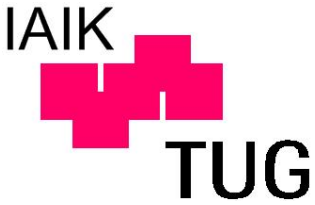

PSK Cipher Suites

iSaSiLk Version 6

Contents

1 Introduction

This manual describes how PSK cipher suites can be used with the SIC/IAIK TLS library iSaSiLk. 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 first chapter gives a short introduction into the idea of PSK cipher suites; for a detailed discussion refer to the PSK specification ([PSK]) of the TLS working group. The second and third chapters provide quick installation and usage guides of the iSaSiLk PSK implementation. The fourth chapter gives a detailed discussion about how to configure and use iSaSiLk with PSK cipher suites, and the final fifth chapter illustrates the usage of PSK cipher suites by means of a simple client/server sample program.

2 PSK Cipher Suites

Most commonly client/server authentication during a TLS handshake is performed by means of public key certificates (see [TLS]). Especially for constrained or closed environments the TLS working group has defined an alternative group of cipher suites using pre-shared keys for peer authentication ([PSK]).

A pre-shared key is a symmetric key that has to be shared by client and server before the TLS communication can take place. Since each party may have more than only one pre-shared key (e.g. for communicating with different partners) it is necessary to negotiate which key shall be used for authenticating the current TLS session. If both parties have agreed upon using a PSK based cipher suite at the beginning of the handshake during exchanging the `Hello` messages, the client then uses the `ClientKeyExchange` message for sending a "PSK identity"[1] to announce which pre-shared key shall be used for the current session. If the server has a key that corresponds to the psk identity received from the client, both client and server use the same pre-shared key for generating the pre-master secret. The pre-master secret is used for calculating the master secret from which then the session keys are derived. Finally the handshake is completed by exchanging `ChangeCipherSpec` and `Finished` messages which only can be correctly built if both parties have used the same pre-shared key.

Depending on the underlying key management technique the PSK specification defines three sets of PSK cipher suites (see [PSK]), which are all supported by iSaSiLk:

(A) PSK cipher suites using only symmetric keys for authentication:

- TLS_PSK_WITH_RC4_128_SHA,
- TLS_PSK_WITH_3DES_EDE_CBC_SHA,
- TLS_PSK_WITH_AES_128_CBC_SHA,
- TLS_PSK_WITH_AES_256_CBC_SHA

(B) PSK cipher suites using a pre-shared key for authenticating a Diffie-Hellman key exchange:

- TLS_DHE_PSK_WITH_RC4_128_SHA,
- TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA,
- TLS_DHE_PSK_WITH_AES_128_CBC_SHA,
- TLS_DHE_PSK_WITH_AES_256_CBC_SHA

[1] For supporting the client in selecting the pre-shared key to be used, the server may send a “PSK identity hint” within the `ServerKeyExchange` message. However, sending a psk identity hint is not recommended by the specification ([PSK]). If not otherwise required by a particular application environment, the server SHOULD NOT send an identity hint and the client MUST ignore an identity hint if received from the server.

(C) PSK cipher suites using RSA based public key authentication of the server and mutual authentication with a pre-shared key:

- TLS_RSA_PSK_WITH_RC4_128_SHA,
- TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA,
- TLS_RSA_PSK_WITH_AES_128_CBC_SHA,
- TLS_RSA_PSK_WITH_AES_256_CBC_SHA

Only the first set does not use public key operations at all. The second set may provide perfect forward security, and the third set uses certificates for authenticating the server as done by common cipher suites. However, all three sets are based on the availability of pre-shared keys which have to be properly protected (see security considerations of [PSK]).

In all three sets the client uses the `ClientKeyExchange` message for sending the "PSK identity" to the server; however the client never sends a `Certificate` or `CertificateVerify` message.

The server only sends a `Certificate` message if RSA based PSK cipher suites are used (C), but he does not send a `CertificateRequest` message since certificate based client authentication is not necessary for PSK cipher suites. The `ServerKeyExchange` message is only sent if the server wants to provide a PSK identity hint (which is not recommended by the specification, see [PSK]), and/or Diffie-Hellman key exchange (B) is used where the server must send its public Diffie-Hellman parameters to the client.

An additional RFC (4785) extends the basic set of PSK cipher suites about none-encryption suites (see [PSK-NULL]):

- TLS_PSK_WITH_NULL_SHA,
- TLS_DHE_PSK_WITH_NULL_SHA,
- TLS_RSA_PSK_WITH_NULL_SHA

These three cipher suites are also supported by iSaSiLk, however, they only offer authentication but do not encrypt the data (NULL encryption method).

3 Installation

There are no specific installation requirements other than when using iSaSiLk with common cipher suites. All you need is a Java™ ([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:` 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 PSK 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_ssl.jar;lib/iaik_ssl_demo.jar
```

First you will have to start the PSK demo server contained in package `demo.psk` and then run the PSK demo client contained in the same package:

```
> java –cp %cp% demo.psk.PSKServer
```

[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.psk.PSKClient
```

You also can go to the `cmd/psk` or `sh/psk` directory and run the `runPSKServer.bat` and `runPSKClient.bat` batch scripts, or the `runPSKServer.sh` and `rundPSKClient.sh` shell scripts, respectively.

4 Quick-Start Guide

In this chapter we provide a quick introduction into the iSaSiLk PSK implementation without giving any deeper explanations. Based on some program fragments of a typical client/server example we show step-by-step how to use iSaSiLk with PSK cipher suites (see chapter 6 and iSaSiLk demo package `demo.psk` for detailed client/server examples).

First of all client and server have to agree on a pre-shared secret key. How this agreement is achieved is out of scope of the [PSK] specification. In this manual we assume that client and server already possess the same pre-shared key.

4.1 PSK Client

The following example shows how to write an iSaSiLk PSK client running on a host named “pskclient.iaik.tugraz.at” and talking with a server “pskserver.iaik.tugraz.at” using the same pre-shared key.

1. Create an `SSLClientContext` object and enable the PSK cipher suites you want to use. In our example we want to support all psk cipher suites.

```
// client context
SSLClientContext context = new SSLClientContext();
...
CipherSuiteList suites = new CipherSuiteList();
suites.add(CipherSuite.CS_ALL_PSK);
context.setEnabledCipherSuiteList(suites);
context.updateCipherSuites();
```

Listing 4-1: Setting PSK cipher suites for an SSLClientContext

2. Create a `PSKCredential` for your pre-shared key. Give the credential an identity – in our example we use the client name “pskclient.iaik.tugraz.at” – and specify the remote peer id of the server you want to connect to. Add the credential to your `SSLClientContext`.

```
// the server we want to connect to
String serverName = "pskserver.iaik.tugraz.at";
...
// the pre-shared key negotiated with the server out-of-band
PreSharedKey psk = ...;
// create PSKCredential with identity "pskclient.iaik.tugraz.at"
String identity = "pskclient.iaik.tugraz.at";
PSKCredential credential = new PSKCredential(psk, identity);
// set remote peer id of server
credential.setRemotePeerId(serverName);
...
// enable pre-shared key by adding it to the context
context.addPSKCredential(credential);
```

Listing 4-2: Creating and setting a PSKCredential for the pre-shared key

3. Create an `SSLSocket` to talk with the server and configure it with the `SSLClientContext`.

```
// create SSLSocket
int serverPort = 443;
SSLSocket socket = new SSLSocket(serverName,
                                 serverPort,
                                 context);
// print debug info to System.out
socket.setDebugStream(System.out);
```

Listing 4-3: Creating an SSLSocket for connecting to the server

4. Open streams on the socket and communicate with the server in the usual way by writing and reading from the streams, respectively.

```
// send GET-request
System.out.println("Sending HTTPS request to " + serverName);
PrintWriter writer =
  Utils.getASCIIWriter(socket.getOutputStream());
BufferedReader reader =
  Utils.getASCIIReader(socket.getInputStream());
writer.println("GET / HTTP/1.0");
writer.println();
writer.flush();

// read response
System.out.println("Reading response...");
while (true) {
  String line = reader.readLine();
  if (line == null) {
    break;
  }
  System.out.print(":");
  System.out.println(line);
}

// close streams and socket
writer.close();
reader.close();
socket.close();
```

Listing 4-4: Sending request to the server and reading response

4.2 PSK Server

On the server side the usage is quite similar except for that you must use an `SSLServerContext` instead an `SSLClientContext` for configuring the server. The server uses an `SSLServerSocket` to listen and accept client connection requests.

The following example represents a TLS server named “pskserver.iaik.tugraz.at” that uses a pre-shared key for TLS sessions with a client running on a host named “pskclient.iaik.tugraz.at”.

1. Create an `SSLServerContext` object and enable the PSK cipher suites you want to use. In our example we want to support all psk cipher suites.

```
// server context
SSLServerContext serverContext = new SSLServerContext();
...
CipherSuiteList suites = new CipherSuiteList();
suites.add(CipherSuite.CS_ALL_PSK);
serverContext.setEnabledCipherSuiteList(suites);
serverContext.updateCipherSuites();
```

Listing 4-5: Setting PSK cipher suites for an SSLServerContext

2. Create a `PSKCredential` for your pre-shared key. Give the credential an identity – in our example we use the client name "pskclient.iaik.tugraz.at. Add the credential to your `SSLServerContext`.

```
// the pre-shared key negotiated with the client out-of-band
PreSharedKey psk = …;
// create PSKCredential with identity "pskclient.iaik.tugraz.at"
String identity = "pskclient.iaik.tugraz.at";
PSKCredential credential = new PSKCredential(psk, identity);

// enable pre-shared key by adding it to the server context
serverContext.addPSKCredential(credential);
```

Listing 4-6: Creating and setting a PSKCredential for the pre-shared key

3. Create an `SSLServerSocket` to listen for client requests and configure it with the `SSLServerContext` to understand PSK cipher suites.

```
// create SSLServerSocket
int port = 443;
SSLServerSocket serverSocket =
  new SSLServerSocket(port, serverContext);
// print debug info to System.out
serverSocket.setDebugStream(System.out);
```

Listing 4-7: Creating an SSLServerSocket for listening on client requests

4. Wait, accept and handle client requests in the usual way (see chapter 5 for a detailed example).

```
// accept client request
while (true) {
  try {
    SSLSocket socket = (SSLSocket)serverSocket.accept();
    ...
  } catch( IOException e ) {
    e.printStackTrace(System.out);
  }
}
```

Listing 4-8: Waiting for client requests

5 Configuration and Usage

In this chapter we describe how to configure iSaSiLk for using pre-shared key based cipher suites. If you already have worked with iSaSiLk, you will know that TLS related security parameters like cipher suite(s), key(s) or certificates have to be specified by an `SSLClientContext` (client side) or `SSLServerContext` (server side) object, respectively. When, for instance, creating an `SSLSocket` for opening a connection to some TLS server, you have to configure the `SSLSocket` with an `SSLClientContext`:

```
// create SSLClientContext
SSLClientContext clientContext = new SSLClientContext();
...
// configure context
...
// the server to which to connect to
String serverName = "...";
int serverPort = 443;
// create SSLSocket to connect to the server
SSLSocket socket = new SSLSocket(serverName, serverPort,
clientContext);
// proceed as usual
...
```

Listing 5-1: SSLClientContext configuration

You are now ready to open in- and output streams on the `SSLSocket` object and exchange data with the server in quite the same way as accustomed from ordinary java.net Socket objects.

So far we only have created an `SSLClientContext` object and passed it as parameter to the `SSLSocket` object. For using pre-shared key based cipher suites we have to configure the `SSLClientContext` with information about

- which PSK based cipher(s) suite shall be used
- which pre-shared key(s) shall be used

5.1 Setting psk cipher suites

Cipher suites are represented as instances of class `CipherSuite`. Each implemented cipher suite can be referenced by the name of the corresponding static variable of class `CipherSuite`. To tell iSaSiLk which cipher suites shall be used for a particular TLS session you will have to use method `setEnabledCipherSuiteList` or `setEnabledCipherSuites` of your `SSLContext` object, for instance:

```
// create SSLClientContext
SSLClientContext clientContext = new SSLClientContext();
...
// create cipher suite list
CipherSuiteList csList = new CipherSuiteList();
// we only want to use one particular cipher suite
csList.add(CipherSuite.TLS_PSK_WITH_AES_128_CBC_SHA);
// enable and update cipher suite list
clientContext.setEnabledCipherSuiteList(csList);
clientContext.updateCipherSuites();
```

Listing 5-2: Setting the cipher suite list of an SSLClientContext

In this example we have told our iSaSiLk client to use only one particular psk based cipher suite, `TLS_PSK_WITH_AES_128_CBC_SHA`. If the server is able to support this cipher suite, too, the handshake will succeed; otherwise – if the server does not support the cipher suite – the handshake will fail. You can enable further cipher suites by repeatedly adding them to the `CipherSuiteList` object. For your convenience class `CipherSuite` also provides static array variables to specify groups of cipher suite objects. For use with pre-shared keys the following pre-defined cipher suite groups are available:

- `CipherSuite.CS_PSK`: cipher suites using only symmetric keys for authentication
- `CipherSuite.CS_DHE_PSK`: PSK cipher suites using a pre-shared key for authenticating a Diffie-Hellman key exchange
- `CipherSuite.CS_RSA_PSK`: PSK cipher suites using RSA based public key authentication of the server and mutual authentication with a pre-shared key
- `CipherSuite.CS_ALL_PSK`: all supported PSK cipher suites

To, for instance, enable all implemented PSK cipher suites use the last one of these array variables:

```
// create SSLClientContext
SSLClientContext clientContext = new SSLClientContext();
...
CipherSuite[] suites = CipherSuite.CS_ALL_PSK;
clientContext.setEnabledCipherSuites(suites);
clientContext.updateCipherSuites();
```

Listing 5-3: Enabling all PSK cipher suites

Be careful to only use cipher suites that your application environment is able to support. RSA based PSK cipher suites, for instance, only will work if RSA signature and encryption algorithms are supported by the registered cryptographic providers (which is no problem if you have installed the IAIK-JCE provider). Calling method `updateCipherSuites` checks cryptographic engine availability and will remove all unsupported cipher suites.

If you are on the server side – and are using RSA based PSK cipher suites – you also must ensure to set RSA private key and server certificate(s) as required for server authentication. DHE based PSK cipher suites will need (temporarily created) DH keys.

5.2 Setting the pre-shared key(s)

For being able to use a PSK based cipher suite both client and server must have the same pre-shared key. A pre-shared key is a symmetric key and therefore is implemented as JCE `SecretKey` (see package `iaik.security.ssl`):

```
public class PreSharedKey extends SecretKeySpec;
```

Since class `PreSharedKey` extends `SecretKeySpec` which implements the interface `javax.crypto.SecretKey`, a `PreSharedKey` object maybe used like any other `SecretKey` object. However, typically a pre-shared key will not be used with `Cipher` or `Mac` engines; it only is required by the iSaSiLk library for deriving the pre-master secret from it. There is no `KeyGenerator` or `KeyFactory` available for the `PreSharedKey` type. A `PreSharedKey` object simply may be created from the raw keying material:

```
byte[] keyBytes = …;
PreSharedKey psk = new PreSharedKey(keyBytes);
```

Listing 5-4: Creating a pre-shared key object

Implementing class `PreSharedKey` as `SecretKey` provides the possibility to use a Java™ KeyStore for securely storing pre-shared keys.

A pre-shared key is identified by a PSK identity. For that reason it is necessary to provide some means for binding a pre-shared key to its identity. This binding is provided by iSaSiLk class `PSKCredential` of package `iaik.security.ssl`:

```
public class PSKCredential;
```

When creating a new `PSKCredential` object it is necessary to specify the identity that shall be used to identify the pre-shared key. There does not exist a specific role how PSK identities have to be built and represented. For instance, an identity might be an IPv4 address (e.g. “129.0.0.1”), or a domain name (like “jce.iaik.tugraz.at”), or a X.500 distinguished name (e.g. “CN=jce.iaik.tugraz.at”). However, when encoding an identity, it first has to be converted to a character string and then encoded into octets according to the UTF-8 ([UTF8]) syntax (see [PSK]). For that reason a PSK identity can be specified in two ways when creating a `PSKCredential` object: as String object (see Listing 5-5), or as byte array representing the UTF-8 encoded identity character string (see Listing 5-6). For instance, client "pskclient.iaik.tugraz.at" and server "pskserver.iaik.tugraz.at" may have agreed on a pre-shared key that shall be identified by the name of the client. Both client and server can create the required `PSKCredential` in the same way by using "pskclient.iaik.tugraz.at" as identity String:

```
// PSK identity as String
String identityStr = "pskclient.iaik.tugraz.at";
// the pre-shared key
byte[] psk = ...;
// create PSKCredential
PSKCredential pskCredential = new PSKCredential(identityStr, psk);
```

Listing 5-5: Creating a PSKCredential from identity string and pre-shared key

or:

```
// PSK identity as UTF-8 encoded byte array
byte[] identity = "pskclient.iaik.tugraz.at".getBytes("UTF-8");
// the pre-shared key
byte[] psk = ...;
// create PSKCredential
PSKCredential pskCredential = new PSKCredential(identity, psk);
```

Listing 5-6: Creating a PSKCredential from identity bytes and pre-shared key

Since we may want to talk with more than only one communication partner, we may have to maintain a certain number of pre-shared keys. As already discussed, a pre-shared key is identified by a psk identity which is sent by the client to the server within the `ClientKeyExchange` message. At this time the server already may have used its `ServerKeyExchange` message to provide a psk identity hint for helping the client to select a proper pre-shared key. Although the usage of psk identity hints is not recom-

mended by the specification ([PSK]) we may have two similar situations on both, client and server side when searching for the right pre-shared key in the local repository:

1. The client, when going to talk with some specific server (e.g. "pskserver.iaik.tugraz.at") will have to search for a pre-shared key that has been previously shared with this server by other means. He may do so by searching based on the server name, IP address, or any other characteristic that is appropriate for identifying the server.
2. The client, when having received a psk identity hint from the server, may use this identity hint to search for the right pre-shared key.

On the server side we may have the following situations:

1. The server, when having received a `ClientHello` message indicating to use a PSK cipher suite, may wish to send a psk identity hint to the client. In this case the server will have to search for a pre-shared key that has been previously shared with this client by other means. He may do so by searching based on the client name, IP address, or any other characteristic that is appropriate for identifying the client.
2. The server, when having received the psk identity within the client key exchange message, may use this identity to search for the right pre-shared key.

In the first situation we search for the pre-shared key based on some information that identifies the peer. In the second situation we already have precise psk identity information that uniquely identifies (by identity or identity hint) some particular pre-shared key to be used.

We already have seen that the identity has to be specified when creating a `PSKCredential` for some particular pre-shared key. If you need to set a psk identity hint or a remote peer id, use method `setIdentityHint` or `setRemotePeerId`, respectively.

Like a psk identity, a psk identity hint may be given as String object or as byte array representing the UTF-8 encoded identity hint character string. For instance, server "pskserver.iaik.tugraz.at" may want to send a psk identity hint to tell the client to use the pre-shared key with identity (hint) "pskclient.iaik.tugraz.at":

```
// PSK identity hint as String
String identityHintStr = "pskclient.iaik.tugraz.at";
pskCredential.setIdentityHint(identityHintStr);
```

Listing 5-7: Setting identity hint string of a PSKCredentials

or:

```
// PSK identity hint as UTF-8 encoded byte array
byte[] identityHint =
  "pskclient.iaik.tugraz.at".getBytes("UTF-8");
pskCredential.setIdentityHint(identityHint);
```

Listing 5-8: Setting identity hint bytes of a PSKCredential

The format of the remote peer id depends on the transport mechanism that is used for the TLS communication. Most commonly you will use TCP based SSLSockets for running TLS over TCP. In this case the remote peer id may be given as, for instance, client/server name or ip address, e.g.:

```
String serverName = "pskserver.iaik.tugraz.at";
pskCredential.setRemotePeerId(serverName);
```

Listing 5-9: Setting the remote peer id of a PSKCredential

However, TLS is a transport independent protocol and for some reason it might be necessary to use a transport mechanism where the peer is identified by a characteristic that is not represented by a name or an ip address. For that reason method `setRemotePeerId` expects a general Java™ object as argument. This allows you to specify remote peer ids in accordance with the `SSLTransport` implementation that is used by your application.

In summary a `PSKCredential` contains the following information:

- A psk identity to uniquely identify the pre-shared key. The client sends the psk identity within the `ClientKeyExchange` message to tell the server which pre-shared key shall be used for authenticating a TLS session.
- The pre-shared key to be used with the psk identity.
- A psk identity hint (optional) which may be sent by the server with the `ServerKeyExchange` message to help the client to select the right pre-shared key.
- The remote peer id (optional) of communication partner.

Finally completing our configuration example we have to tell iSaSiLk to use our `PSKCredential` by passing it to the `SSL(Client)Context` object:

```
// create SSLClientContext
SSLClientContext clientContext = new SSLClientContext();
// enable psk cipher suites
CipherSuite[] suites = CipherSuite.CS_ALL_PSK;
clientContext.setEnabledCipherSuites(suites);
clientContext.updateCipherSuites();
// PSK identity as String
String identityStr = "pskclient.iaik.tugraz.at";
// the pre-shared key
PreSharedKey psk = ...;
// create PSKCredential
PSKCredential pskCredential = new PSKCredential(identityStr, psk);
// activate psk credential
clientContext.setPSKCredential(pskCredential);
// the server to which to connect to
String serverName = "pskserver.iaik.tugraz.at";
int serverPort = 443;
// create SSLSocket to connect to the server
SSLSocket socket = new SSLSocket(serverName, serverPort,
clientContext);
// proceed as usual
...
```

Listing 5-10: Configuring an SSLClientContext for using a pre-shared keys

Please note that in this example we do not specify a psk identity hint or a remote peer id for our `PSKCredential`. We do not want to support psk identities and we do not need to search for a pre-shared key based on a remote peer id. We already know that we want to use the pre-shared key of this credential for the following TLS session with server "pskserver.iaik.tugraz.at"; thus we use method setPSKCredential for setting it as the one and only pre-shared key of our `SSLClientContext` object.

On the server side, however, we usually may have to configure one `SSLServerContext` for one `SSLServerSocket` to communicate with a possibly large number of clients. At configuration time we do not know which client(s) may connect to the server during its operation interval. Thus, for each client for which we want to support a PSK based TLS communication, we have to tell iSaSiLk the corresponding pre-shared key in advance. For this purpose we use method `addPSKCredential` of the `SSL(Server)Context` object:

```
// create a SSLServerContext to configure the server
SSLServerContext serverContext = new SSLServerContext();
// enable psk cipher suites
CipherSuite[] suites = CipherSuite.CS_ALL_PSK;
serverContext.setEnabledCipherSuites(suites);
serverContext.updateCipherSuites();
// PSK identity as String
String identityStr = "pskclient.iaik.tugraz.at";
// the pre-shared key
PreSharedKey psk = ...;
// create PSKCredential
PSKCredential pskCredential = new PSKCredential(identityStr, psk);
// activate psk credential
serverContext.addPSKCredential(pskCredential);
...
// add any further psk credentials if required
...
// create SSLServerSocket to listen for connections
SSLServerSocket socket = new SSLSocket(443, serverContext);
// proceed as usual
...
```

Listing 5-11: Configuring an SSLServerContext for using a pre-shared keys

Now both client and server are configured to use a pre-shared key with the client name as identity. In its `ClientKeyExchange` message the client sends the identity ("pskclient.iaik.tugraz.at") to the server who then knows which pre-shared key to be used for the TLS session.

As you see, on the server side our `PSKCredential` does not contain a PSK identity hint or remote peer id, too. Both are only required if the server wants to support psk identity hints. In this case he would need to know the remote peer id of the client to send him a hint for key the client shall use:

```
// PSK identity as String
String identityStr = "pskclient.iaik.tugraz.at";
// the pre-shared key
byte[] psk = ...;
// create PSKCredential
PSKCredential pskCredential = new PSKCredential(identityStr, psk);
// psk identity hint
String identityHintStr = identityStr;
pskCredential.setIdentityHint(identityHintStr);
// remote peer id of the client
pskCredential.setRemotePeerId("pskclient.iaik.tugraz.at");
// activate psk credential
serverContext.addPSKCredential(pskCredential);
// configure ServerContext to send psk identity hints
serverContext.setSendPSKIdentityHint(true);
```

Listing 5-12: Using PSK identity hints requires to know the remote peer id of the client

The last line tells the iSaSiLk server that he shall send a psk identity hint within the `ServerKeyExchange` message.

Since by default – as recommended by the TLS [PSK] specification – psk identity hints are ignored, an iSaSiLk client has to be explicitly configured to recognize a psk identity hint sent by the server:

```
...
// psk identity hint
String identityHintStr = identityStr;
pskCredential.setIdentityHint(identityHintStr);
// remote peer id of the client
pskCredential.setRemotePeerId("pskserver.iaik.tugraz.at");
// activate psk credential
clientContext.addPSKCredential(pskCredential);
// configure ClientContext to recognize an identity hint
clientContext.setIgnorePSKIdentityHint(false);
...
```

Listing 5-13: Configuring a client to recognize psk identity hints sent by the server

When the server now sends a psk identity hint the client searches for a matching psk credential and the handshake can continue (note, that in our example we have used the same id for identity and identity hint).

As you might have noticed the last client example uses method `addPSKCredential` instead of `setPSKCredential` to pass the psk credential to the `SSLClientContext` object. This is quite possible, however, please not the difference:

Method `setPSKCredential` associates a `PSKCredential` (and its pre-shared key) with one particular `SSLContext`. This means that this `SSLContext` only can be used with this `PSKCredential`. During the handshake, when iSaSiLk asks for a pre-shared key to calculate the premaster secret, the `SSLContext` method `getPSKCredential` will return the `PSKCredential` that has been set by calling method `setPSKCredential`. Be careful when using method `setPSKCredential` on the server side. In this case your server will only have one single pre-shared key to be used with any client that requests a PSK based TLS communication.

Method `addPSKCredential` adds a `PSKCredential` to the internal credential repository. When used on the client side, any `PSKCredential` to be added has to contain a remote peer id to tell iSaSiLk for which server the credential shall be used. When used on the server side, a remote peer id is only required if the server wants to send psk identity hints. In this case he must know which psk credential shall be sent to which client. If psk identitiy hints must not be used, remote peer ids are not required for credentials on the server side because any credential is uniquely identified by the psk identity received from the client.

When calling method `addPSKCredential` the `PSKCredential` internally is forwarded to the so-called `PSKManager` which is responsible for psk credential maintaining. If you want to know more about the iSaSiLk psk credential management you may read the next section. However, if you do not intend to write and plug-in your own `PSKManager`, you already have learned enough to design and run your own psk based TLS client-server application. All you have to do is to create and configure PSKCredentials for the pre-shared keys you have negotiated with the client(s)/server(s) you want to talk with. See chapter 6 for a final client-server example that uses pre-shared key based cipher suites.

If required, you also may override the `SSLContext` method `getPSKCredential`. In this case you may not need `add/setPSKCredential` at all. When iSaSiLk calls method `getPSKCredential` to ask for a particular pre-shared key you may, for instance, pop-up a dialog window to let the user enter the required pre-shared key.

5.3 Pre-shared key management

As we have learned in the last chapter, a `PSKCredential` associates a pre-shared key with identity, (optional) identity hint and (optional) remote peer id. Depending on the number of client or servers we want to be able to hold psk based TLS communications, we may have to maintain a certain number of psk credentials.

PSK credential management is the responsibility of the iSaSiLk `PSKManager`. Generally you can use iSaSiLk for psk based TLS communications without any knowledge of the PSK manager architecture. The PSK manager works silently in background. Anytime you add a `PSKCredential` to an `SSLContext` object, it is forwarded to the internal `PSKManager`. And anytime iSaSiLk needs a pre-shared key for authenticating a TLS session with some particular peer, it asks the internal `PSKManager` for a proper `PSKCredential`.

The `PSKManager` itself is an abstract class of package `iaik.security.ssl`:

```
public abstract class PSKManager;
```

It defines some methods that may have to be overridden by a specific `PSKManager` implementation. For a detailed description of all `PSKManager` methods please see the iSaSiLk Javadoc documentation. In this chapter we only will discuss the two most important methods `addPSKCredential` and `getPSKCredential`.

Method `addPSKCredential` adds a `PSKCredential` to the `PSKManager` repository; method `getPSKCredential` searches the repository for a `PSKCredential` based on its identity, identity hint or remote peer id.

Method `addPSKCredential` expects the `PSKCredential` object to be added as argument:

```
public void addPSKCredential(PSKCredential pskCredential);
```

When adding a `PSKCredential` you already decide about the information that later can be used by iSaSiLk when searching for a `PSKCredential` by calling method `getPSKCredential`. Since a `PSKCredential` must have a psk identity you always can search for psk credentials based on their identities. However, to search for a psk credential based on a psk identity hint or remote peer id the `PSKCredential` you want to add shall contain a psk identity hint and/or a remote peer id, respectively:

```
...
String identity = ...;
// the pre-shared key
PreSharedKey psk = ...;
// create PSKCredential
PSKCredential pskCredential = new PSKCredential(identity, psk);
// psk identity hint
String identityHint = ...;
pskCredential.setIdentityHint(identityHint);
// remote peer id of the client
pskCredential.setRemotePeerId(...);
// activate psk credential
context.addPSKCredential(pskCredential);
...
```

Listing 5-14: Creating and adding a PSKCredential

The last line from Listing 5-14 has the same effect as when calling:

```
context.getPSKManager().addPSKCredential(pskCredential);
```

However, when is it necessary to set an identity hint and/or remote peer for a `PSKCredential`? On the server side a `PSKCredential` only will have to contain a psk identity hint if the server wants to send a psk identity hint to the client (`serverContext.setSendPSKIdentityHint(true)`, see 4.2). In this case the `PSKCredential` must contain a remote peer id, too. The identity hint is sent in the `ServerKeyExchange` message to help the client to select the pre-shared key for the forthcoming TLS session. Before sending the `ServerKeyExchange` message the server must decide which of his pre-shared keys shall be used for the session with the client that has initiated the TLS handshake. Thus the server must search his `PSKCredential` repository based on the remote peer id of the client. If he finds a `PSKCredential` that contains a psk identity hint, he can send the hint to the client. If he does not find a proper `PSKCredential` he may continue the handshake without sending an identity hint, or he may abort the handshake.

If, for instance, client "pskclient.iaik.tugraz.at" has connected to server "pskserver.iaik.tugraz.at" the remote peer id may be the DNS name of the client. If the credential repository of the server does contain a `PSKCredential` with remote peer id "pskclient.iaik.tugraz.at", the server may send the psk identity hint – if included – of this `PSKCredential` to the client.

Please note that the TLS [PSK] specification does not recommend using psk identity hints. For that reason, in general, your server-side PSKCredentials may not have to contain psk identity hint or remote peer id information at all.

Client-side PSKCredentials only must contain psk identity hints if the client allows to select pre-shared keys based on a psk identity hint received from the server (`clientContext.setIgnorePSKIdentityHint(false)`, see Listing 5-12 of chapter 5.2). In this case the `PSKCredential` may not have to contain a remote peer id because it is uniquely identified by the identity hint.

However, in general psk identity hint processing should be disabled. Thus any `PSKCredential` that is added on the client side by calling `clientContext.addPSKCredential` shall contain the remote peer id of the server! In contrast to method `setPSKCredential` (which exclusively sets a particular `PSKCredential` for an `SSLClientContext`), method `addPSKCredential` puts the `PSKCredential` into the `PSKManager` repository.

If the client, for instance, wants to connect to server "pskserver.iaik.tugraz.at", he must know which of his pre-shared keys has to be used for a PSK based TLS session with this server. The client will call his `PSKManager` to ask him for a `PSKCredential` with remote peer id "pskserver.iaik.tugraz.at". The `PSKManager` will return the appropriate `PSKCredential`, if included in his psk database. Now the client can send the psk identity of the `PSKCredential` to the server to indicate which pre-shared key shall be used.

Note the difference: On the server side remote peer ids must only be included in the PSKCredentials if the server wants to send psk identity hints; on the client side remote peer ids shall be included in any case, but must be included if psk identity hint processing is NOT enabled.

Figure 5-1 and Figure 5-2 demonstrate the two different situations.

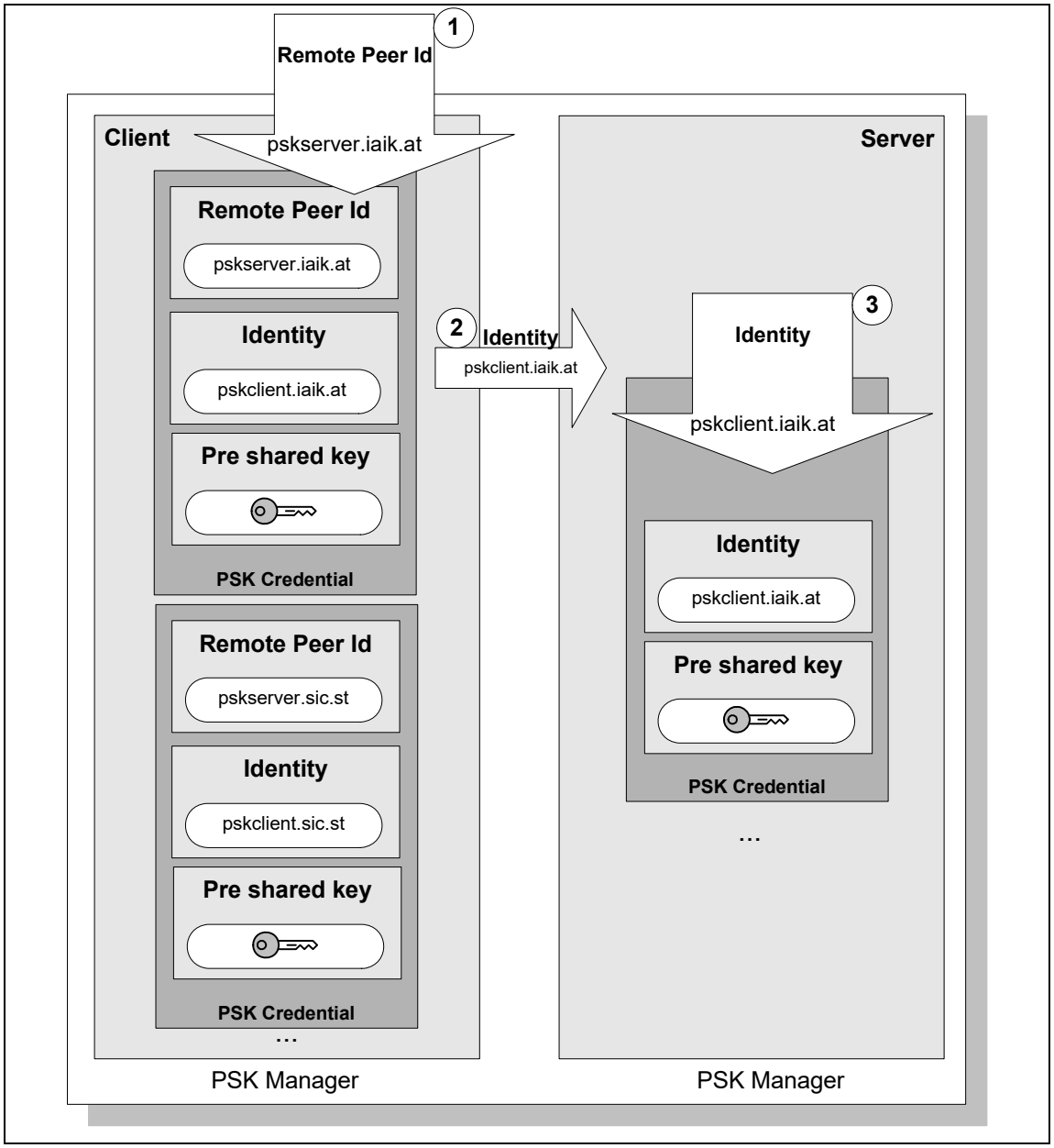

Figure 5-1: PSK management without using identity hints

In scenario a) (Figure 5-1) psk identity hint processing is disabled. The client starts the handshake and asks his `PSKManager` for a `PSKCredential` with the remote peer id of the server "pskserver.iaik.at". In his `ClientKeyExchange` message the client sends the identity of the `PSKCredential` to the server, who uses the identity to search for the proper pre-shared key. On the client side PSKCredentials must contain a remote peer id (since the Client-PSKManager searches based on the remote peer id of the server); on the server side no remote peer ids are required (since the PSKManager searches based on the psk identity received from the client).

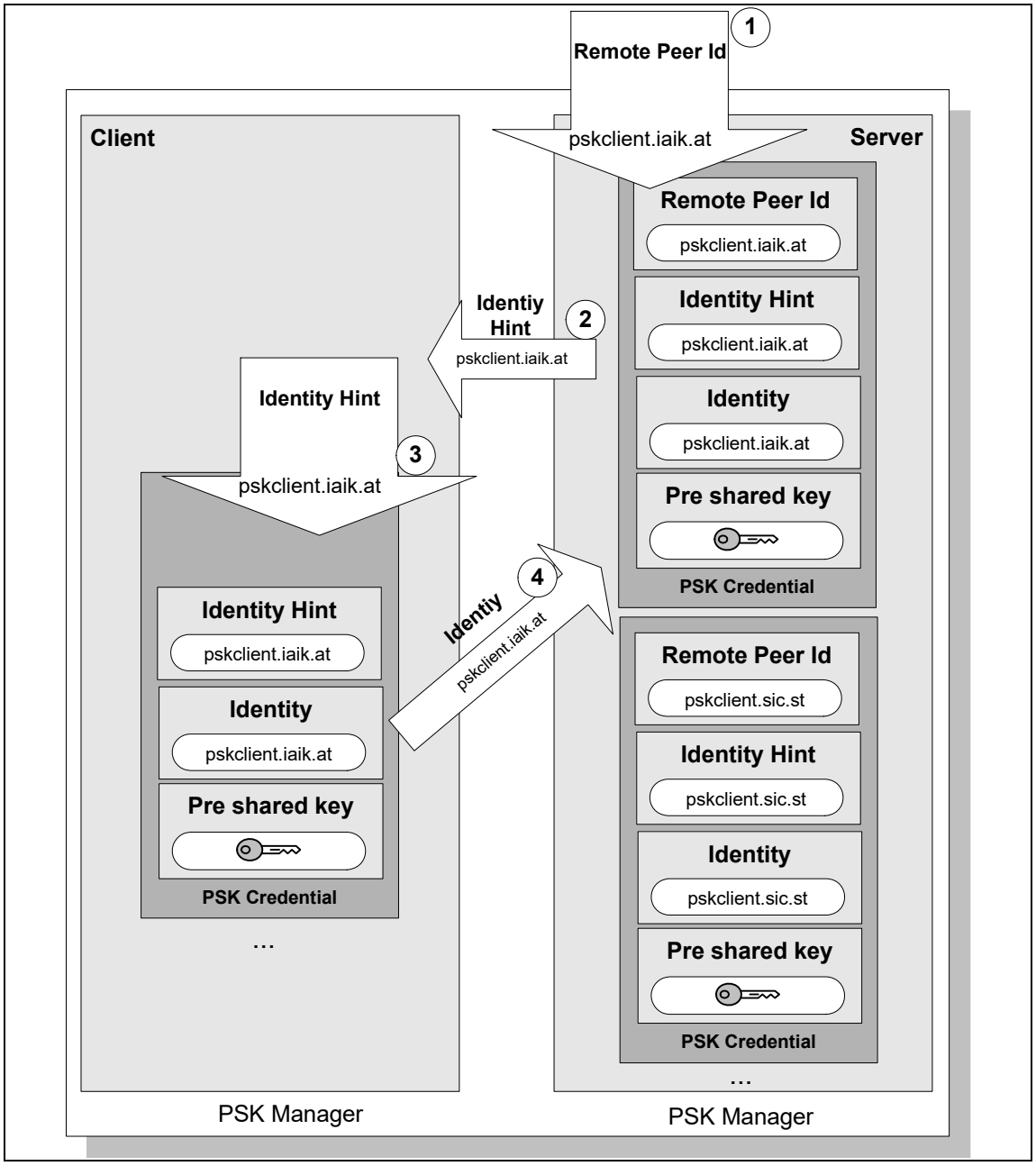

Figure 5-2: PSK management with identity hints

Scenario b) (Figure 5-2) shows the proceeding when psk identity hint processing is enabled. After having received the `ClientHello` message, the server asks his `PSKManager` for a `PSKCredential` with the remote peer id of the client "pskclient.iaik.at". In his `ServerKeyExchange` message the server sends the identity hint to the client, who uses the identity hint to search for a proper `PSKCredential`. In this case the PSKCredentials on the client side may not contain a remote id (since the identity hint of the server is used as search key), but server-side PSKCredentials must contain a remote peer id (since the Server-PSKManager searches based on the remote peer id of the client).

The choice of identity and identity hint values may depend on the specific application environment. In our example identity, identity hint and client peer id all have the same value.

When consulting the `PSKManager` to ask for a proper `PSKCredential`, iSaSiLk calls the `PSKManager` method `getPSKCredential`:

```
public PSKCredential getPSKCredential(byte[] identity,
                                      SSLTransport transport);
```

The first argument (identity) – if not null – represents the psk identity received from the client (server-side), or the psk identity hint received from the server (client-side). In this case the default `PSKManager` ignores the second argument (transport) and searches for a `PSKCredential` based on the given identity bytes. On the server side this behavior represents scenario a) from Figure 5-1(the server has received an identity from the client); on the client side we are in scenario b) as shown in Figure 5-2where we have to search based on the identity hint received from the server. If you write and plug-in your own `PSKManager` implementation your psk management policy may need additional SSLTransport based information (like the remote peer id) also when searching for a `PSKCredential` based on its identity. For instance, you may want to allow that a `PSKCredential` used with one peer may have the same identity as one used with another peer). In such case you may get the required peer identification information from the `SSLTransport` argument.

If the first argument (`identity`) is `null` the search has to be done based on information got from the second argument (`transport`) only. The default `PSKManager` implementation gets the remote peer id from the `SSLTransport` object and searches his database for a proper `PSKCredential`. On the server side we will have to search based on transport information only if we want to send a psk identity hint (scenario b); on the client side we have to use the remote peer id from the `SSLTransport` to search for a `PSKCredential` (scenario a).

When asking the given `SSLTransport` for peer identification information, the default `PSKManager` implementation calls the following `SSLTransport` methods (in that order):

1. `transport.getRemotePeerName();`
2. `transport.getRemoteIndetAddress().getHostAddress();`
3. `transport.getRemotePeerId();`

If you are, for instance, on the client side and want to go into a psk based TLS session with a server "pskserver.iaik.tugraz.at" with ip address "129.27.142.47" listening on port 4433, the SSL(Socket)Transport methods above will return the following values (in that order):

1. pskserver.iaik.tugraz.at
2. 129.27.142.47
3. 129.27.142.47:4433

As mentioned you can write your own `PSKManager` by extending the abstract `PSKManager` class:

```
public class MyPSKManager extends iaik.security.ssl.PSKManager {
   …
}
```

Listing 5-15: Extend class PSKManager to implement your own pre-shared key management

You then can install your `PSKManager` class as new default `PSKManager` to be used, or set it for one particular `SSLContext` only:

```
MyPSKManager myPSKManager = new MyPSKManager();
PSKManager.setDefault(myPSKManager);
```

Listing 5-16: Setting a default PSKManager

or:

```
MyPSKManager myPSKManager = new MyPSKManager();
SSLContext context = …;
context.setPSKManager(myPSKManager);
```

Listing 5-17: Setting a PSKManager for a particular SSLContext

If you install your `PSKManager` as default `PSKManager` it will be used by any further `SSLContext` objects that may be created; if you enable it for one specific `SSLContext` object, it will be used by this `SSLContext` only. Please see the iSaSiLk Javadoc for a detailed description of the `PSKManager` methods you will have to implement for your own `PSKManager` implementation.

In chapter 5.2we have said that a Java™ KeyStore can be used for securely storing pre-shared keys. However, when using a KeyStore you only can associate an alias with the secret (pre-shared) key. This might be quite sufficient if your PSKCredentials only will contain pre-shared keys and identities. In this case you may use the psk identity as alias for the KeyStore entry, e.g.:

```
// create KeyStore instance
KeyStore keyStore = KeyStore.getInstance("IAIKKeyStore");
// load KeyStore from file
InputStream is = ...;
Char[] pwd = ...;
keyStore.load(is, pwd);
// get identity and pre-shared key from PSKCredential
PSKCredential pskCredential = ...;
String identityStr = pskCredential.getIdentityString();
SecretKey psk = pskCredential.getPSK();
// use identity as alias when adding the psk to the KeyStore
keyStore.setKeyEntry(identityStr, psk, pwd, null);
// store KeyStore to file
OutputStream os = ...;
keyStore.store(os, pwd);
```

Listing 5-18: Saving pre-shared keys to a Java™ KeyStore

When loading the KeyStore again you will have to use the identity string as alias to read the pre-shared from the KeyStore. If, for instance, we have used the name of the client "pskclient.iaik.tugraz.at" as identity string we now have to use it as alias for getting the corresponding pre-shared key:

```
// create KeyStore instance
KeyStore keyStore = KeyStore.getInstance("IAIKKeyStore");
// load KeyStore from file
InputStream is = ...;
Char[] pwd = ...;
keyStore.load(is, pwd);
// get the pre-shared key
String alias = "pskclient.iaik.tugraz.at";
SecretKey psk = (SecretKey)keyStore.getKey(alias, pwd);
...
// create PSKCredential for pre-shared key
PSKCredential pskCredential = new PSKCredential(alias, psk);
// add PSKCredential to SSLContext
SSLContext context = ...;
context.addPSKCredential(pskCredential);
...
```

Listing 5-19: Reading pre-shared keys from a Java™ KeyStore

When using a Java™ KeyStore as storage medium, be aware that you only can keep the identity associated with the pre-shared key, but not the remote peer id and/or psk identity hint of a `PSKCredential` (except for when identity, identity hint and remote peer id are all the same). For that reason, the iSaSiLk `DefaultPSKManager` provides `store` and `load` methods allowing to password based encrypted store the full PSKManager contents:

```
// the stream to which to store the PSKManager
OutputStream os = …;
char[] pwd = …;
// get DefaultPSKManager
DefaultPSKManager pskManager =
  (DefaultPSKManager)PSKManager.getDefault();
// store DefaultPSKManager
pskManager.store(os, pwd);
```

Listing 5-20: Password-based storing the contents of the DefaultPSKManager

When loading the (Default)PSKManager again use the same password:

```
// the stream from which to read the PSKManager
InputStream is = …;
char[] pwd = …;
// create and laod DefaultPSKManager
DefaultPSKManager pskManager = new DefaultPSKManager();
pskManager.load(is, pwd);
// enable PSKManager
PSKManager.setDefault(pskManager);
// store DefaultPSKManager
```

Listing 5-21: Reading the contents of a stored DefaultPSKManager

Please note that the `store` and `load` methods are only available for the `DefaultPSKManager`, but must not be provided by any `PSKManager` implementation. The `DefaultPSKManager` uses PKCS#5 `PBKDF2` as key derivation function, `AES` for symmetric content encryption and `HmacSHA256` for content integrity protection.

6 PSK Client/Server example

In this chapter finally we present the source code of a simple TLS client/server example using pre-shared key based cipher suites. We assume that client "pskclient.iaik.tugraz.at" and server "pskserver.iaik.tugraz.at" have negotiated a pre-shared key out of band. The pre-shared key shall be identified by the DNS name of the client "pskclient.iaik.tugraz.at". PSK identity hints shall not be supported. Thus the `PSKCredential` on the client-side has to contain the remote peer id of the server ("pskserver.iaik.tu.graz"), whereas the server does not need to know the peer id of the client because he uses the psk identity sent by the client to search for the right `PSKCredential`.

6.1 PSK Client

The sample client uses a pre-shared key with identity "pskclient.iaik.tugraz.at" to connect to server "pskserver.iaik.tugraz.at" listening for TLS connections on port 4433. In this example we use PSK cipher suites with symmetric key based authentication only.

```
import iaik.security.provider.IAIK;
import iaik.security.ssl.CipherSuite;
import iaik.security.ssl.CipherSuiteList;
import iaik.security.ssl.PSKCredential;
import iaik.security.ssl.SSLClientContext;
import iaik.security.ssl.SSLSocket;
import iaik.security.ssl.Utils;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * Sample PSKClient using a pre-shared key with identity
 * "pskclient.iaik.tugraz.at" to held a psk cipher suite based
 * TLS session with server "pskserver.iaik.tugraz.at".
 */
public class PSKClient {

  /**
   * Default constructor.
   * Creates a PSKClient object.
   */
  public PSKClient() {
  }
```

```
/**
 * Connects to the given server at the given port.
 *
 * @param serverName the server name
 * @param serverPort the port the server is listening for connections
 * @param context the SSLContext with the TLS client configuration
 */
public void connect(String serverName,
                    int serverPort,
                    SSLClientContext context) {

  SSLSocket socket = null;
  PrintWriter writer = null;
  BufferedReader reader = null;

  try {

    System.out.println("Connect to " + serverName + " on port " +
                       serverPort);
    // create SSLSocket
    socket = new SSLSocket(serverName, serverPort, context);
    // print debug info to System.out
    socket.setDebugStream(System.out);
    // start handshake
    socket.startHandshake();
    System.out.println();

    // informationen about the server:
    System.out.println("TLS-Connection established.");
    System.out.println("Session-Parameters:");
    System.out.println("Active cipher suite: " +
                       socket.getActiveCipherSuite());
    System.out.println("Active compression method: " +
                       socket.getActiveCompressionMethod());
    // dump psk identity
    String pskIdentity = socket.getPSKIdentity();
```

```
    if (pskIdentity != null) {
      System.out.println("PSK Identity: " + pskIdentity);
    }
    System.out.println();

    // send GET-request
    System.out.println("Sending HTTPS request to " + serverName);
    writer = Utils.getASCIIWriter(socket.getOutputStream());
    reader = Utils.getASCIIReader(socket.getInputStream());
    writer.println("GET / HTTP/1.0");
    writer.println();
    writer.flush();

    // read response
    System.out.println("Reading response...");
    while( true ) {
      String line = reader.readLine();
      if( line == null ) {
        break;
      }
      System.out.print(":");
      System.out.println(line);
    }

  } catch( IOException e ) {
    System.err.println("IOException:");
    e.printStackTrace(System.err);
  } finally {
    if (writer != null) {
      writer.close();
    }
    if (reader != null) {
      try {
        reader.close();
      } catch (IOException ex) {
        // ignore
      }
    }
```

```
      if (socket != null) {
        try {
          socket.close();
        } catch (IOException ex) {
          // ignore
        }
      }
    }
  }

  /**
   * Main method. Starts the client, connects to the server,
   * sends a HTTP GET request and reads the response.
   */
  public static void main(String arg[]) throws IOException {

    // the server we want to connect to
    String serverName = "pskserver.iaik.tugraz.at";
    int serverPort = 4433;

    // add the IAIK crypto provider
    IAIK.addAsProvider(true);

    // client context
    SSLClientContext context = new SSLClientContext();

    // the pre-shared key negotiated with the server out-of-band
    PreSharedKey psk = …;
    // create PSKCredential with identity "pskclient.iaik.tugraz.at"
    String identity = "pskclient.iaik.tugraz.at";
    PSKCredential credential = new PSKCredential(psk, identity);
    // set remote peer id of server
    credential.setRemotePeerId(serverName);
    // enable psk credential pre-shared key
    context.addPSKCredential(credential);

    // enable PSK cipher suites (in this sample we want use
    // only symmetric key operations for authentication)
```

```
    CipherSuiteList suites = new CipherSuiteList();
    suites.add(CipherSuite.CS_PSK);
    context.setEnabledCipherSuiteList(suites);
    context.updateCipherSuites();

    // dump context
    System.out.println("Context:\n" + context);
    System.out.println();

    // create PSKClient
    PSKClient client = new PSKClient();
    // connect to server
    client.connect(serverName, serverPort, context);
  }
}
```

6.2 PSK Server

The sample server listens on port 4433 and uses a pre-shared key with identity "pskclient.iaik.tugraz.at" to communicate with the client presented in the previous section. To be able to simultaneously talk with any number of clients, each client request is handled by a separate Thread. In this example we use PSK cipher suites with symmetric key based authentication only.

```
import iaik.security.provider.IAIK;
import iaik.security.ssl.CipherSuite;
import iaik.security.ssl.CipherSuiteList;
import iaik.security.ssl.PSKCredential;
import iaik.security.ssl.PreSharedKey;
import iaik.security.ssl.SSLServerContext;
import iaik.security.ssl.SSLServerSocket;
import iaik.security.ssl.SSLSocket;
import java.io.IOException;

/**
 * Sample PSKServer using a pre-shared key with identity
 * "pskclient.iaik.tugraz.at" to held a psk cipher suite based
 * TLS session with client "pskclient.iaik.tugraz.at".
 */
public class PSKServer {
  // port number to listen on
  static int port_ = 4433;
  // server context
  private SSLServerContext serverContext_;

  /**
   * Creates a new PSK server.
   *
   * @param serverContext the configured server context
   */
  public PSKServer(SSLServerContext serverContext) {
    serverContext_ = serverContext;
  }
```

```
/**
 * Starts the PSK Server.
 */
public void start() {

  // create SSLServerSocket
  SSLServerSocket serverSocket;
  try {
    serverSocket = new SSLServerSocket(port_, serverContext_);
  } catch( IOException e ) {
    System.err.println("Error binding to port " + port + ":");
    e.printStackTrace();
    return;
  }
  System.out.println("Listening for HTTPS connections on port " +
                     port + "...");

  // for each request create a new Thread
  while (true) {
    try {
      SSLSocket socket = (SSLSocket)serverSocket.accept();
      (new PSKServerThread(socket)).start();
    } catch( IOException e ) {
      e.printStackTrace(System.out);
    }
  }
}

/**
 * Main method. Configures and runs the server.
 */
public static void main(String args[]) throws IOException {

  // add IAIK-JCE crypto provider
  IAIK.addAsProvider(true);
```

```
    // the server context
    SSLServerContext serverContext = new SSLServerContext();

    // the pre-shared key negotiated with the server out-of-band
    PreSharedKey psk = null;
    // create PSKCredential with identity "pskclient.iaik.tugraz.at"
    String identity = "pskclient.iaik.tugraz.at";
    PSKCredential credential = new PSKCredential(identity, psk);
    // enable psk credential
    serverContext.addPSKCredential(pskCredential);

    // enable PSK cipher suites (in this sample we want use
    // only symmetric key operations for authentication)
    CipherSuiteList suites = new CipherSuiteList();
    suites.add(CipherSuite.CS_PSK);
    serverContext.setEnabledCipherSuiteList(suites);
    serverContext.updateCipherSuites();

    // display configuration
    System.out.println("ServerContext:\n" + serverContext);

    // create and run the server
    PSKServer sslServer = new PSKServer(serverContext);
    sslServer.start();

  }
}
```

Class `PSKServerThread` is responsible for handling each client request:

```
import iaik.security.ssl.CipherSuite;
import iaik.security.ssl.SSLContext;
import iaik.security.ssl.SSLOutputStream;
import iaik.security.ssl.SSLSocket;
import iaik.security.ssl.Utils;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.Enumeration;
import java.util.Vector;

/**
 * This class implements the server-side thread for handling a client
 * request. Each client request is handled by a separate thread which
 * sends back a HTML-page that simply dumps the client request.
 */
public class PSKServerThread extends Thread {

  // the Socket for communicating with the client
  private SSLSocket socket_;

 /**
  * Creates a new PSKServerThread.
  *
  * @param socket the socket to be used for communicating with the
  *               client
  */
  public PSKServerThread(SSLSocket socket) {
     super("PSKServerThread");
     socket_ = socket;
  }

  /**
```

```
 * Handles a client request.
 */
public void run() {
  try {
    System.out.println("Accepted connection from " +
                       socket_.getInetAddress());
    // print debug-information to System.out:
    socket_.setDebugStream(System.out);
    socket_.setSoTimeout(1000*30);
    OutputStream os = socket_.getOutputStream();
    ((SSLOutputStream)os).setAutoFlush(false);
    InputStream is = socket_.getInputStream();
    BufferedReader reader = Utils.getASCIIReader(is);
    PrintWriter writer = Utils.getASCIIWriter(os);
    // read client request:
    Vector request = new Vector();
    System.out.println("Client Request:");
    while( true ) {
      String line = reader.readLine();
      if( (line == null) || (line.length() == 0) ) {
        if (!reader.ready())
          break;
      }
      System.out.println(line);
      request.addElement(line);
    }

    // send response:
    System.out.println("Sending reply...");
    writer.println("HTTP/1.0 200 OK");
    writer.println("Content-Type: text/html");
    writer.println("Server: IAIK-SSL Demoserver");
    writer.println("Pragma: no-cache");
    writer.println("Cache-control: no-cache");
    writer.println();
    // create HTML page:
    writer.println("<HTML><HEAD><TITLE>SSLTest</TITLE></HEAD>");
    writer.println("<BODY><H1>TLS PSK Test ok.</H1>");
```

```
      CipherSuite suite = socket_.getActiveCipherSuite();
      writer.println("Active ciphersuite: <CODE>" + suite +
                     "</CODE><P>");
      System.out.println("Active ciphersuite: " + suite);
      String version =
        Utils.getVersionString(socket_.getActiveProtocolVersion());
      writer.println("Active protocol version: " + version);
      System.out.println("Active protocol version: " + version);
      String pskIdentity = socket_.getPSKIdentity();
      if (pskIdentity != null) {
        writer.println("<P>PSK Identity: <CODE>" + pskIdentity +
                       "</CODE><P>");
        System.out.println("PSK Identity: " + pskIdentity);
      }
      writer.println("<P>The request sent by your client was: " +
                     "<BLOCKQUOTE><PRE>");
      for (Enumeration e = request.elements(); e.hasMoreElements(); ) {
        writer.println(e.nextElement());
      }
      writer.println("</PRE></BLOCKQUOTE><HR>Generated by " +
                     "<A HREF=\"http://jce.iaik.tugraz.at/\">iSaSiLk ");
      writer.println(SSLContext.LIBRARY_VERSION_STRING +
                     "</A>.</BODY></HTML>");
      writer.flush();
      writer.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    } finally {
      if (socket_ != null) {
        try {
          socket_.close();
        } catch (IOException e) {
           // ignore
        }
      }
    }
  }
}
```

7 Acronyms

AES	Advanced Encryption Standard; Symmetric block cipher, designed by Joan Daemen and Vincent Rijmen; NIST standardized successor of the DES (Data Encryption Standard) cipher
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)
HmacSHA256	Keyed-Hashing for Message Authentication as described in RFC 2104 using SHA-256 as message digest algorithm
PBKDF2	Password Based Key Derivation Function 2, specified in PKCS#5v2.0
PKCS#5	Password Based Encryption Standard (Public Key Cryptography Standard No. 5, by RSA Data Security Inc.)
PSK	Pre-Shared Key (Symmetric key shared among two parties)
RSA	Public-key algorithm, developed by Ron Rivest, Adi Shamir and Leonard Adleman; may be used for data encryption or digital signing.
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

8 References

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

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

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

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

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

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

[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.

[PSK] P. Eronen, H. Tschofenig: "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS) ", RFC 4279, December 2005

[PSK-NULL] U. Blumenthal, P. Goel: "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS) ", RFC 4785, January 2007.

[UTF8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.