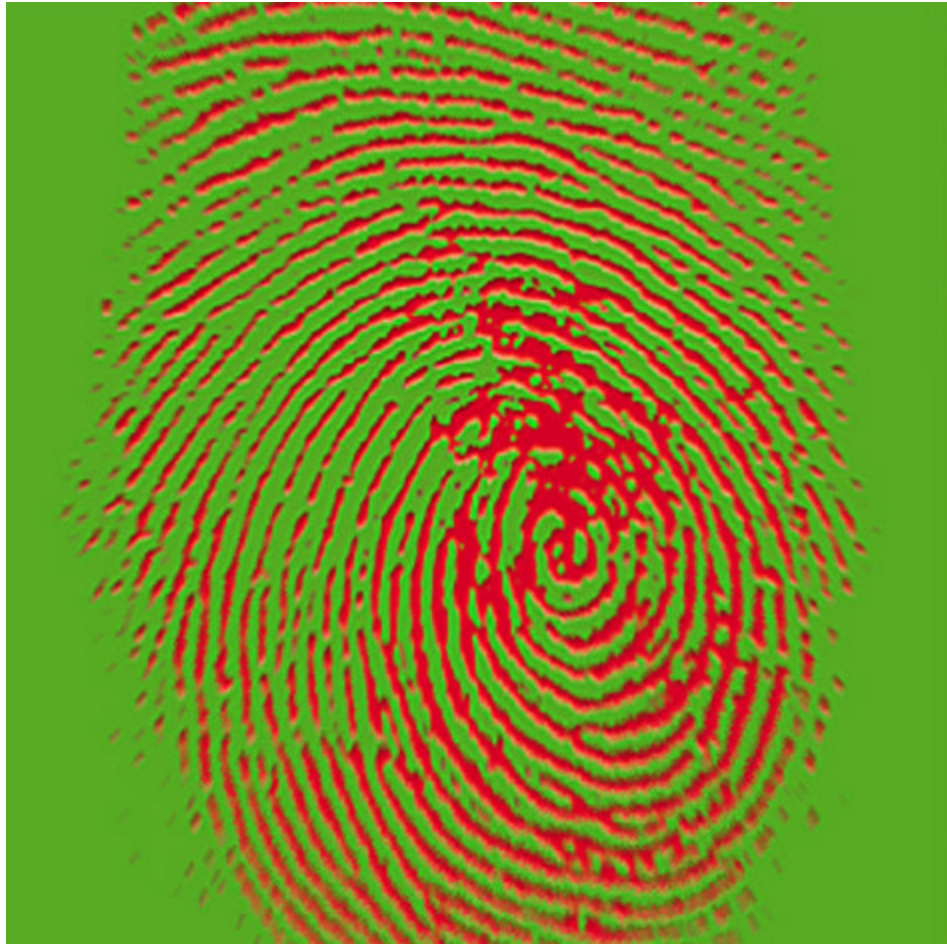


FORGE

PROTEKT Encryption

Full-Strength Encryption for Java



Protekt Encryption 3.0

Programmers Guide

This publication is copyright and contains information that is the property of Forge Research Pty Ltd. It is not to be reproduced in any form, supplied to any other party or organisation, applied for any purpose or to any contract without the prior written permission of Forge Research Pty Ltd. Information in this document is subject to change without notice.

Copyright © 1997 - 2000 Forge Research Pty Ltd ACN 003 491 576.
All rights reserved.

Java is a trademark of Sun Microsystems, Inc.
The RSA public-key cryptosystem is protected by U.S. Patent 4,405,829.
RSA and RC4 are registered trademarks of RSA Security Inc.
IDEA is a registered trademark of Ascom Systec.

Third-Party Trademarks: All other trademarks, trade names or company names referenced herein are used for identification purposes only and are the property of their respective owners.

Contact information:

Forge Research Pty Ltd
Suite 116, Bay 9 The Locomotive Workshop
Australian Technology Park
NSW, 1430
Australia

PO Box 598 Alexandria
NSW, 1435
Australia

Phone: +61 2 9209 4152

Fax: +61 2 9209 4172

Technical Support: protekt@forge.com.au

Protekt Sales: sales@forge.com.au

World Wide Web: www.forge.com.au
www.protekt.com

CONTENTS

1	PREFACE.....	1
1.1	AUDIENCE.....	1
1.2	ORGANISATION OF THIS GUIDE.....	1
1.3	DOCUMENT CONVENTIONS.....	2
2	AN INTRODUCTION TO TLS.....	3
3	GETTING STARTED WITH PROTEKT.....	5
3.1	THE SERVER.....	5
3.2	THE CLIENT.....	7
3.3	WORKING WITH CIPHER SUITES.....	9
3.3.1	<i>Supported cipher suites.....</i>	<i>9</i>
3.3.2	<i>Specifying which cipher suites can be negotiated for a connection.....</i>	<i>9</i>
3.4	USING A SPECIFIC CERTIFICATE CHAIN.....	10
3.5	USING CLIENT AUTHENTICATION.....	10
3.6	SPECIFYING A HANDSHAKE TIMEOUT.....	11
3.7	REUSING SESSIONS.....	12
3.8	USING PROTEKT IN AN APPLET.....	13
3.8.1	<i>The <code>Protekt.properties</code> file.....</i>	<i>13</i>
3.8.2	<i>The keystore.....</i>	<i>13</i>
3.8.3	<i>Including JCE Providers in the jar file.....</i>	<i>14</i>
4	AUTHENTICATION.....	15
4.1	PUBLIC KEY CRYPTOGRAPHY.....	15
4.2	CERTIFICATE CHAINS AND TRUSTED CERTIFICATES.....	15
4.3	THE JAVA KEYSTORE.....	16
4.3.1	<i>Specifying the keystore to use.....</i>	<i>17</i>
4.3.2	<i>Specifying the server private key/certificate chain.....</i>	<i>18</i>
4.3.3	<i>Specifying the client private keys/certificate chains.....</i>	<i>18</i>
5	THE PROTEKT.PROPERTIES FILE.....	20
5.1	SPECIFYING THE LOCATION OF THE PROTEKT.PROPERTIES FILE.....	20
5.2	JCE PROVIDER PROPERTIES.....	20
5.2.1	<i>Services used by Protekt.....</i>	<i>20</i>
5.2.2	<i>The providers to be used.....</i>	<i>21</i>
5.2.3	<i>The random number generator.....</i>	<i>22</i>
5.2.4	<i>RSA cipher initialisation.....</i>	<i>22</i>
5.3	CIPHER SUITES.....	23
5.4	KEYSTORE PROPERTIES.....	24
5.5	ADDITIONAL PROPERTIES.....	24
6	APPENDIX A - GLOSSARY.....	25
7	APPENDIX B – TROUBLESHOOTING.....	27
7.1	KEYSTORE PROBLEMS.....	27
7.1.1	<i>Could not load keystore from <filename>.....</i>	<i>27</i>
7.1.2	<i>Could not load specified certificate chain (<alias> not found in the keystore).....</i>	<i>27</i>
7.1.3	<i>Could not load specified certificate chain (<alias> is a trusted certificate entry, not a key entry).....</i>	<i>27</i>
7.2	JCE PROBLEMS.....	27
7.2.1	<i>No Cipher could be found for algorithm <algorithm>.....</i>	<i>27</i>
8	APPENDIX C – IMPORTING A CERTIFICATE SIGNED BY A CA.....	28
8.1	MAKING JCE PROVIDERS AVAILABLE TO THE KEYTOOL.....	28
8.2	GENERATING A KEY PAIR.....	28
8.3	CREATING A CERTIFICATE SIGNING REQUEST.....	29
8.4	IMPORTING THE CA ROOT CERTIFICATE (OPTIONAL).....	29

8.5	IMPORTING THE CERTIFICATE RETURNED FROM THE CA	30
9	APPENDIX D – EXAMPLE PROTEKT.PROPERTIES FILE.....	31

1 Preface

Protekt is an all Java implementation of version 3 of the SSL protocol and version 1 of the TLS protocol. Unless otherwise specified, when this manual mentions TLS, it should be read SSL or TLS.

Protekt provides high strength encryption over TCP/IP connections between any platforms that have Java 2 support.

Features of Protekt include:

- No native code – Protekt will run anywhere there is a Java 2 *VM*.
- Cryptography service provider independent – Javasoft's Java Cryptographic Architecture (*JCA*) allows Protekt to use cryptography services provided by any *JCE* (Java Cryptographic Extension) compliant provider.
- Standard Java key and certificate management using Java *keystores*.
- Session caching and reuse, giving substantial performance benefits.
- Renegotiation of secret keys throughout a TLS conversation enhancing security.

1.1 Audience

Programmers wishing to add secure communications to Java applications should read this guide.

This guide assumes readers are familiar with TCP/IP networking and programming networking applications using the `java.net` package.

A brief introduction to how the TLS protocol is also given.

1.2 Organisation of this Guide

This guide is divided into chapters as follows:

- An introduction to TLS

This chapter gives an introduction to the TLS protocol, including the handshake mechanism used to negotiate the encryption parameters for the connection.

- Getting started with Protekt

This chapter presents sample client and server programs, showing how to use the Protekt API.

- Authentication

This chapter explains how public key cryptography can be used for authentication purposes, introduces the Java keystore and shows how Protekt performs authentication when acting as a TLS client or server.

- The Protekt properties file

This chapter documents the settings in the Protekt properties file.

Four appendices are provided – a glossary, a trouble-shooting guide, details of generating key pairs and importing certificates received from Certificate Authorities and an example `Protekt.properties` file.

1.3 Document Conventions

This document uses the following typographical conventions:

<code>Constant width</code>	Constant width words or characters represent source code or literal values such as commands.
<i>Italics</i>	Italics are used to highlight technical terms or terms that have specific meanings within the context of TLS that appear in the glossary the first time they are used in the guide. Italics are also used when referring to another chapter or section of the guide.
...	Horizontal or vertical ellipses indicate material has been trimmed for clarity.

2 An introduction to TLS

TLS (Transport Layer Security¹) is a protocol that operates over TCP/IP connections providing secure communications over insecure networks. TLS is an IETF (Internet Engineering Task Force) standard that is based on an earlier protocol called SSL (Secure Sockets Layer²). SSL was originally designed by Netscape Communications Corporation.

TLS allows two parties (a client and a server) to agree on a set of cryptographic parameters to protect any data sent over a TCP/IP connection. Optionally, the parties can *authenticate* each other using *X.509 certificates* as the identification information.

When a TLS connection is opened the first step is called the TLS handshake. The handshake is a series of messages exchanged between the two parties where the identity of each party is established and the parameters for the data encryption are generated. The server usually identifies itself to the client. The server can also request the client to identify itself.

For an explanation of authentication using certificates, see chapter 4, *Authentication*.

The TLS handshake starts with the client sending the server a “client hello” message. The “client hello” message contains information such as the highest version of the TLS protocol the client can use and which *cipher suites* (encryption algorithms) the client can use.

When the server receives a “client hello” message it checks the protocol version being used by the client and looks for the session specified by the client to see if existing cryptographic parameters can be used. If the requested session cannot be found, the server selects one of the encryption algorithms presented by the client as the one to be used for this connection. The encryption algorithm selected will typically be the strongest encryption that both the client and server can use. The order in which encryption algorithms will be selected by Protekt is set in the properties file (see chapter 5, *The Protekt.properties file*).

The server responds with a “server hello” message containing the version of the TLS protocol to be used for the connection and the encryption algorithm to be used to protect data sent over the connection.

Following the “server hello” message, the server will send a “certificate” message containing the *certificate chain* it is using to identify itself to the client.

The server may also send a “certificate request” message to the client, if the application requires client authentication. If this is sent, the server requires the client to identify itself using a certificate chain, exactly as the server has done.

¹ RFC2246, <http://www.ietf.org>

² draft-freier-ssl-version3-02, <http://www.netscape.com/eng/ssl3>

Finally the server sends a “server hello done” message. This tells the client that the server has completed its part of the handshake.

When the client receives the “server hello done” message it responds with a certificate chain (if the server sent a certificate request message), a “client key exchange message” (always) and a “certificate verification” message if it sent a “certificate message” to the server.

The “client key exchange” message contains a random value generated by the client. This random value is used by the client and the server to generate the values used to encrypt the data for the rest of the connection. This random value is encrypted with the server’s *public key* (see chapter 4, *Authentication*, for an explanation of public key cryptography) to protect it while in transit.

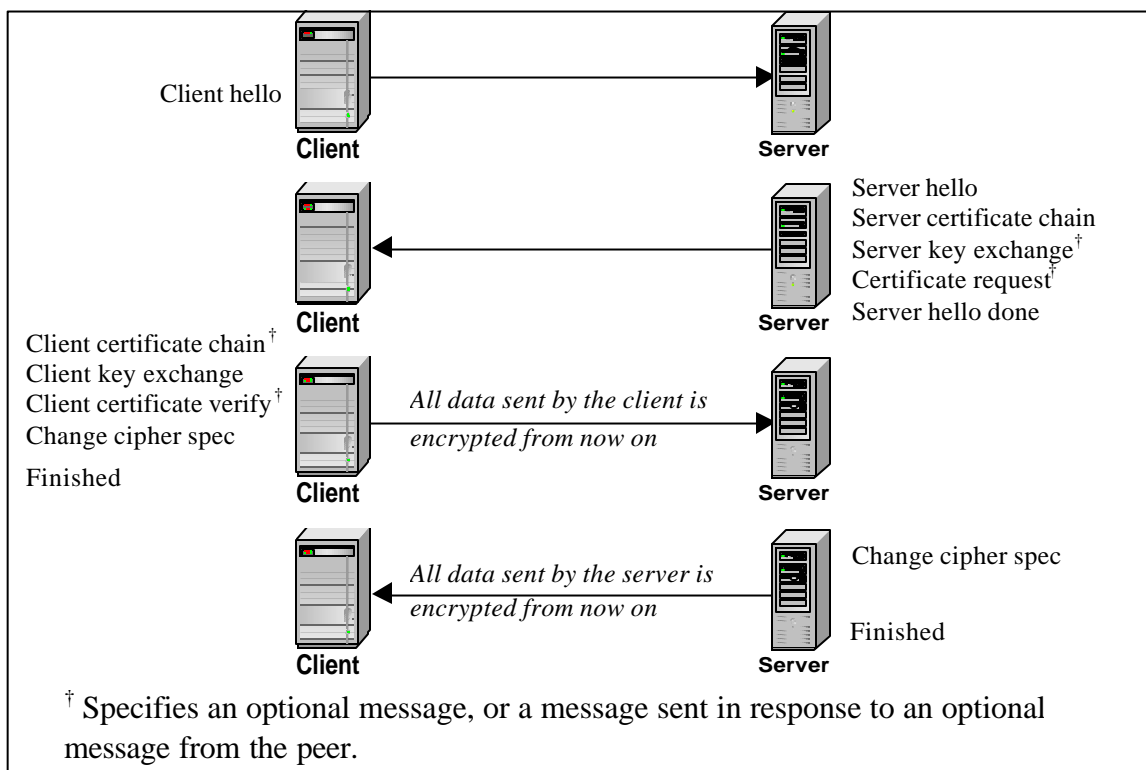
After these messages have been sent, the client sends a “change cipher” message, which tells the server that all data it sends in the future will be encrypted using the just negotiated cipher suite.

Finally the client sends an encrypted “finished” message, using the selected cipher suite which allows the server to verify it has generated the correct values from the “client key exchange” message.

After the server has received and processed these messages it sends a “change cipher” message and “finished” message to the client so the client can also verify the cryptographic settings.

From this point on, all communications over the connection are encrypted using the keys agreed upon earlier.

The handshake message exchange is shown below:



3 Getting started with Protekt

This chapter presents the example client and server programs included in the Protekt distribution.

The source code for the client and server programs presented below has the Protekt specific changes highlighted with a grey background. The following code is only for example purposes so error handling and robustness are not of a standard suitable for production quality code.

3.1 The Server

The example server is a typical threaded Java network server. It uses one thread to accept incoming requests and one thread per request to service it.

The code is shown below, with the lines changed to support TLS highlighted. An explanation for each line is below the code.

```
1 package au.com.forge.security.tls.test;
2
3 import java.io.*;
4 import java.net.*;
5 import au.com.forge.security.tls.*;
6
7 /**
8  * An example multi-threaded server using Protekt
9  * TLS/SSL sockets.<p>
10 *
11 * Copyright © 1997 - 2000 Forge Research Pty Ltd ACN 003 491 576<p>
12 */
13 public class ExampleServer extends Thread {
14
15     /*
16     * Default port number to listen on
17     */
18     private static int port = 4433;
19
20     /**
21     * The main method for the example server.<p>
22     *
23     * Creates and starts the thread that accepts connections
24     * from a Protekt server socket.<p>
25     *
26     * @param args command line arguments.
27     */
28     public static void main(String[] args) {
29         ExampleServer ts = new ExampleServer();
30         ts.start();
31     }
32
33     /**
34     * Accept connections from the Protekt server socket.<p>
35     */
36     public void run() {
37         try {
38             SSLEnvironment.initialise(true);
39             SSLServerSocket ss = new SSLServerSocket(port);
40
41             while (true) {
42                 SSLSocket s = (SSLSocket) ss.accept();
```

```

43         ExampleServerThread est = new ExampleServerThread(s);
44
45         est.setDaemon(true);
46         est.start();
47     }
48 }
49 catch (SSLException ssle) {
50     ssle.printStackTrace();
51 }
52 catch (IOException ioe) {
53     ioe.printStackTrace();
54 }
55 }
56
57 /**
58  * Each connection will be passed into here for
59  * further processing.<p>
60  *
61  * This is an inner class of testServer.<p>
62  */
63 class ExampleServerThread extends Thread {
64     private SSLSocket s;
65
66     /**
67      * Creates a new thread to handle a
68      * connection.<p>
69      *
70      * @param s the TLS/SSL socket accepted
71      * by the server.
72      */
73     public ExampleServerThread(SSLSocket s) {
74         super();
75         this.s = s;
76     }
77
78     /**
79      * Handle a request.<p>
80      */
81     public void run() {
82         try {
83             s.startHandshake();
84             s.waitForHandshake();
85
86             InputStream is = s.getInputStream();
87             InputStreamReader isr = new InputStreamReader(is);
88             BufferedReader reader = new BufferedReader(isr);
89             OutputStream os = s.getOutputStream();
90             String inputLine;
91             boolean foundGetRequest = false;
92
93             while (true) {
94                 inputLine = reader.readLine();
95
96                 if (inputLine == null || inputLine.length() == 0)
97                     break;
98
99                 if (inputLine.startsWith("GET "))
100                     foundGetRequest = true;
101             }
102
103             String HTMLString;
104
105             if (foundGetRequest == true)
106                 HTMLString = new String("Lots of HTML code");
107             else
108                 HTMLString = new String("400 Bad Request");
109
110             os.write(HTMLString.getBytes());

```

```

111         os.flush();
112         os.close();
113     }
114     catch (Exception e) {
115         e.printStackTrace();
116     }
117 }
118 }
119 }

```

Line 5 imports the `Protekt` classes so they can be used in the program.

Line 38 initialises `Protekt` by loading the keys and certificates and starting the random number generator. This is optional; the first time a connection is created or accepted this will be done. The parameter (`true`) tells `Protekt` it can take on the role of server in an SSL handshake.

Line 39 creates the `Protekt SSLServerSocket`. Except for the different class name, this is exactly the same as creating a standard `ServerSocket`.

Line 42 accepts a connection from the server socket and casts it to a `Protekt SSLSocket`. The accepted socket needs to be cast to a `SSLSocket` so the handshake methods can be called.

Lines 49-51 catch the `SSLException` that can be thrown by the `SSLEnvironment.initialise` call.

Line 64 declares the `SSLSocket` that the `ExampleServerThread` will work with.

Line 73 takes a `SSLSocket` as a parameter, rather than a standard `Socket`.

Lines 83 and 84 perform the SSL handshake that determines the cryptographic parameters used to protect the application data. This must be done before the Input and Output streams are accessed.

3.2 The client

As with the server, very little code needs to be added to use `Protekt` in the client.

Again, the code is shown below, with the lines changed to support TLS highlighted. An explanation for each line is below the code.

```

1  package au.com.forge.security.tls.test;
2
3  import java.io.*;
4  import java.net.*;
5  import au.com.forge.security.tls.*;
6
7  /**
8   * An example client using Protekt
9   * TLS/SSL sockets.<p>
10  *
11  * Copyright © 1997 - 2000 Forge Research Pty Ltd ACN 003 491 576<p>
12  */
13  public class ExampleClient {
14      /*

```

```

15     * Default host and port number to connect to
16     */
17     private static int port = 4433;
18     private static String host = "localhost";
19
20     /**
21     * Connects to a server using TLS/SSL, issues
22     * a HTTP GET request and reads the response.<p>
23     *
24     * @param args command line arguments.
25     */
26     public static void main(String args[]) {
27         try {
28             SSLEnvironment.initialise(false);
29             SSLSocket s = new SSLSocket(host, port);
30             s.startHandshake();
31             s.waitForHandshake();
32
33             OutputStream os = s.getOutputStream();
34             InputStream is = s.getInputStream();
35             InputStreamReader isr = new InputStreamReader(is);
36             BufferedReader reader = new BufferedReader(isr);
37
38             String cmd = "GET / HTTP/1.0\n\n";
39             os.write(cmd.getBytes());
40             os.flush();
41
42             String inputLine;
43             while (true) {
44                 inputLine = reader.readLine();
45                 if (inputLine == null || inputLine.length() == 0)
46                     break;
47
48                 System.out.println(inputLine);
49             }
50
51             reader.close();
52         }
53         catch (SSLException ssle) {
54             ssle.printStackTrace();
55         }
56         catch (IOException ioe) {
57             ioe.printStackTrace();
58         }
59     }
60 }

```

Line 5 imports the `Protekt` classes so they can be used in the program.

Line 28 initialises `Protekt` by loading the keys and certificates and starting the random number generator. This is optional; the first time a connection is created or accepted this will be done. The parameter (`false`) tells `Protekt` it will usually be the client in an SSL handshake.

Line 29 creates the connection to the server using a `Protekt SSLSocket`. Except for the class name, this is exactly the same call that would be made using a standard `Socket`.

Lines 30 and 31 perform the SSL handshake that determines the cryptographic parameters used to protect the application data. This must be done before the `Input` and `Output` streams are accessed.

Lines 53-55 catch the `SSLException` that can be thrown by the `SSLEnvironment.initialise` call.

3.3 Working with cipher suites

3.3.1 Supported cipher suites

Protekt is distributed with the following cipher suites:

```
SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_IDEA_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
```

3.3.2 Specifying which cipher suites can be negotiated for a connection

The default list of available cipher suites (the cipher suites returned by the `SSLSocket.getSupportedCipherSuites` method) is defined in the `Protekt.properties` file. See chapter 5, *The protekt.properties file* for more information on enabling or disabling cipher suites in this manner.

Individual sockets can have this list of available cipher suites modified before the handshake has started using the `SSLSocket.setEnabledCipherSuites` method. This method takes an array of `Strings`, with each element of the array holding the name of a cipher suite that is to be available on that socket.

For example, the following code will ensure that only cipher suites using RSA public keys for authentication and key exchange will be used:

```
String[] suites = {
    "SSL_RSA_WITH_RC4_128_SHA",
    "SSL_RSA_WITH_RC4_128_MD5",
    "SSL_RSA_WITH_IDEA_CBC_SHA",
    "SSL_RSA_WITH_3DES_EDE_CBC_SHA",
    "SSL_RSA_WITH_DES_CBC_SHA",
    "SSL_RSA_EXPORT_WITH_RC4_40_MD5"
};

...
socket.setEnabledCipherSuites(suites);
socket.startHandshake();
socket.waitForHandshake();
```

A list of cipher suites that have been enabled for a particular socket can be retrieved using the `SSLSocket.setEnabledCipherSuites` method. This returns a `String` array, with each element of the array holding the name of a cipher suite.

3.4 Using a specific certificate chain

Protekt allows a specific private key and certificate chain to be used for any connection. The `SSLSocket.setCertificateChain(String alias, String password)` method is used to specify which certificate chain to use. The `alias` argument is the keystore alias for the certificate chain and the `password` argument is the password for that keystore entry. This password may be obtained via a GUI or other interactive means.

For example, to use the private key and certificate chain identified by the “SecureServer” alias protected with the password “48jso3&5rxxw43”:

```
try {
    SSLServerSocket ss = new SSLServerSocket(port);
    while(true) {
        SSLSocket s = (SSLSocket)ss.accept();
        s.setCertificateChain("SecureServer", "48jso3&5rxxw43");

        // code to start the handshake and use the SSL socket...
    }
}
catch(Exception e) {
    e.printStackTrace();
}
```

The `setCertificateChain` method can be used for both TLS servers and clients. If a TLS client receives a certificate request from the server it will send the certificate chain set by the `setCertificateChain` call. If the certificate chain has not been set this way and a client certificate chain has been set in the properties file, the default chain will be sent. If no certificate chain has been set a “no certificate” alert is sent to the server. The “no certificate” alert is not fatal, however a Protekt TLS server will close the connection if it receives this alert when it asks for a client certificate. Other TLS implementations may not close the connection.

See §4.3 *The Java keystore* and chapter 5, *The protekt.properties file* for more information.

3.5 Using client authentication

Non-anonymous TLS servers can ask the TLS client to identify itself with a certificate chain.

As previously mentioned, if a Protekt client receives a certificate request it will send the certificate chain set by the `setCertificateChain` call. If the certificate chain has not been set this way and a client certificate chain has been set in the properties file, the default chain will be sent. If no certificate chain has been set a “no certificate” alert is sent.

A Protekt server socket can be set to issue a certificate request message during the TLS handshake using the `SSLSocket.setNeedClientAuth(boolean flag)` method:

```

try {
    SSLServerSocket ss = new SSLServerSocket(port);
    while(true) {
        SSLSocket s = (SSLSocket)ss.accept();
        s.setNeedClientAuth(true);

        // code to start the handshake and use the SSL socket,
        // as in previous server example
    }
}
catch(Exception e) {
    e.printStackTrace();
}

```

The `SSLServerSocket.setNeedClientAuth(boolean flag)` method is a shortcut for `SSLServerSocket` use. When a connection is accepted and TLS server mode has been set (the default for `SSLServerSockets`), the newly created `SSLSocket` will have its client authentication flag set to follow the `SSLServerSocket` setting.

See §4.3 *The Java keystore* for more information.

3.6 Specifying a handshake timeout

A timeout for the TLS handshake can be specified. If the handshake takes longer than the timeout, then a “handshake failure” alert will be sent.

The timeout stops attacks from clients, which connect, but never handshake, tying up resources. In the case of a plain text `Socket` connecting to a `SSLServerSocket`, it also closes the connection, rather letting the connection stay open forever.

If a timeout is not specified then a default value is used. The default is found in the `Protekt` properties file.

The default timeout value may be overridden for a particular `SSLSocket` using the `SSLSocket.setHandshakeTimeout(long timeout)` method:

```

try {
    SSLServerSocket ss = new SSLServerSocket(port);
    while(true) {
        SSLSocket s = (SSLSocket)ss.accept();
        s.setHandshakeTimeout( 60000 );

        // code to start the handshake and use the SSL socket,
        // as in previous server example
    }
}
catch(Exception e) {
    e.printStackTrace();
}

```

The timeout value is specified in milliseconds. Either the client or the server may set the handshake timeout.

It is important to allow enough time for the handshake, in the case of a web browser, the user may be prompted by numerous dialog boxes to decide whether they trust this server. Setting the timeout to less than 20 seconds will not give the user enough time to respond.

3.7 Reusing sessions

A TLS connection between two hosts consists of a TLS session, and one or more TLS connections associated with the TLS session.

The TLS session holds the identity information for each party (the certificates), the cipher suite that was agreed upon during the initial handshake (the cryptographic algorithms to use) and some basic material for creating the keys for the ciphers used to protect the application data.

The TLS connections associated with the TLS session hold the cryptographic keys in use for that particular connection between the hosts. These keys were generated using the key material from the TLS session.

The first time a TLS connection is established between two hosts, a TLS session must be created which becomes an agreement between the hosts regarding their identity and the cryptographic algorithms they will use. The initial TLS connection is added to this newly created TLS session. Subsequent TLS connections between the hosts can either join this existing TLS session or create a new TLS session.

Reusing existing TLS sessions is significantly faster and more efficient in terms of resources than creating new TLS sessions for each connection between a pair of hosts.

To reuse a TLS session in subsequent connections, the session identifier must be used when creating an `SSLSocket`.

Presented below is an example method that open `SSLockets`, attempting to reuse TLS sessions when possible. The method caches TLS session identifiers in a hashtable (keyed by hostname) and uses those TLS session identifiers when connecting to hosts that have already had TLS sessions created for them.

```
private Hashtable sessions = new Hashtable();

public SSLSocket connectToHost(String host, int port)
    throws IOException {

    byte[] sessionId;
    SSLSocket socket;
    if(sessions.containsKey(host) == true) {
        byte[] sessionId = (byte[])sessions.get(host);
        socket = new SSLSocket(host, port, sessionId);
    }
    else
        socket = new SSLSocket(host, port);

    socket.startHandshake();
    socket.waitForHandshake();
}
```

```
        sessionId = socket.getSession().getSessionId();
        sessions.put(host, sessionId);

    return socket;
}
```

The method always stores the session identifier after creating the connection. This is done because the server is not required to re-use an existing TLS session, even when the client requests an existing, valid TLS session.

By storing the latest session identifier from the server, the client is attempting to store the latest session information, assuming this is most likely to be acceptable to the server for reuse.

If this method is to be used in a multi-threaded application, access to the session identifier cache must be regulated using synchronization.

3.8 Using Protekt in an applet

This section describes how to use Protekt in an applet. It assumes the applet is being run using the Java plug-in VM rather than browser supplied VMs.

To use Protekt in an applet, you must ensure the `Protekt.properties` file and keystore are available in the jar file the applet is contained in, and that the applet either has permission to dynamically add JCE providers at runtime, or the necessary JCE providers are already installed on the client machine.

3.8.1 The Protekt.properties file

When deploying Protekt with an applet, the `Protekt.properties` file must be included in the jar file in the same directory as the Protekt classes – `au/com/forged/security/tls`. This will allow Protekt to read the properties file at runtime without access to the local machine and also ensures the correct `Protekt.properties` file is being loaded.

3.8.2 The keystore

When deploying Protekt with an applet, the keystore must be included in the jar file in the same directory as the Protekt classes – `au/com/forged/security/tls`. This will allow Protekt to read the keystore file at runtime without access to the local machine and also ensures the correct keystore is being loaded.

The `keystore.filename` property should be set to the keystore's name with no leading path. For example, if the keystore is in a file called `au/com/forged/security/tls/AppletKeystore`, the `keystore.filename` property should be set as:

```
Keystore.filename=AppletKeystore
```

3.8.3 Including JCE Providers in the jar file

The easiest way to package Protekt for use with an applet is to include the Protekt classes and any necessary JCE provider classes in the same jar as the applet (along with the Protekt.properties file and the keystore, as described above).

Applet restrictions

The only restriction that has an effect on Protekt running in an applet is the permission to dynamically add JCE providers and set security properties.

There are two solutions: include the necessary JCE providers in the system-wide `<JRE home>/lib/security/java.security` file or have either a system-wide or user security policy that explicitly allows the applet to add the providers and properties it needs.

The first solution means moving the `security.provider.n` lines from the Protekt.properties file to the java.security file. If the providers are listed in this system-wide file, Protekt no longer needs to register them and the lines can be removed from the Protekt.properties file entirely.

The second solution requires an entry similar to the following in a security policy file:

```
grant CodeBase "http://applet.host.com/jars/*" {
    permission java.security.SecurityPermission "insertProvider.*";
    permission java.security.SecurityPermission "putProviderProperty.*";
};
```

This entry will enable classes loaded from the given URL to add providers. The permissions can be made more specific by specifying both lines for each provider used and replacing the "*" with the provider name:

```
grant CodeBase "http://applet.host.com/jars/*" {
    permission java.security.SecurityPermission "insertProvider.Forge";
    permission java.security.SecurityPermission "putProviderProperty.Forge";

    // repeat the above lines as necessary...
};
```

4 Authentication

In addition to data encryption, TLS also provides authentication of one or both parties³.

4.1 *Public key cryptography*

Authentication is achieved using public key cryptography – public and private key pairs. Public key cryptography ensures that if a message can be decrypted using a public key, it must have been created using that public key's matching private key. Similarly, a message created using the public key can only be decrypted successfully using its matching private key.

For authentication purposes, public keys are made publicly available in the form of certificates and are used as identification by the entity named on those certificates.

The matching private key must be kept secret by the entity using the certificate for its identification.

If the private key is only known to the entity identified by the certificate, messages that can be decrypted using the certificate must have been created by the entity identified by that certificate. Similarly, messages encrypted using the public key on the certificate can only be decrypted by the entity identified by that certificate.

If the private key is known to anyone else, they can masquerade as the entity identified by the certificate.

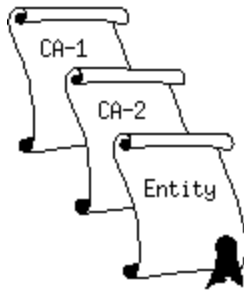
Certificates have safeguards to ensure a third party does not replace the public key in a certificate with one matching their own private key. Certificates also have a date range in which they are valid and are usually issued by Certification Authorities that are supposed to verify the identification of the entity on the certificate before issuing and signing it.

4.2 *Certificate chains and trusted certificates*

A certificate issued by a Certification Authority is issued in a certificate chain. The chain consists of the certificate identifying the private key holder as the first certificate in the chain followed by one or more certificates representing Certification Authorities and ending with the “root” Certification Authorities certificate.

Each certificate in the chain is signed by the Certification Authority above it. For example, a chain with three certificates has two Certification Authority certificates (represented as CA-1 and CA-2 in the following figure) and a certificate identifying the holder of a private key. The chain is:

³ There are anonymous cipher suites defined in the SSL specification but Protekt does not implement them as they offer no authentication of either the client or server.



In this example, CA-1 has identified CA-2 by issuing a certificate to CA-2 and signing that certificate to signify the information on it is correct. When CA-2 creates a certificate chain for an entity, it does the same thing – verifies that entity’s identity, issues a certificate and signs it.

When the entity produces the certificate chain as its identification, the receiver only needs to trust either CA-1 or CA-2 to accept the entity’s identification. To check the certificate chain, the receiver starts at the first signer and works back up the chain until it finds a certificate representing an entity it trusts. If the receiver trusts CA-2, the checking stops as soon as it reads CA-2’s certificate. If the receiver does not recognise CA-2, but trusts CA-1 the checking process will continue past CA-2’s certificate to CA-1’s certificate. If the receiver trusts CA-1 and CA-1 issued CA-2’s certificate, the implication is CA-2 is who it claims to be and the entity is who it claims to be.

The receiver specifies which certificate authorities it trusts using trusted certificates. Trusted certificates are a list of certificates held by the receiver which are checked against incoming certificate chains, as in the above example. Once a certificate in the chain matches a trusted certificate, the chain is accepted.

This does not guarantee the matching private key has not been stolen and is being used by someone else.

4.3 The Java keystore

Protekt uses the Java keystore to hold trusted certificates and private key/certificate chain pairs. Each trusted certificate entry and private key/certificate chain entry is identified by a keystore alias – a short name used to retrieve the entry from the keystore. Each private key/certificate chain entry is also protected by a password.

The trusted certificate entries are used to verify incoming certificate chains and the private key/certificate chain entries are used as identification.

If Protekt is acting as a TLS server, it uses one of the private key/certificate chain entries to identify itself to the client. The specific entry used is determined by a number of factors including the types of public key the client can understand and settings from the `Protekt.properties` file. If client authentication has been requested, the list of trusted certificates is sent to the client so it can make a decision about which certificate chain to send back and the returned certificate chain is checked against the trusted certificates.

If Protekt is acting as a TLS client, the certificate chain received from the TLS server is checked against the trusted certificates in the keystore. If one of the trusted certificates matches a certificate in the incoming certificate chain the connection is accepted. If the server requests client authentication, one of the private key/certificate

chain entries is used to identify the client. The specific entry used is determined by a number of factors including the list of trusted certificates sent by the server, the types of public key the server can understand and settings from the `Protekt.properties` file.

The `keytool` utility provided with Sun's JDK can be used to add trusted certificates, create new private keys and import certificate chains returned from Certification Authorities.

4.3.1 Specifying the keystore to use

The Java keystore can be implemented in a number of ways – as an encrypted disk file, in a tamper-proof hardware device or as an interface to an LDAP directory, for example.

Protekt is independent of the keystore implementation. Which specific keystore to use is defined in the `Protekt.properties` file using the following properties:

<code>keystore.type</code>	<p>The type of keystore to use. Keystore vendors may use different types to identify the keystore implementation.</p> <p>The default is "JKS" which is the keystore provided by Sun.</p>
<code>keystore.provider</code>	<p>The keystore provider to use. The same type of keystore may be implemented by more than one vendor. If there is a specific keystore vendor to be used it can be specified using this property.</p> <p>The default vendor is "SUN".</p>
<code>keystore.loadFromFile</code>	<p>Not all keystores are disk files. Some keystores will be on hardware tokens such as smart cards or cryptographic accelerator boards. This property tells Protekt to ignore the <code>keystore.filename</code> property and to not try and load the keystore from a disk file.</p> <p>This property is optional and defaults to true.</p>
<code>keystore.filename</code>	<p>If the keystores is being loaded from a disk file, this property specifies the name of that file.</p>
<code>keystore.password</code>	<p>Keystores are protected by passwords. This property specifies the password used to unlock the keystore.</p>

If the keystore is to be loaded from a disk file, Protekt first attempts to open the file as a resource, which means it will try to open the named file in the directory that the Protekt classes are loaded from (`au/com/forge/security/tls`). If the Protekt classes have been loaded from a jar file, Protekt will look in that jar file rather than on the disk. If

the file is not there, Protekt tries to open the file using the given name as a filename relative to whatever directory Protekt is running in (or as an absolute path, if one is given).

For example, if the Protekt classes have been loaded from `protekt.jar`, the `keystore.filename` is set to `myKeystore` and `keystore.loadFromFile` is set to `true` (or left unspecified), Protekt will search for the keystore in the following places:

1. as the file `au/com/forge/security/tls/myKeystore` in the `protekt.jar` file
2. as the file `myKeystore` in the current directory

If the `keystore.filename` property is set to “`c:\\home\\myKeystore`”, Protekt will search for the keystore in the following places:

1. as the file `au/com/forge/security/tls/c:/home/myKeystore`
2. as the file `c:\\home\\myKeystore`

If the keystore is being loaded from a hardware token, a file is not necessary. The keystore implementation supplied with the hardware token will know where the keys are stored.

4.3.2 Specifying the server private key/certificate chain

The default private key/certificate chain keystore entry to use when Protekt is in TLS server mode is specified in the `Protekt.properties` file using the following properties:

<code>server.key.alias</code>	The alias of the private key/certificate chain entry.
<code>server.key.password</code>	The password used to unlock the private key/certificate chain entry.

If the default entry should not be used for a particular connection, the `SSLSocket.setCertificateChain` method can be used to specify an alternative keystore entry. The arguments to this method are the keystore alias and password of the private key/certificate chain entry to be used.

4.3.3 Specifying the client private keys/certificate chains

When Protekt is acting as a TLS client, more than one private key/certificate chain entry can be specified as a default. The entry to be used is determined by the list of trusted certificates sent by the server and the key types recognisable by the server.

The following properties in the `Protekt.properties` file are used to specify the default list of private key/certificate chain entries, where `n` is the order of preference:

<code>client.key.alias.n</code>	The alias of the private key/certificate chain entry.
<code>client.key.password.n</code>	The password used to unlock the private key/certificate chain entry.

For example:

```
client.key.alias.1=ClientFromCA-1  
client.key.password.1=password1  
client.key.alias.2=ClientFromCA-2  
client.key.password.2=password2
```

If the default entry should not be used for a particular connection, the `SSLSocket.setCertificateChain` method can be used to specify an alternative keystore entry. The arguments to this method are the keystore alias and password of the private key/certificate chain entry to be used (see the `SSLSocket` javadoc page for details).

5 The `Protekt.properties` file

Protekt uses Java properties to store settings that are likely to change from site to site. These settings govern such behaviour as which encryption algorithms to negotiate with a peer and which certificate chains to send as identification and which JCE providers to use.

This chapter details the properties that can be set. An example `Protekt.properties` file is given in appendix C.

5.1 *Specifying the location of the `Protekt.properties` file*

By default, the JVM will look for the `Protekt.properties` file in each classpath entry by appending `au/com/forge/security/tls/Protekt.properties` to the entry. This will also find the properties file if it is included in a jar file with the Protekt classes. If the properties file is included in the jar file, it must be in the `au/com/forge/security/tls` directory.

You can also specify the location of the `Protekt.properties` file using the `Protekt.properties.path` system property:

```
java -DProtekt.properties.path=/usr/local/app/Protekt.properties
some.app.Main
```

5.2 *JCE provider properties*

Protekt uses JCE providers for all cryptographic processing. Examples include creating or verifying a digital signature, encrypting data or retrieving a private key from a keystore.

5.2.1 Services used by Protekt

Protekt must have all of the following services available from the JCE providers to function⁴:

Message Digests

- MD5
- SHA-1

These are provided by Sun in the JDK/JRE, hence they are always available.

Digital signatures

- MD5withRSA

⁴ Different providers can supply these services, but they must all be present.

Any provider that supplies RSA services is likely to supply this digital signature service.

- SHA1withDSA

This is provided by Sun in the JDK/JRE, hence is always available.

- SSLHandshake

This is only available from the FORGE cryptographic provider, supplied with Protekt.

Ciphers

- DES/ECB/NoPadding
- DESede/ECB/NoPadding
- IDEA/ECB/NoPadding
- RC4

Most JCE providers have these services. Not all of these ciphers need be present, but the appropriate cipher suites must be disabled if a cipher is missing (see §5.3 *Cipher suites*).

Key factories

- RSA
- DES
- IDEA
- RC4
- Diffie-Hellman

Key agreement

- Diffie-Hellman

Most JCE providers have these services. The RSA key factory is necessary for processing certificate chains and must be present. The other key factories may not be necessary, depending on the JCE provider. If a key factory is not present, Protekt attempts to create the key using a different mechanism. This alternative mechanism is not available in all providers, but most providers will supply one or the other mechanism.

The Diffie-Hellman key factory and key agreement method are only necessary if the ephemeral Diffie-Hellman (DHE) cipher suites are used.

5.2.2 The providers to be used

At least two JCE providers must be used – but there is no upper limit. The providers are specified using the a list of `security.provider.n` properties:

```
security.provider.1=com.crypto.provider.SomeProvider
security.provider.2=au.com.forge.provider.ForgeProvider
```

The FORGE provider must be used as it supplies TLS specific signature algorithms. However, the FORGE provider does not supply any data encryption services so at least one other provider must be specified.

Any JCE compliant provider that supplies data encryption and RSA key services should work, but some providers use non-standard parameters or have bugs and consequently do not work as well as others.

5.2.3 The random number generator

The random number generator is very important. If a weak random number generator is used, security is compromised because attackers can more easily guess the sequence of numbers that will be produced. All secret keys used by a TLS implementation (not just Protokt) are created through the use of random number generators.

Java specifies a vendor independent `SecureRandom` interface for programs to access random number generators. A program can specify the random number generation algorithm it wants to use (and optionally a vendor) and the Java code will use that scheme if it is available. To specify the settings for Protokt, use the following properties:

<code>security.SecureRandom.algorithm</code>	The PRNG (pseudo random number generation) algorithm to use. A random number generator based on thread run times is provided with Protokt. To use this generator, specify FORGE as the algorithm.
<code>security.SecureRandom.provider</code>	If a specific vendor's implementation of a random number algorithm is desired, the vendor can be named here. The documentation provided by the vendor should give both the algorithm and provider values.

If these properties are not set, Protokt will use the default random number generator returned by the Java Runtime.

5.2.4 RSA cipher initialisation

Testing against different JCE providers has shown a discrepancy in how RSA ciphers are initialised. The `rsa.cipher.specifier` property is used to tell Protokt how to

initialise an RSA cipher. There is no fixed set of values, but the most common initialiser is `RSA/ECB/PKCS1Padding`. However, some providers need other settings such as `RSA/2/PKCS1Padding`.

This problem is caused by the initialisation of RSA ciphers not being covered in current versions of the JCE specification. Javasoft is aware of this problem and the next version of the specification is promised to be more specific about how RSA ciphers should be initialised.

Until then, the documentation supplied with the JCE provider must be used to determine how this property is set. Any providers that have been tested will be listed on the [Protekt TLS web site](http://www.protekt.com)⁵, with the necessary settings.

5.3 Cipher suites

A TLS cipher suite specifies the secret key exchange algorithm, encryption algorithm and message integrity algorithm used to protect traffic on a connection.

Protekt supports RSA and Diffie-Hellman for secret key exchange, RSA and DSS for key signing, IDEA, DES, triple DES and RC4 for data encryption, and MD5 and SHA-1 for message integrity.

Depending on the cryptographic services available and other factors such as licensing conditions for particular algorithms, some of these algorithms may not be available at run time.

The cipher suites Proték will use are defined with the `ciphersuite.n` properties. The cipher suites should be defined in order of preference – the lower numbered cipher suites will be used in preference to the higher numbered ones. The entries must start at 1 and be consecutively numbered. For example:

```
Ciphersuite.1=au.com.forge.Proték.CS_TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
ciphersuite.2=au.com.forge.Proték.CS_SSL_RSA_WITH_3DES_CBC_SHA
ciphersuite.3=au.com.forge.Proték.CS_SSL_RSA_WITH_IDEA_CBC_SHA
ciphersuite.4=au.com.forge.Proték.CS_SSL_RSA_WITH_RC4_128_SHA
ciphersuite.5=au.com.forge.Proték.CS_SSL_RSA_WITH_RC4_128_MD5
ciphersuite.6=au.com.forge.Proték.CS_SSL_RSA_WITH_DES_CBC_SHA
Ciphersuite.7=au.com.forge.Proték.CS_TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
ciphersuite.8=au.com.forge.Proték.CS_SSL_RSA_EXPORT_WITH_RC4_40_MD5
```

In this case, the triple DES encryption algorithm will be used in preference to IDEA, RC4, or DES and ephemeral Diffie-Hellman key exchange will be used in preference to RSA key exchange if possible.

If IDEA and RC4 are not available, the cipher suite list might look like:

```
Ciphersuite.1=au.com.forge.Proték.CS_TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
ciphersuite.2=au.com.forge.Proték.CS_SSL_RSA_WITH_3DES_CBC_SHA
ciphersuite.3=au.com.forge.Proték.CS_SSL_RSA_WITH_DES_CBC_SHA
Ciphersuite.4=au.com.forge.Proték.CS_TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
```

⁵ <http://www.proték.com>

Another example is if the export (weak) cipher suites and normal DES should not be used, they can be taken out of the list so it becomes:

```
Ciphersuite.1=au.com.forge.Protect.CS_TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
ciphersuite.2=au.com.forge.Protect.CS_SSL_RSA_WITH_IDEA_CBC_SHA
ciphersuite.3=au.com.forge.Protect.CS_SSL_RSA_WITH_3DES_CBC_SHA
ciphersuite.4=au.com.forge.Protect.CS_SSL_RSA_WITH_RC4_128_SHA
ciphersuite.5=au.com.forge.Protect.CS_SSL_RSA_WITH_RC4_128_MD5
```

5.4 Keystore properties

For details of the properties used to specify the private key/certificate chains and trusted certificates to use, see §4.3 *The Java keystore* for more information.

5.5 Additional properties

The default handshake time is specified in the Protect properties file. It is the maximum number of milliseconds that the handshake can take. The example below will set the handshake timeout to 60 seconds.

```
#
# The default handshake timeout in milliseconds.
# This can be overridden using setHandshakeTimeout in SSLSocket.
#
HandshakeTimeout = 60000
```

For more details see §3.6 *Specifying a Handshake Timeout*.

6 Appendix A - Glossary

authenticate

Public and private keys can be used for authentication. If a person holding a private key distributes the certificate with the matching public key, anyone with that certificate can verify signatures created by the private key holder.

This also means if a person can create a signature that is verified by the public key on a certificate they must have access to the matching private key.

cipher suite

The TLS specification defines a number of cipher suites that can be used in a TLS conversation. The cipher suite defines a method of secret key exchange, a signature algorithm to use, a bulk data cipher algorithm and a message authentication code algorithm.

The key exchange and signature algorithms are used during the TLS handshake for authentication and to agree on the keys used for the bulk data cipher.

The bulk cipher and message authentication code algorithms are used for reading and writing data to the SSL socket's input and output streams.

JCE

The Java Cryptography Extension (JCE) 1.2 provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. JCE 1.2 is designed so that other cryptography libraries can be plugged in as a service provider, and new algorithms can be added seamlessly.

key pair

A key pair refers to matching public and private keys. Data can be signed using a private key and the resulting signature can be verified using the matching public key. Data can be encrypted using the public key and decrypted using the matching private key.

public key

A public key can be used to verify data signed using a matching private key and can be used to encrypt data that can only be decrypted using the matching private key.

private key

A private key is used to sign data and to decrypt data encrypted using the matching public key.

root CA

A root CA is the last certificate in a certificate chain. If a chain consists of 3 certificates – A, B and C – where A represents the entity and has been signed by the holder of B and B is an intermediate CA that has been verified by C, then C is the root CA.

VM (or JVM)

The Java Virtual Machine. To give Java platform independence, Javasoft created what is called the Java Virtual Machine. The JVM defines a basic computer architecture that can be implemented on almost any real computing platform. All Java applications and applets are compiled to run in a JVM rather than being compiled to use the machine instructions and operating system calls of any specific system.

X.509 certificate

An X.509 certificate binds a public key to more recognisable names. An entity holding a private key usually has a certificate with its public key. The certificate can be given to other parties so they can both: verify signatures created by the private key, and encrypt data to send to the holder of the private key.

7 Appendix B – Troubleshooting

7.1 Keystore problems

7.1.1 Could not load keystore from <filename>

Protekt throws an `SSLException` with this message if the keystore is to be loaded from a file and the file cannot be found.

See §4.3.1 *Specifying the keystore to use* for a description of where Protekt looks for the keystore file.

7.1.2 Could not load specified certificate chain (<alias> not found in the keystore)

Protekt throws an `IOException` with this message during the handshake if it cannot find the requested keystore entry in the keystore.

Ensure <alias> is in the keystore by issuing a `keytool -list -keystore keystore` command, and ensure the correct keystore is being loaded. See §4.3.1 *Specifying the keystore to use* for a description of where Protekt looks for the keystore file.

7.1.3 Could not load specified certificate chain (<alias> is a trusted certificate entry, not a key entry)

Protekt throws an `IOException` with this message during the handshake if the keystore entry specified to be used for identification is a trusted certificate entry, rather than a key (private key, public key certificate chain) entry.

This often happens when a certificate signing request has been created from a locally generated key pair (key entry) and the signed certificate is imported under a different alias, rather than under the alias of the original key pair.

See Appendix C, *Importing a certificate signed by a CA* for a detailed description of generating a key pair and getting the public key certificate signed by a CA.

7.2 JCE problems

7.2.1 No Cipher could be found for algorithm <algorithm>

The JCE providers are not installed correctly or the RSA cipher specifier is set incorrectly.

Protekt requires at least two JCE providers; the Forge provider that supplies a TLS specific digital signature algorithm and a second provider that supplies all the encryption algorithms (including the RSA cipher used by the Forge provider).

See §5.2 *JCE provider properties* for more information about how Protekt uses JCE providers.

8 Appendix C – Importing a certificate signed by a CA

This appendix presents a detailed example of generating a key pair, creating a certificate signing request to send to a CA and importing the certificate returned from the CA into a keystore.

This example assumes the keystore is a standard JKS keystore contained in a file called `example.ks`.

8.1 Making JCE Providers available to the keytool

The `keytool` command line tool does not use a classpath and does not dynamically register JCE Security Providers. This means the Java runtime must have the providers registered and be able to find the necessary classes by default.

To allow the Java runtime to find the classes for the providers, place the jar file(s) for each provider in the `<JRE Home>/lib/ext` directory, and list each provider's main class in the `<JRE Home>/lib/security/java.security` file. For example, the Forge and ABA providers are distributed with Protekt; to make these providers available to all programs put the `protekt.jar` and `aba-1.1.jar` files in the `<JRE Home>/lib/ext` directory and modify the `<JRE Home>/lib/security/java.security` file to include the lines shown in bold:

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=au.com.forge.provider.ForgeProvider
security.provider.3=au.net.aba.crypto.provider.ABAProvider
```

8.2 Generating a key pair

The following command generates an RSA key pair, storing the private key and self-signed certificate under the name `www.example.com`.

```
$ keytool -genkey -alias www.example.com -keyalg RSA \
-keysize 1024 -keystore example.ks<enter>
Enter keystore password: password<enter>
What is your first and last name?
 [Unknown]: www.example.com<enter>
What is the name of your organizational unit?
 [Unknown]: <enter>
What is the name of your organization?
 [Unknown]: Example<enter>
What is the name of your City or Locality?
 [Unknown]: <enter>
What is the name of your State or Province?
 [Unknown]: <enter>
What is the two-letter country code for this unit?
 [Unknown]: <enter>
Is <CN=www.example.com, OU=Unknown, O=Example, L=Unknown, ST=Unknown,
C=Unknown> correct?
 [no]: yes<enter>
```

```
Enter key password for <www.example.com>
(RETURN if same as keystore password): <enter>
```

When creating a key pair, at least the name (CN), organisation (O), State (ST) and country (C) attributes should be entered.

8.3 Creating a certificate signing request

The certificate created by the `-genkey` command is a self-signed certificate. It will not be trusted by other entities until a well-recognised certificate authority (CA) has signed it.

To get a self-signed certificate signed by a CA you must generate a certificate-signing request for that certificate and send it to a CA. The following keytool command line creates the certificate signing request and saves it in a file called `example.csr`:

```
$ keytool -certreq -alias www.example.com -file example.csr \
-keystore example.ks<enter>
Enter keystore password: password<enter>
```

The certificate-signing request should look similar to:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBtDCCAR0CAQAwdDEQMA4GA1UEBhMHVW5rbm93bjeQMA4GA1UECBMHVW5rbm93bjeQMA4GA1UE
BxMHVW5rbm93bjeQMA4GA1UEChMHRXhhbXBsZTEQMA4GA1UECzMHVW5rbm93bjeYMBYGA1UEAxMP
d3d3LmV4YW1wbGUuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJFmPxtJK6kd7uVmC
hLz9UgoeV3ud5yw/JNnOg0KTzyHWp16IMfnyi/6iHAZxWlueWEaWG2cMoLlMkajyTPo0GQRnJBPi
QDxtQRkaITHqK/z/RthXoSSDgbsec510x/tHQ36v/egsZgE/ECskc19rQynrIaTHT+w4LyzA+nbU
lQIDAQABoAAwdQYJKoZIhvcNAQEEBQADgYEAoeDiwjhWGHqFKvCYObaqjoky79eN4CmqNL9aQ5hJ
FT07TLGS0Gw7Y//VVGRcrtLzsW2yHdICiJNeb8j4vDL3Be9i/kV8I9P52UTQzqbZWQYNC7SwTnzm
hFlbzxpOKxKcJawBe2fM44MtP31a/cC2q8JznkcqC7w8BvHD8hdhHEw=
-----END NEW CERTIFICATE REQUEST-----
```

8.4 Importing the CA root certificate (optional)

`keytool` requires that the certificate chain returned from the CA can be traced to a trusted certificate before it will import the certificate chain (see §4.2 *Certificate chains and trusted certificates*).

If `keytool` reports an error similar to “keytool error: Failed to establish chain from reply”, it means the certificate chain returned from the CA has not been signed by any of the trusted certificates in the keystore and the CA’s root certificate will need to be imported.

To get a CA’s root certificates, contact the CA directly. Once you have the CA’s certificate, do the following:

```
$ keytool -import -alias serverbasic -file serverbasic.crt \
-keystore example.ks<enter>
Enter keystore password: password<enter>
Owner: EmailAddress=server-certs@thawte.com, CN=Thawte Server CA,
OU=Certification Services Division, O=Thawte Consulting cc, L=Cape
Town, ST=Western Cape, C=ZA
Issuer: EmailAddress=server-certs@thawte.com, CN=Thawte Server CA,
OU=Certification Services Division, O=Thawte Consulting cc, L=Cape
Town, ST=Western Cape, C=ZA
```

```

Serial number: 1
Valid from: Thu Aug 01 10:00:00 GMT+10:00 1996 until: Fri Jan 01
10:59:59 GMT+11:00 2021
Certificate fingerprints:
    MD5:  C5:70:C4:A2:ED:53:78:0C:C8:10:53:81:64:CB:D0:1D
    SHA1: 23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
Trust this certificate? [no]:  yes<enter>
Certificate was added to keystore

```

8.5 Importing the certificate returned from the CA

If the CA returns the signed certificate as a base 64 encoded DER file, the `keytool -import` command can be used to replace the self-signed certificate generated in step 2 with the certificate signed by the CA.

The following is an example of a base 64 encoded DER file:

```

-----BEGIN CERTIFICATE-----
MIIDECCAnmgAwIBAgIDAKVGMA0GCSqGSIb3DQEBAUAMIHEMQswCQYDVQQGEwJa
QTEVMBMGALUECBMMV2VzdGvYbiBDYXB1MRIwEAYDVQQHEw1DYXB1IFRvd24xHTAb
BgnVBAAoTFFRoYXZ0ZSBDb25zdWx0aW5nIGNjMSgwJgYDVQQLEx9DZXJ0aWZpY2F0
aW9uIFNlcnZpY2VzIERpdmlzaW9uMRkwFwYDVQQDExBUaGF3dGUgU2VydmlVYIENB
MSYwJAYJKoZIhvcNAQkBFhdzZXJ2ZXI0tY2VydHNAAGhhd3RlLmNvbTAeFw05OTEy
MDEwNjMzMTBaFw0wMDEyMTQwNjMzMTBaMIGtMQswCQYDVQQGEwJBVTEEMMAoGA1UE
CBMDElNXMQ8wDQYDVQQHEwZTeWRuZkxkMTAvBgNVBAoTKEYVcmQ1IE1uZm9ybWFO
aW9uIFRlY2hub2xvZ3kgUHR5IEExpbW10ZWQxMTAvBgNVBAsTKEYVcmQ1IE1uZm9y
bWFOaW9uIFRlY2hub2xvZ3kgUHR5IEExpbW10ZWQxMTAvBgNVBAMTEHd3dy5mb3Jn
ZS5jb20uYXUwZ3RwDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAIuW3Xc0Vgy+0r8/
wPRPnUneDTHJ8oF6yVKLeQXAUN1pEK4Hc+q1/XV0j80aOLEbqn9rseQ7TDh062TT
LDpq603kN8FO/eHx1NdpV52EUhNPC7kZZUbjCpCx2Ts22JtkEwxYsrOaRPr+AUB
AUPruZ+YrpomQtG4CwKmcPbPFXabAgMBAAGjJTAjMBMGALUdJQQMAoGCCsGAQUF
BwMBMAwGALUdEwEB/wQCMAAwDQYJKoZIhvcNAQEBBQADgYEAWE1nmvnEn3jZuMkK
8kwuoagj/JwzzdFvmq8qZ6ytFIOs5MpWp4peLYI+j+8nN0ve7XAWjy6hOAEmxA
jHWwESSKcDuIm25sIvwji74fY303xQr2vDNuqC+GJKY72mMHdAlgkRr7G5ZzvfOQ
mwPChVO/+90necnt+oPhJ+JvNM=
-----END CERTIFICATE-----

```

The command to replace the self-signed certificate with the certificate returned from the CA is:

```

$ keytool -import -alias www.example.com \
  -file exampleCertificate.pem \
  -keystore example.ks<enter>
Enter keystore password:  password<enter>

```

9 Appendix D – Example protekt.properties file

```
#
# Protekt 3.0
#
#
# Whether the Protekt library can perform server
# handshakes.
#
# If set to true, Protekt can be either the client or
# server side of a connection.
#
# If this is set to true the library can act as either
# an SSL client or server. If false, only client
# handshakes can be done.
#
Server=true
#
# List of JCE security providers that will be used for
# cryptographic services. The class named is the provider
# main class that is used to register the provider with
# the JCE.
#
# The FORGE provider must always be used. It can be put
# last but must be present.
#
# The FORGE and ABA providers are included in the
# distribution. See the readme.txt file for more
# information on providers.
#
security.provider.1=au.com.forge.provider.ForgeProvider
security.provider.2=au.net.aba.crypto.provider.ABAProvider
#
# Some JCE providers have a bug in the Cipher implementation
# that causes encryption/decryption to fail if the same
# buffer is used for input and output. Setting this property
# to true will cause Protekt to allocate a temporary buffer
# for the encryption/decryption to work around the bug.
#
# The ABA 1.1 provider is one of the providers that has this bug.
#
useSeparateEncryptionBuffer=true
#
# Use one or both of these to specify the SecureRandom
# implementation you want to use. If you want to use
# anything other than Sun's default you should use these.
#
# If both are filled in, that algorithm from that provider
# is requested. If the algorithm is filled in, that
# algorithm from any provider is requested. If neither are
# used, the default is used. This will usually be Sun, as
# the providers listed about are added to the end of the
# provider list, not the front.
#
security.SecureRandom.algorithm=
security.SecureRandom.provider=
#
# Different providers require different specifiers when
# creating raw RSA ciphers. The two most common forms are
# given below, but if neither of them work, consult the
# providers programming documentation.
#
# The most likely part to change is the middle part - the
# mode. Javasoft have said that ECB is what should go there
# but some providers require the PKCS1 block type to be
# there instead.
#
#
# This works for DSTC
```

```

#
#rsa.cipher.specifier=RSA/2/PKCS1Padding

#
# The list of cipher suites enabled in Protekt.
#
# The cipher suites are listed in order of preference.
#
# To disable a cipher suite, comment it out and change the
# numbers on the following cipher suites.
#
ciphersuite.1=au.com.forge.security.tls.CS_SSL_RSA_WITH_RC4_128_SHA
ciphersuite.2=au.com.forge.security.tls.CS_SSL_RSA_WITH_RC4_128_MD5
ciphersuite.3=au.com.forge.security.tls.CS_SSL_RSA_WITH_IDEA_CBC_SHA
ciphersuite.4=au.com.forge.security.tls.CS_SSL_RSA_WITH_3DES_EDE_CBC_SHA
ciphersuite.5=au.com.forge.security.tls.CS_SSL_RSA_WITH_DES_CBC_SHA
ciphersuite.6=au.com.forge.security.tls.CS_SSL_RSA_EXPORT_WITH_RC4_40_MD5

#
# This properties file has the server certificate set to the
# one with the RSA key, so these cipher suites will not work
# in server mode unless the SSLSocket.setCertificateChain method
# is used (the signature will fail because the wrong key type
# will be used). To use these cipher suites in server mode,
# change the server.key.alias to "dsa1024" and comment out
# the RSA based cipher suites above.
#
ciphersuite.7=au.com.forge.security.tls.CS_SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
ciphersuite.8=au.com.forge.security.tls.CS_SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
ciphersuite.9=au.com.forge.security.tls.CS_SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

#
# Keystore definitions
#
# The type defaults to JKS and the provider is optional. The
# filename and password are not optional.
#

keystore.type=JKS
#keystore.provider=SUN
keystore.filename=keystore
keystore.password=password

#
# Server private key and certificate chain
#
# The example key store distributed with Protekt has three
# server certificates included:
#
# "rsa512" has a 512 bit RSA key
# "rsa1024" has a 1024 bit RSA key
# "dsa1024" has a 1024 bit DSA key
#
# It is important to have the appropriate key listed below
# for the cipher suites you are using. The RSA key used with
# a DSA cipher suite will cause the connection to fail.
#

server.key.alias=rsa1024
server.key.password=password

#
# Client private keys and certificate chains
#
# These should be listed in the order of preference.
# If a certificate request is received from a server, these
# chains will be checked and the first one that matches
# a server acceptable CA and the cipher suite will be used.
#
# The first chain is specified as client.key.alias.1 and
# client.key.password.1. The next is client.key.alias.2
# and so on.
#
# As with servers, the chain to be sent can also be set
# at run time using the SSLSocket.setCertificateChain
# method. This can only be done before the certificate
# request message is received, so if that chain doesn't

```

```
# match the server acceptable CAs and the cipher suite,
# no luck.
#

client.key.alias.1=clrsa1024
client.key.password.1=password
client.key.alias.2=cldsa1024
client.key.password.2=password

#
# The session manager is a thread that reads through
# the session list and removes any sessions that are
# set to inactive (ie cannot have any more connections
# added to them), have no active connections attached
# to them and have not seen any activity for at least
# SessionManager.timeout milliseconds.
#
# SessionManager.period specifies how often the
# session manager reads the session list.
#
SessionManager.period=300000
SessionManager.timeout=600000

#
# If a handshake has not completed within the time
# specified by HandshakeTimeout (in milliseconds)
# the handshake is deemed to have failed. If
# SSLSocket.waitForHandshake has been called, it
# will throw an IOException.
#
HandshakeTimeout=120000
```