificateStatusRequest` extension. The handshake will also be aborted if the server does not provide a *Certificate Status* message containing status information about the server certificate.

Till now we only have configured our iSaSiLk client to request status information from the server. However, we also must take care that the iSaSiLk client actually verifies the status information provided from the server. For that purpose we use an instance of `OCSPCertStatusChainVerifier` instead of the standard iSaSiLk `ChainVerifier` for server certificate chain verification. The `OCSPCertStatusChainVerifier` works in the same way as the standard iSaSiLk `ChainVerifier`, except that it uses the OCSP status information sent by the server for checking the revocation status of the server certificate:

```
// create an OCSPCertStatusChainVerifier:
OCSPCertStatusChainVerifier chainVerifier =
  new OCSPCertStatusChainVerifier();
...
// set OCSPCertStatusChainVerifier for the SSLClientContext:
SSLClientContext clientContext = new SSLClientContext();
...
clientContext.setChainVerifier(chainVerifier);
```

**Listing 4-22: Use an OCSPCertStatusChainVerifier for verifying status information from the server**

If you expect that you have to verify status responses coming from a delegated OCSP responder (where the signer of the OCSP response is different from the signer of the server certificate), you may want to check if the responder is authorized to sign an OCSP response for the server certificate. According to the OCSP specification ([OCSP]) it is required that the signer certificate of a delegated OCSP responder contains an *ExtendedKeyUsage* extension of type `ocspSigning` and must be signed by the same CA certificate that has issued the (server) certificate for which status information is provided.
If you want that the OCSP cert status chain verifier checks responder authorization, you may configure it with trusted responder information. Just add a `TrustedResponders` object to your `OCSPCertStatusChainVerifier`. For each responder you want to check add an entry to the `TrustedResponders` telling that the referenced responder is authorized by the given CA certificate for signing OCSP responses on behalf of this CA:

```
// server certificate chain
X509Certificate[] serverCerts = ...;
// OCSP responder certificate
X509Certificate responderCert = ...;
TrustedResponders trustedResponders = new TrustedResponders();
ResponderID responderID =
  new ResponderID((Name)responderCert.getSubjectDN());
trustedResponders.addTrustedResponderEntry(responderID,
                                           serverCerts[1]);
// add to ChainVerifier
OCSPCertStatusChainVerifier chainVerifier =
  new OCSPCertStatusChainVerifier();
...
chainVerifier.setTrustedResponders(trustedResponders);
```

**Listing 4-23: Adding trusted responders to the OCSP cert status chain verifier**

In the example of Listing 4-23 we configure the OCSP cert status chain verifier to accept the given responder as being authorized for signing OCSP responses for certificates issued by the server certificate issuer (i.e. the CA certificate contained in `serverCerts[1]`).

### 4.2.6.2 Server Side

On the server side enable support for the *Certificate Status Request* extension by adding an empty `CertificateStatusRequest` extension object to your extension list:

```
// create empty CertificateStatusRequest
CertificateStatusRequest statusRequest =
  new CertificateStatusRequest();
// add to ExtensionList
ExtensionList extensions = new ExtensionList();
...
extensions.addExtension(statusRequest);
...
// set extensions for the SSLServerContext configuration:
SSLServerContext serverContext = new SSLServerContext();
...
serverContext.setExtensions(extensions);
...
```

**Listing 4-24: Enabling CertificateStatusRequest extension support on the server side**

Server-side extensions are *not critical* by default. Setting your `CertificateStatusRequest` extension object to *critical* would mean that iSaSiLk would abort the handshake if the client does not send `CertificateStatusRequest` extension.
In addition to configure your iSaSiLk server for supporting the *Certificate Status Request* extension you also must tell him where he can get (OCSP) status information for his server credentials. Instead of using `KeyAndCert` objects when adding the server credentials to your SSLServerContext use instances of class `OCSPCertStatusKeyAndCert`. In addition to private key and certificate chain of the server, an `OCSPCertStatusKeyAndCert` object also may contain the URL of the OCSP responder which has to be contacted for getting revocation status information about the corresponding server certificate (note that only http-URLs are supported).
Optionally an `OCSPCertStatusKeyAndCert` object may contain a list that maps responder IDs to responder URLs. This information may help the server to search for the responder url of a particular responder id that may be sent by the client within the *Certificate Status Request* extension:

```
// the server certificate chain
X509Certificate[] serverCerts = ...;
// the private key of the server
PrivateKey serverKey = ...;
// the URL of the OCSP responder
String responderUrl = "http://...";
// create server credentials
OCSPCertStatusKeyAndCert kayAndCert =
  new OCSPCertStatusKeyAndCert(serverCert, serverKey, responderUrl);
...
// optionally specify responderID-URL mappings
// the public key of a responder
PublicKey responderPublicKey = ...;
// create responder id
ResponderID byKeyID = new ResponderID(serverPublicKey);
String responderUrl = "http://...";
kayAndCert.addOCSPResponder(byKeyID, responderUrl);
// the Name of a responder
Name responderName = ...;
// create responder id
ResponderID byNameID = new ResponderID(responderName);
responderUrl = "http://...";
kayAndCert.addOCSPResponder(byNameID, responderUrl);
// add credentials to server context
SSLServerContext serverContext = new SSLServerContext();
serverContext.addServerCredentials(keyAndCert);
...
```

**Listing 4-23: Adding trusted responders to the OCSP cert status chain verifier**
**Listing 4-25: Specifying server credentials with OCSP responder URL(s)**

# 5 Acronyms

| | |
|---|---|
| ASN.1 | Abstract Syntax Notation One; language for describing data structures in an abstract and platform independent manner |
| DER | Distinguished Encoding Rules; set of rules for non ambiguously binary encoding ASN.1 objects |
| DH | Diffie-Hellman; public key algorithm; used for key exchange |
| DSA | Digital Signature Algorithm; public-key digital signature algorithm, standardized by the Digital Signature Standard (DSS) |
| RSA | Public-key algorithm, developed by Ron Rivest, Adi Shamir and Leonard Adleman; may be used for data encryption or digital signing. |
| OCSP | Online Certificate Status Protocol; protocol for providing revocation status information about X.509 certificates |
| TLS | Transport Layer Security; IETF standardaized successor of the SSL (Secure Socket Layer) protocol |
| X.509 | ITU-T (International Telecommunication Union) recommendation for an authentication system and certificate syntax; profiled by the PKIX working group of the IETF |

# 6 References

[IAIK-JCE] IAIK-JCE Provider, IAIK, Stiftung SIC, 2003, http://jce.iaik.tugraz.at

[JAVA] Java$^{TM}$ Technology, Sun Microsystems, Inc., http://java.sun.com/

[JCA] Java$^{TM}$ Cryptography Architecture API Specification & Reference, SUN Microsystems, Inc.,

http://java.sun.com/j2se/1.4/docs/guide/security/CryptoSpec.html

[JCE] Java$^{TM}$ Cryptography Extension (JCE) API Specification & Reference, SUN Microsystems, Inc.,

http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html

[OCSP] M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams: "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", June 1999

[TLS] T. Dierks and C. Allen: "The TLS Protocol Version 1.0", RFC 2246, January 1999.

T. Dierks and E. Rescorla, "The TLS Protocol, Version 1.1", RFC 4346, April, 2006.

[RFC 4366] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, T. Wright: "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006

[RFC 4507] J. Salowey, H. Zhou, P. Eronen, H. Tschofenig: "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 4507, May 2006