



ClassGuard Quick Start Guide

Version 3.2

Table Of Contents

1	Introduction.....	3
1.1	Feature list.....	3
2	Release notes.....	4
2.1	Notes for users of version 1.x.....	4
2.2	Notes for users of version 2.0.....	4
2.3	Notes for users of version 2.5.....	4
2.4	Notes for users of version 3.0.....	4
3	First demo application.....	5
4	Encrypting applications.....	6
4.1	The Command Line Interface.....	6
4.2	Encryption by command line.....	6
4.3	The ClassGuard crypt ant task.....	6
4.4	Encrypting multiple jar files.....	7
4.5	Reflection based frameworks.....	8
5	Using ClassGuard as a license manager.....	9
5.1	Encrypting your application.....	9
5.2	Creating licenses by GUI.....	9
5.2.1	Creating a product key.....	10
5.2.2	Creating a license.....	10
5.3	Creating licenses by command line.....	10
5.4	Implementing your own license GUI or license storage.....	10
6	Using ClassGuard in combination with J2EE containers.....	11
6.1	Using ClassGuard in combination with Tomcat.....	11
6.2	Using ClassGuard in combination with Jboss.....	12

1 Introduction

ClassGuard is a tool to prevent Java decompiling and for license management of Java applications. The Java class files are encrypted using a 128Bit AES encryption. The AES key is generated randomly every time you start the encryption tool.

The decryption is done transparently by a custom class loader. The main part of this class loader is written in C to prevent decompiling and other tampering.

If license management is activated, classes are only decrypted when a valid license is available.

ClassGuard supports Sun Java 1.5 and 1.6 (aka Java5 and Java6). OpenJDK currently is not supported. Other Java versions may work, but are not supported.

There are two ways of invoking ClassGuard:

- ➔ A simple command line interface (CLI)
- ➔ Ant tasks

An evaluation version of ClassGuard is available for download at

<http://www.jsecurity.net/classguard/download.html>

1.1 Feature list

This version contains the following features:

- ➔ Transparent class encryption
- ➔ License management (by CLI and GUI)
- ➔ Built-in application starter (new)
- ➔ Tomcat support
- ➔ Support for multiple jar files
- ➔ Ant task
- ➔ Resource encryption
- ➔ JBoss support

2 Release notes

2.1 Notes for users of version 1.x

- ➔ The package structure of ClassGuard has changed. The complete path of the ClassGuard class is now `net.jsecurity.classguard.bootstrap.ClassGuard`.
- ➔ The workflow for creating multiple jar files sharing the same encryption key has changed.

2.2 Notes for users of version 2.0

- ➔ ClassGuard now has a built in application starter, including a GUI for requesting licenses. Implementing your own startup code or setting the system class loader is now only necessary in exceptional cases.
- ➔ The syntax for setting custom attributes in licenses has slightly changed.

2.3 Notes for users of version 2.5

- ➔ ClassGuard now has a GUI for creating licenses.

2.4 Notes for users of version 3.0

- ➔ TomcatClassGuard.jar and JbossClassGuard.sar have changed.

3 First demo application

For your first test, use an application which is completely contained in one jar file. You may create your own application jar file (e.g. using ant or the jar command line utility) or you can use an existing demo application. In this example, we use the SwingSet2 demo from the JDK demo directory.

Encrypt your jar file using

```
java -jar ClassGuard.jar crypt -in SwingSet2.jar -out SwingSet2ClassGuard.jar  
-resource java
```

You may now start your encrypted application using

```
java -jar SwingSet2ClassGuard.jar
```

The SwingSet2 demo now starts up.

4 Encrypting applications

4.1 The Command Line Interface

The ClassGuard CLI is started using

```
java -jar ClassGuard.jar <action> <options>
```

If started without an action, the license GUI appears. If started with the action *help*, a short description of command line parameters is displayed.

4.2 Encryption by command line

Applications are encrypted using the *crypt* action. The following options are available:

<i>-in my_project.jar</i>	Jar file to encrypt
<i>-out crypted_jarfile.jar</i>	Destination jar file. Must not be identical to source jar file.
<i>-include packages</i>	Comma-separated list of packages or class files to encrypt in jar file. Format is <i>com/example/package1, com/example/package2/Class1.class</i> . Default is all packages are included.
<i>-exclude packages</i>	Comma-separated list of packages or class files to exclude from encryption in same format. Default is no packages are excluded.
<i>-platform platforms</i>	Comma-separated list of platforms for the destination file. Possible platforms in the evaluation version are <i>x86_win, x86_osx, x86_linux and x64_linux</i> .
<i>-cryptkey keyfile</i>	Import encryption key from file
<i>-includelibs</i>	Force inclusion of ClassGuard libs in importkey mode
<i>-war</i>	War mode, useful for web applications
<i>-resource</i>	Comma-separated list of extensions of resources to encrypt. Default is no resource encryption.
<i>-startclass</i>	Class for starting up this jar file. By default, the class defined in the attribute <i>Main-Class</i> of <i>META-INF/MANIFEST.MF</i> is used.

So encrypting the package *myproject.business* including all property files in your jar file *myproject.jar* (leaving all other packages unencrypted) would look like this:

```
java -jar ClassGuard.jar crypt -in myproject.jar -out myproject_c.jar -include myproject/business -resource properties
```

4.3 The ClassGuard crypt ant task

The ClassGuard ant task is defined using

```
<taskdef classpath="ClassGuard.jar"
  resource="net/jsecurity/classguard/bootstrap/antlib.xml" />
```

Currently there is one task *crypt* with roughly the same parameters as the crypt command line action. The following options are available:

<i>in my_project.jar</i>	Jar file to encrypt
<i>out crypted_jarfile.jar</i>	Destination jar file. Must not be identical to source jar file.
<i>include packages</i>	Comma-separated list of packages or class files to encrypt in jar file. Format is <i>com/example/package1, com/example/package2/Class1.class</i> . Default is all packages are included.
<i>exclude packages</i>	Comma-separated list of packages or class files to exclude from encryption in same format. Default is no packages are excluded.
<i>platform platforms</i>	Comma-separated list of platforms for the destination file. Possible platforms in the evaluation version are <i>x86_win, x86_osx, x86_linux and x64_linux</i> .
<i>cryptkey keyfile</i>	Import encryption key from file
<i>includelibs true/false</i>	Force inclusion of ClassGuard libs in importkey mode. Default false.
<i>war true/false</i>	War mode, useful for web applications. Default false.
<i>resource</i>	Comma-separated list of extensions of resources to encrypt. Default is no resource encryption.
<i>startclass</i>	Class for starting up this jar file. By default, the class defined in the attribute <i>Main-Class</i> of <i>META-INF/MANIFEST.MF</i> is used.

Using the ant task, the above example would look like this:

```
<taskdef classpath="ClassGuard.jar"
  resource="net/jsecurity/classguard/bootstrap/antlib.xml" />
<crypt in="myproject.jar" out="myproject_c.jar" include="myproject/business"
  resource="properties" />
```

4.4 Encrypting multiple jar files

Usually ClassGuard generates a new encryption key every time it is started. In some cases, this is inconvenient. An application consisting of multiple jar files must share the same encryption key. In this case, first create the encryption key using

```
java -jar ClassGuard.jar createkey -cryptkey key.txt
```

Then use the created encryption key for crypting each of the jar files. Only one of the jar files should contain the ClassGuard libraries (usually the launcher).

```
java -jar ClassGuard.jar crypt -in file1.jar -out file1c.jar -includelibs
  -cryptkey key.txt
java -jar ClassGuard.jar crypt -in file2.jar -out file2c.jar -cryptkey key.txt
```

You should change the encryption key as often as possible, e.g. for each new version.

An encryption key can also be created using the createkey ant task, e.g.

```
<createkey cryptkey="key.txt" />
```

4.5 Reflection based frameworks

When using reflection based frameworks (e.g. Spring), it may be useful to use ClassGuard as the system class loader. In this case, the automated launch feature of ClassGuard may not be used. Launching your application may look like this:

```
java -cp myproject_c.jar  
-Djava.system.class.loader=net.jsecurity.classguard.bootstrap.ClassGuard  
myproject.Start
```

5 Using ClassGuard as a license manager

The license management features of ClassGuard are only available in the full version. The evaluation version contains license management, but all generated licenses expire when your evaluation copy of ClassGuard expires.

5.1 Encrypting your application

For using the license manager, you have to create a product key firstly:

```
java -jar ClassGuard.jar createkey -productkey myproduct.key
```

A product key is unique for one of your products. The resulting key file should be kept secret, as everyone owning this key file may generate licenses for this product.

When encrypting your licensed application, you have to supply your product key and your product name:

```
java -jar ClassGuard.jar crypt -in ... -out ... -productkey myproduct.key  
-productname myproduct
```

The resulting application will only run if a matching license is available.

If no license is available, the application starter will ask for a license. This license will be stored in the java registry. On windows, the storage path is in the windows registry in *HKCU\Software\JavaSoft\Prefs\net\jsecurity\classguard\bootstrap\<productname>*. On Unix and Linux, the file for storage of the license is *~/java/.userPrefs/net/jsecurity/classguard/bootstrap/prefs.xml*.

5.2 Creating licenses by GUI

After starting ClassGuard without command line parameters, the license GUI appears.

Attribute	Value
-----------	-------

5.2.1 Creating a product key

A product key may be generated and saved to disk using *File/New product key*. An existing product key can be loaded using *File/Open product key*.

5.2.2 Creating a license

For creating a license, you have to load or create a product key and fill in the system id delivered by your customer. For evaluation licenses, a system id is not necessary. A new license may be created using the *Create license* button.

5.3 Creating licenses by command line

Licenses for your product may be created using

```
java -jar ClassGuard.jar license -productkey myproduct.key -licensefile
  license.txt licenseoptions
```

The following options are available:

<i>-type H U E</i>	License type host-based/user-based/eval (required)
<i>-productkey keyfile</i>	Create license for this product (required)
<i>-licensefile file</i>	License file for output (required)
<i>-comment comment</i>	License comment
<i>-expire days</i>	License expiration in days
<i>-systemid id</i>	System id delivered by customer (required for type H and U)
<i>-attribute attrib</i>	Custom attribute (key=value), may be used multiple times

5.4 Implementing your own license GUI or license storage

If you want to store the license in your own file or you want your own GUI for requesting the license, you have to implement the interface *net.jsecurity.classguard.bootstrap.LicenseInterface*. You have to define your own implementation by setting the attribute *ClassGuard-License-Interface* in *META-INF/MANIFEST.MF*.

6 Using ClassGuard in combination with J2EE containers

6.1 Using ClassGuard in combination with Tomcat

As of Version 1.5, ClassGuard supports Tomcat containers. Dynamic starting, stopping and reloading of web applications is also supported.

To use ClassGuard in combination with tomcat, perform these simple steps:

1. Configure your web application for using the ClassGuard tomcat class loader. The easiest way to do so is to put a file "context.xml" into the META-INF directory of your web application, containing the following snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
  <Loader loaderClass="net.jsecurity.classguard.support.TomcatClassGuard"
    useSystemClassLoaderAsParent="false" />
</Context>
```

Other ways of configuring the context of your web application will also work (see <http://tomcat.apache.org/tomcat-5.5-doc/config/context.html>).

2. Encrypt your war file using the -war option, e.g.

```
java -jar ClassGuard.jar crypt -in myapp.war -out my_encrypted_app.war -war
```

All classes contained in WEB-INF/classes will be encrypted.

OR

Encrypt the jar file containing your application and put it into the WEB-INF/lib directory of your war file.

```
java -jar ClassGuard.jar crypt -in myapp.jar -out my_encrypted_app.jar
```

Don't use the -war option in this case. It is not possible to combine both methods for one web application.

3. For tomcat 5.5, put the ClassGuard tomcat class loader *TomcatClassGuard.jar* in the *server/lib* directory of your tomcat installation. The *common/lib* and *shared/lib* directories won't work. For tomcat 6, put this file in the *lib* directory of your tomcat installation. TomcatClassGuard.jar does not contain any keys and may be shared between applications and even between different application vendors.

4. Start tomcat and deploy your application as usual.

If you want to protect jsp files, pre-compile them as classes. You can use JSPC to do this (see <http://tomcat.apache.org/tomcat-5.5-doc/jasper-howto.html>).

The Tomcat class loading mechanism works differently for listeners. They are not loaded by the class loader configured in the application context. Therefore, it is not possible to encrypt listener classes or any classes which are called directly by listeners. You can exclude these classes from encryption using the -exclude parameter.

6.2 Using ClassGuard in combination with Jboss

For encrypting Jboss application, you have to include *JbossClassGuard.sar* in your ear file.

The following example *jboss-app.xml* registers the Jboss ClassGuard service:

```
<jboss-app>
  <module>
    <service>JbossClassGuard.sar</service>
  </module>
</jboss-app>
```

If multiple encrypted jar files are included in the ear file, all jars have to share the same encryption key. J2EE bean files may not be encrypted, because Jboss needs access to the byte code.