

Cinnabar Canner User's Guide

Cinnabar Canner User's Guide

Copyright © 2003-2010 Cinnabar Systems, LLC. All Rights Reserved.

Cinnabar License Manager, CLM, Cinnabar Canner, and Canner are trademarks of Cinnabar Systems LLC.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries.

Table of Contents

| | |
|--|----|
| 1. Welcome to Cinnabar Canner | 1 |
| 1.1. Introduction | 1 |
| 1.2. Overview | 1 |
| 2. Quickstart Tutorial | 2 |
| 2.1. Canner GUI | 2 |
| 2.2. Launch Canner | 2 |
| 2.3. Select Application JAR File | 2 |
| 2.4. Select a Template File | 3 |
| 2.5. Select an Output File | 3 |
| 2.6. Encryption Key | 3 |
| 2.7. Start the Canning Process | 3 |
| 2.8. Run the Application | 4 |
| 3. Canner Configuration | 6 |
| 3.1. Canner Options | 6 |
| 3.2. Application Options | 6 |
| 3.3. JVM Options | 7 |
| 3.4. Encryption Options | 10 |
| 4. Running Canner | 12 |
| 4.1. Command Line Operation | 12 |
| 4.2. Command Line Options | 12 |
| 4.3. Command Line Parameters | 13 |
| 4.4. Configuration via XML | 13 |
| 5. Ant Integration | 18 |
| 5.1. Ant Task | 18 |
| 5.2. Defining the Task | 18 |
| 5.3. Invoking the Canner Task | 18 |

Chapter 1. Welcome to Cinnabar Canner

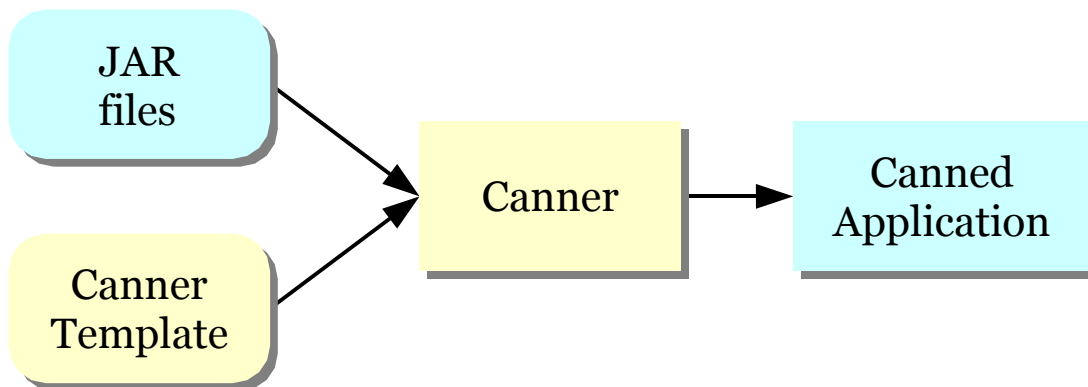
1.1. Introduction

With Cinnabar Canner, you can protect your Java applications against reverse engineering and decompilation. Traditionally, developers have used obfuscators to perform this task, but obfuscation merely renames your classes and methods, leaving the structure and logic of your application intact and visible. Obfuscation only slows down a determined attacker; Canner prevents attack altogether by encrypting your code and resources and then packaging them into a native executable file. Decryption is performed transparently and on the fly, so no cleartext version of your application is ever available to prying eyes.

1.2. Overview

There are two inputs to the "canning" process: one or more JAR files, and a template file. The JAR files contain the classes and resources which compose the application you wish to protect with Canner. The template file, included with your Canner installation, contains the native code framework needed to decrypt and launch your application. These files are read by the Canner application and combined to create the final output: a single executable file which contains your application's classes and resources, secure from reverse engineering.

Figure 1.1. The Canning Process



Chapter 2. Quickstart Tutorial

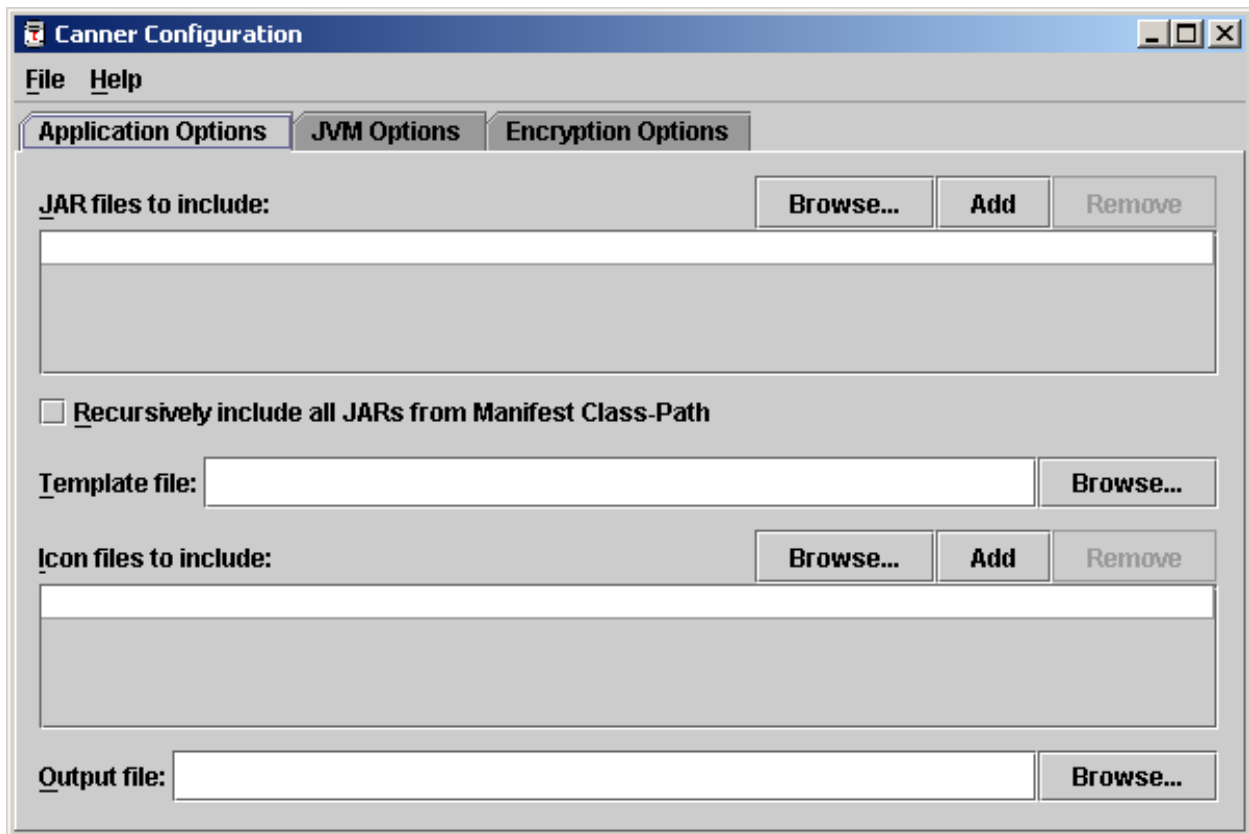
2.1. Canner GUI

The easiest way to understand how to use Canner is to use its GUI to build a simple application. In this example, you will "can" the SwingSet2 JFC sample application included with the JDK.

2.2. Launch Canner

Launch the Canner GUI from the Start menu by clicking Start/Programs/Cinnabar Systems/Canner/Canner, or by running `<installation directory>/canner/bin/canner -u`. The Canner Configuration window should appear:

Figure 2.1. Canner Configuration Window



2.3. Select Application JAR File

First, you must tell Canner the location of your application's classes. Press the "Browse..." button next to the label "JAR files to include...". In the dialog that appears, navigate to where your JDK was installed. Under this directory, navigate to `demo/jfc/SwingSet2` and select the file `SwingSet2.jar`. Press the "Open" button.

2.4. Select a Template File

Next, you must specify a Template File, which is a special file included in your Canner installation that contains the native code framework necessary to launch and decrypt your canned application. Two templates are included, `template.can` and `templatew.can`. `template.can` is for command-line Java applications, and `templatew.can` is for GUI-based Java applications. (These files correspond to the JRE commands **java** and **javaw** respectively.) Since `SwingSet2` is a GUI application, you should use `templatew.can` for this example. Press the "Browse..." button next to the label "Template file:". In the dialog that appears, select the file `templatew.can` and press the "Select" button.

2.5. Select an Output File

In the field next to "Output File:", type `SwingSet2.exe`. Canner will combine the application JAR file (`SwingSet2.jar`) and the template file (`templatew.can`) and create this file, your "canned" application.

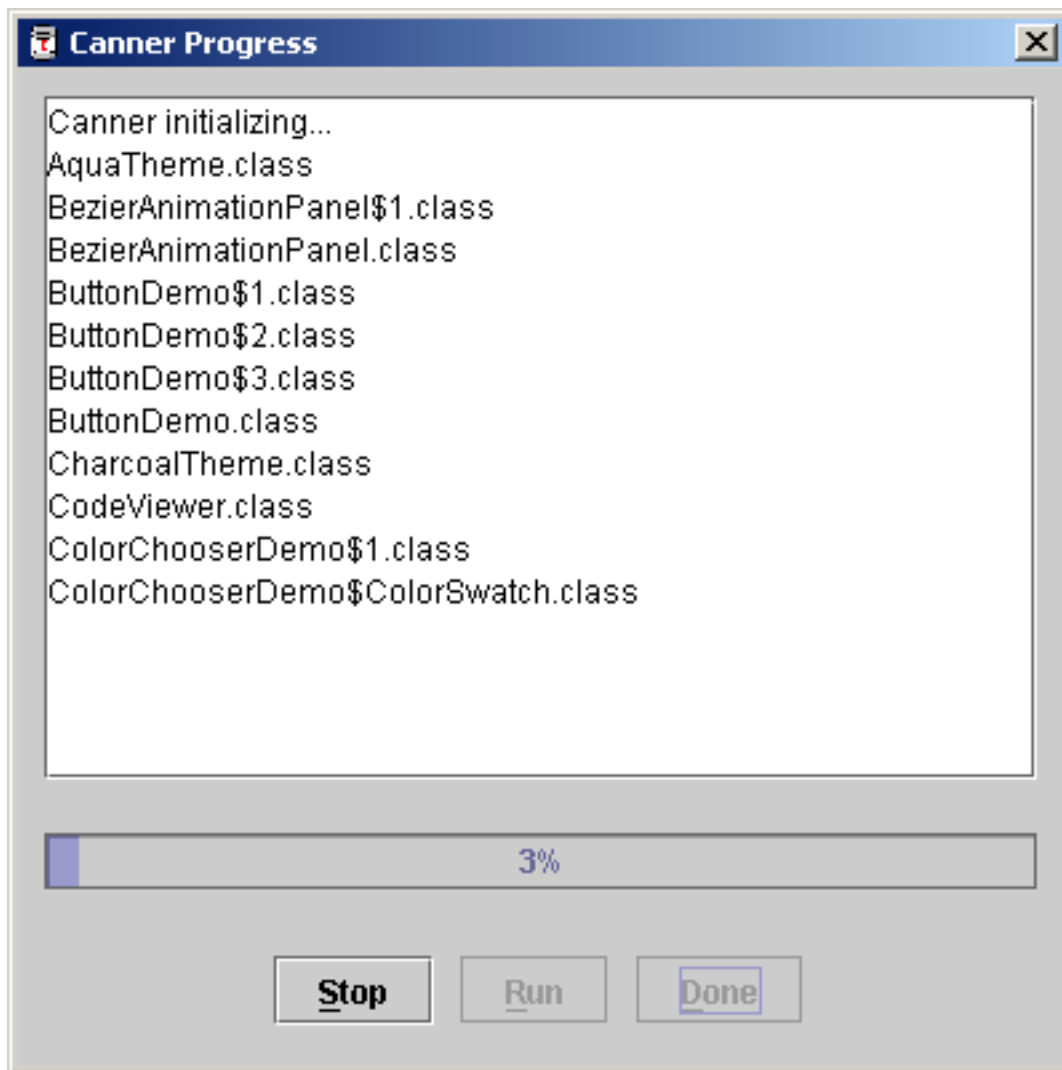
2.6. Encryption Key

Click on the "Encryption Options" tab. Choose the option "Generate a new encryption key". This option will create a new, random key to encrypt your application's classes.

2.7. Start the Canning Process

Under the "File" menu, choose the option "Can application...". This will start the process of creating your canned application. While the application is being processed, a progress dialog will display:

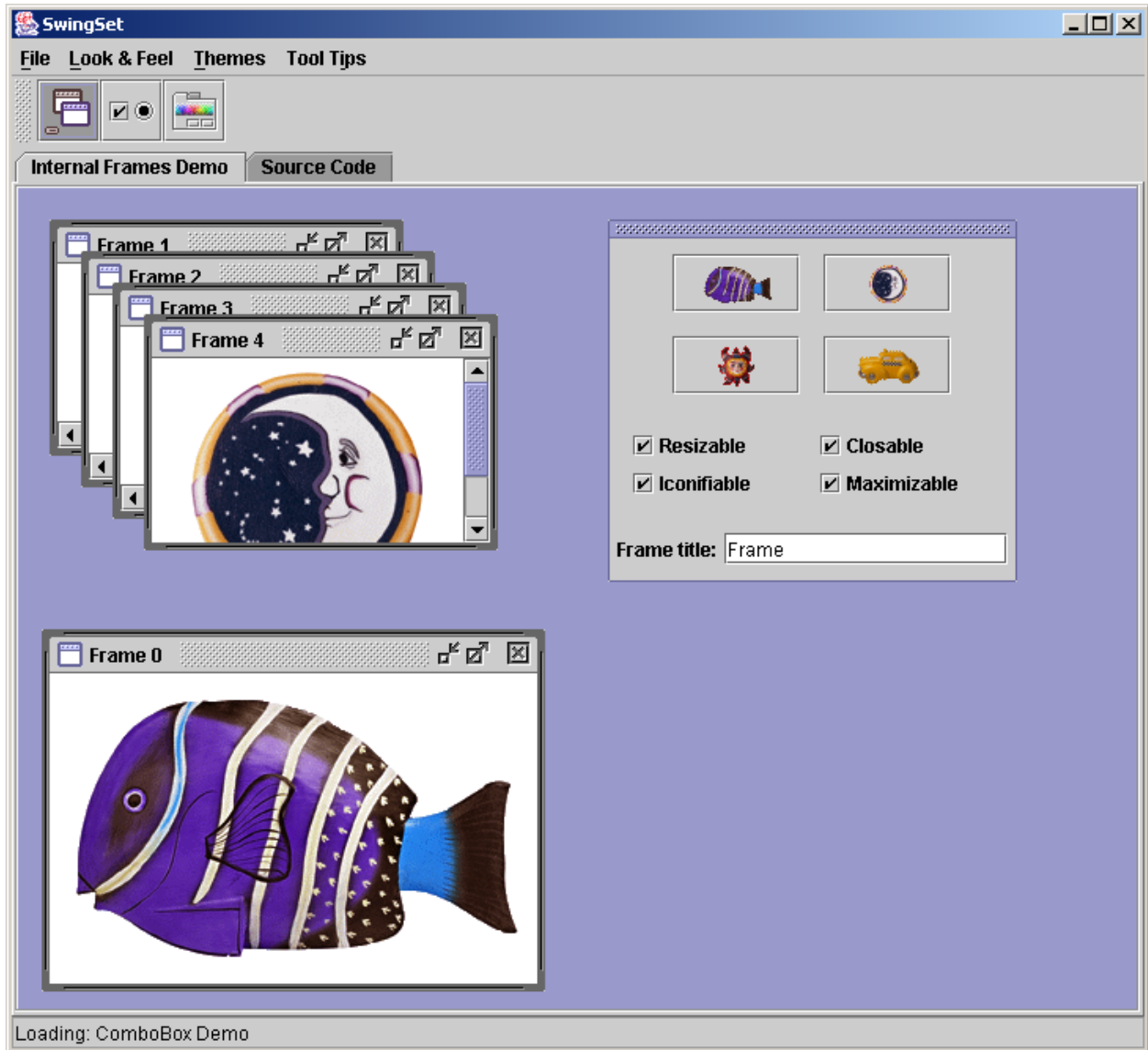
Figure 2.2. Canner Progress



2.8. Run the Application

When the canning process is finished, press the "Run" button. The output file (`SwingSet2.exe`) will be launched, and in a few seconds you should see the SwingSet demo window:

Figure 2.3. SwingSet Window



Chapter 3. Canner Configuration

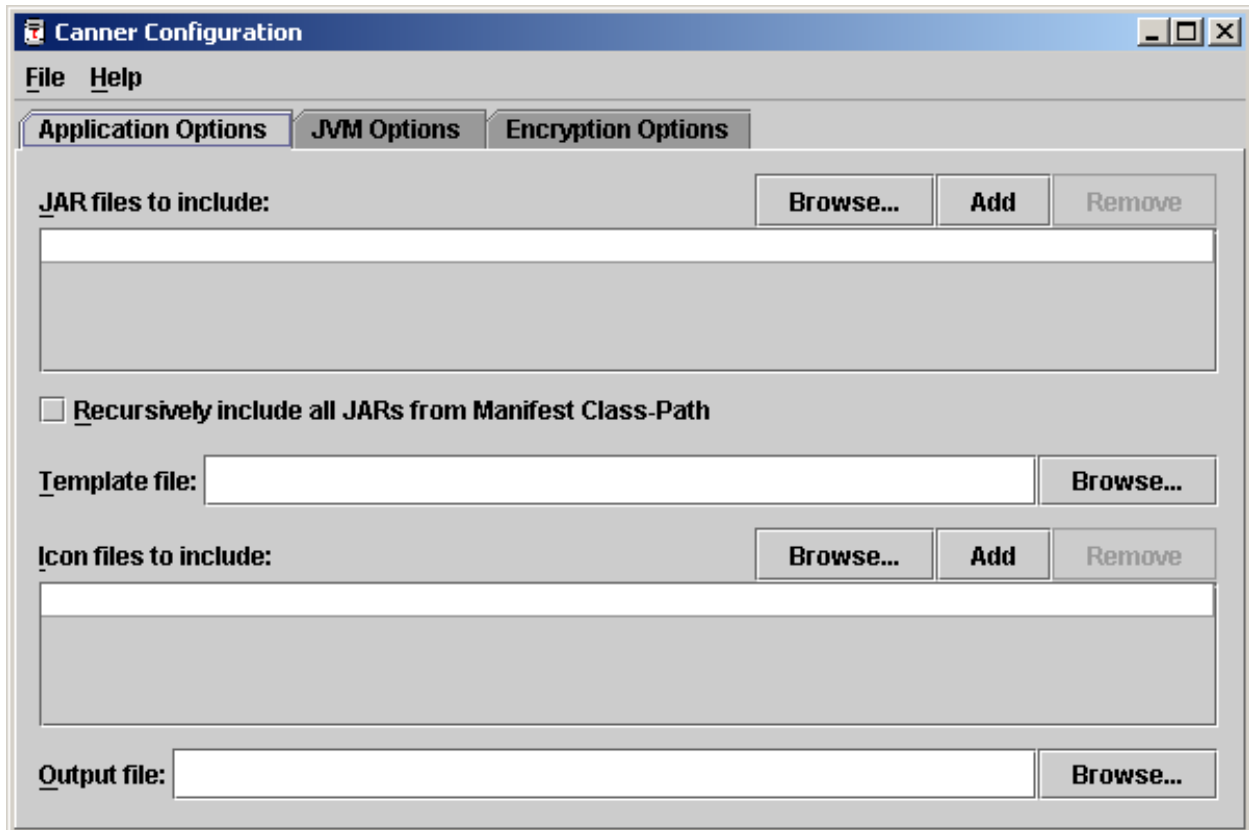
3.1. Canner Options

Now that you have seen how simple it is to can an application, we can examine the full set of Canner options. This document explains the options in the order they are displayed in the GUI. If you wish to automate the canning process, there are several other methods for specifying these options: from the command line, via an XML configuration file, or with an Ant Task. These other mechanisms are detailed later in this document.

3.2. Application Options

The *Application Options* panel controls the basic settings of what is included in your canned application, and where the application should be saved.

Figure 3.1. Application Options



JAR Files to include (required)

This list should encompass all the JAR files containing code and resources that should be encrypted and packaged into your output file. It is not required that all of your code be included here; a canned

application can still load from external JARs if necessary (see JVM Options, in the next section, for more details). For instance, license restrictions may prohibit you from repackaging JDBC drivers or other third-party libraries. Of course, these external JARs will still need to be present when you distribute your application.

Note that all of your canned classes must reside in JAR files; "loose" or un-jarred classfiles cannot be included in your canned application.

Recursively include all JARs from Manifest Class-Path

If this option is enabled, Canner will examine the Class-Path manifest attribute of all included JAR files and include all referenced JAR files in the canned output. This process is recursive; for example, if `a.jar` is the only JAR file explicitly included in the configuration, and `a.jar` references `b.jar`, and `b.jar` references `c.jar`, then `a.jar`, `b.jar`, and `c.jar` will all be canned together.

Template file (required)

This property must be the name of a template file supplied by Cinnabar Systems. The template file contains the native framework code needed to load the JRE and decrypt your classes as needed. Canner combines it with your application's JAR files to produce the output file.

Two template files are included with Canner, `template.can` and `templatew.can`. `template.can` should be used with applications that interact with the command line, and `templatew.can` should be used with GUI-only applications. These templates correspond to the **java** and **javaw** commands included with your JRE.

Check back with Cinnabar Systems for other template files which will provide additional functionality.

Icon Files to include (optional)

With this option, you may bundle one or more icon (`.ico`) files in your application's resource section. The first icon in the list will be the default icon for your application; your users can select from the others using Windows Explorer.

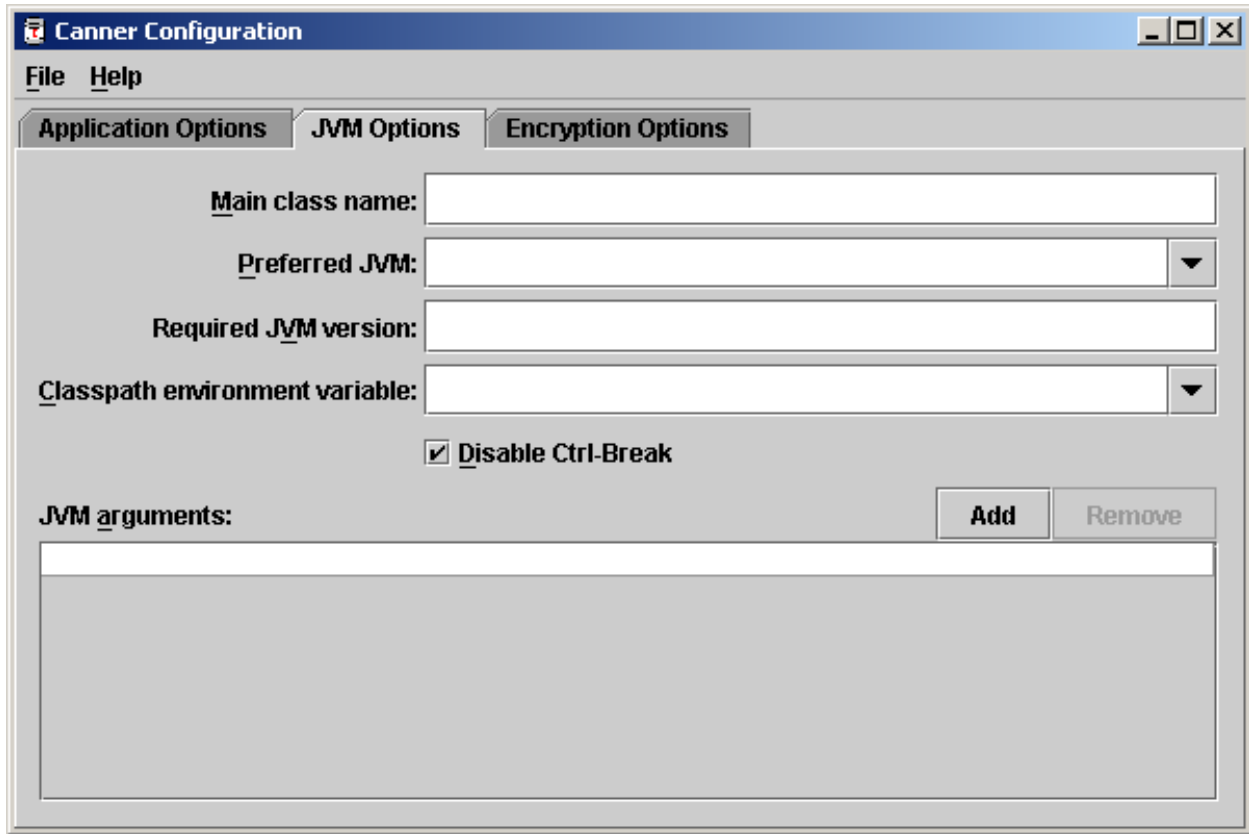
Output file (required)

This property is the name of the executable (`.exe`) file which Canner should generate.

3.3. JVM Options

The *JVM Options* panel controls how your application's Java Virtual Machine is created and invoked.

Figure 3.2. JVM Options Panel



Main class name (optional)

This property must be the fully-qualified name of a public class which has a `main()` method (e.g. `com.mypackage.MainClass`). If one of the included JAR files has a `Main-Class` manifest property, this setting may be omitted, and the first included JAR file which contains a `Main-Class` specification will determine the name of the main class.

If you do specify a class name here, it will override any `Main-Class` manifest entries in the included JAR files.

Preferred JVM (optional)

This property allows you to select which type of JVM should be loaded; available JVM types differ with the JRE release. This option provides the same functionality as, for example, the `-classic` or `-server` command line options in certain versions of the JRE. If the requested type is available, it will always be used. If it is not available, the canned application will attempt to load a fallback JVM based on the following table:

Table 3.1. JVM Fallback Types

| Requested JVM Name | Fallback #1 | Fallback #2 | Fallback #3 |
|--------------------|-------------|-------------|-------------|
| classic | client | hotspot | (none) |
| hotspot | client | classic | (none) |

| Requested JVM Name | Fallback #1 | Fallback #2 | Fallback #3 |
|----------------------------|----------------------|---------------------------|-------------|
| client | classic | hotspot | (none) |
| server | hotspot | classic | (none) |
| (other, or none specified) | client (1.4 default) | classic (1.2/1.3 default) | hotspot |

Most applications will want to leave this field empty, indicating to Canner that the default JVM should be used.

Required JVM version (optional)

If this property is present, it specifies a minimum JRE version number that must be present in order for the application to be successfully launched. For example, if this property is set to "1.3.1", then a 1.3.1 or newer JRE must be present for the canned application to run. If an older JRE is found, the canned application will exit with an error message.

Classpath environment variable (optional)

If this property is present, it specifies the name of an environment variable to use for the application's runtime classpath. The standard JRE uses the CLASSPATH environment variable for this purpose, but you can set it to a custom value to avoid conflicts with other Java applications that might have their own CLASSPATH settings.

Most applications will not need to set this property at all, as they will include all necessary JAR files to begin with and not need to load any external resources. Note that even if you specify an environment variable to use, canned applications will always attempt to load encrypted classes and resources first, even if the environment variable points to valid classfiles with the same name as your canned classes.

If you leave this setting blank, no external classes or resources will ever be loaded when the canned application runs, even if a CLASSPATH environment variable exists.

Disable Ctrl-Break

If this option is enabled, the CTRL-BREAK key will be ignored when your application is running. Normally, pressing CTRL-BREAK will cause the JRE to output a dump of all running threads along with their current stack backtraces, including class and method names. Disabling this ability will help limit the amount of information available to an attacker.

JVM arguments (optional)

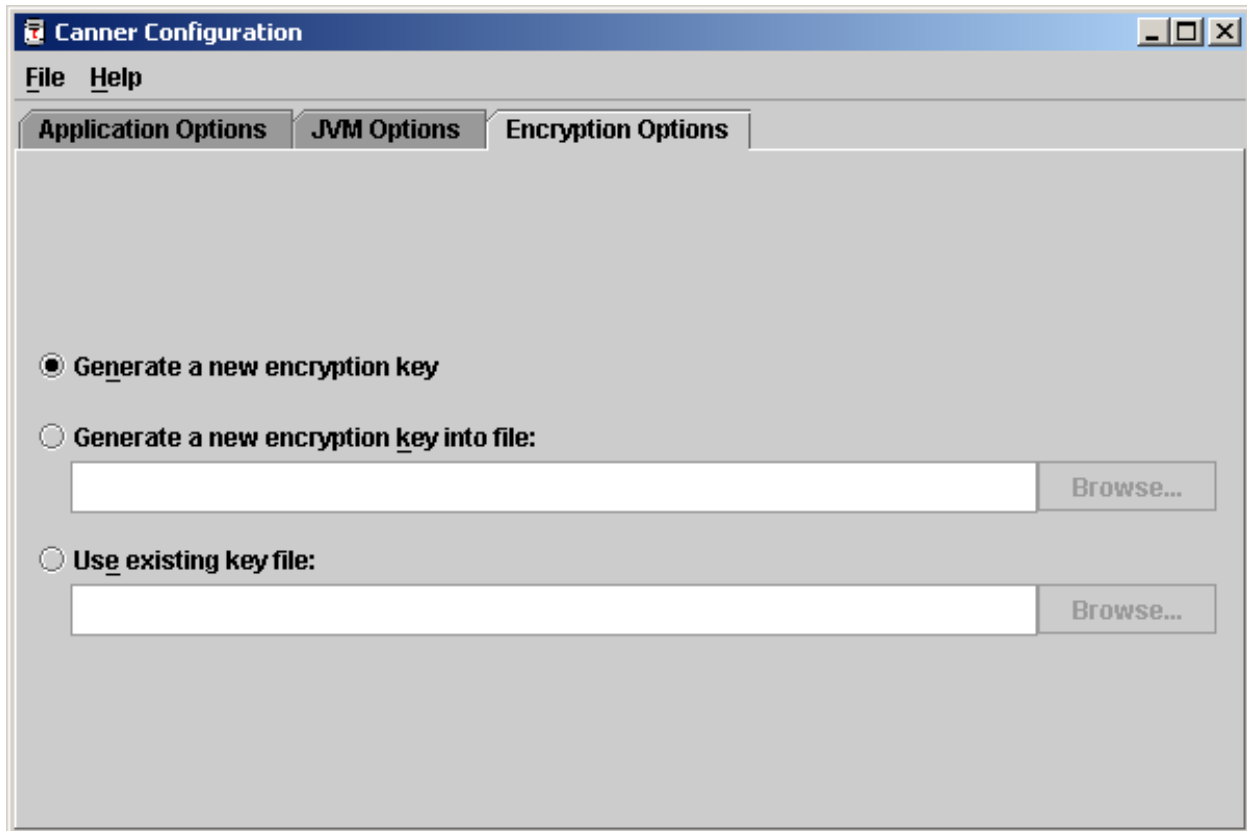
This option allows you to add any extra JRE options you might need to successfully launch your application. You will most likely need to use this option if your application expects system properties to be set with the -D option.

Note that the standard JVM options `-classpath` and `-cp` are not supported here. You will find that in most cases you will not need these options, since you will usually include all necessary JARs in your "canned" application. If you do find it necessary to specify a runtime classpath, however, you can use the option `-Djava.class.path=<classpath spec>` instead.

3.4. Encryption Options

The *Encryption Options* panel allows you to control how your application's classes and resources are encrypted. You can choose to have Canner generate a new encryption key, or use an existing key. You can vary your encryption keys at levels of granularity ranging from a single encryption key for all your products, to a unique encryption key for each product or even each executable. Cinnabar Systems recommends that you use a different encryption key for each of your canned products.

Figure 3.3. Encryption Options Panel



Generate a new encryption key

If this option is selected, Canner will generate a random key each time the application is canned.

Generate a new encryption key into file

If this option is selected, Canner will generate a random key when the application is canned, and save the key data into the specified keyfile.

Use existing key file

If this option is selected, Canner will not generate a new key; instead, it will load the key from the specified file. The keyfile must exist in order for the canning process to complete successfully. To generate

a keyfile, you can use the "Generate a new encryption key into file" option first.

Chapter 4. Running Canner

4.1. Command Line Operation

Since a GUI is not convenient to use from within an automated build system, Canner can also be run from the command line. It can be configured either by command line options, or via an XML configuration file.

4.2. Command Line Options

Canner supports the following options. See the previous section for a more detailed description of the meaning of each one.

Table 4.1. Canner Command Line Options

| Option | Meaning | Example |
|--|--|--------------------------------------|
| -? | Outputs a summary of allowed options. | -? |
| -u | Launches the Canner configuration GUI. | -u |
| -t <template file> | Specifies the name of the template file to use. | -t template.can |
| -o <output file> | Specifies the name of the canned output file. | -o application.exe |
| -r | Specifies that Canner should recursively include all JARs in the manifest Class-Path. | -r |
| -i <icon file name> | Specifies the name of an icon file to include. May be used multiple times if more than one icon file should be included. | -i icon1.ico -i icon2.ico |
| -m <main class name> | Specifies the fully-qualified name of the application's main class. | -m com.mypackage.Main |
| -v <JVM name> | Specifies the name of the JVM to load when the application is launched. | -v server |
| -e <minimum JVM version> | Specifies the minimum version number of the JRE that must be present when the application is launched. | -e 1.3.1 |
| -s <classpath environment variable name> | Specifies the name of the environment variable to use when loading classes at runtime. | -s MYCLASSPATH |
| -b | If present, CTRL-BREAK will be enabled in the output executable. CTRL-BREAK is disabled by default. | -b |
| -j <JVM argument> | Specifies an option to pass to the JVM when the application is launched. May be used multiple times if more than one option is needed. | -j -Dvar1=value1 -j -Dvar2=value2 |
| -n | Generates a new, random encryption key. The new key is saved to the file specified by the -k option, if given. | -n |

| Option | Meaning | Example |
|------------------|---|---------------------|
| -k <key file> | If the -n option is given, the newly-generated encryption key is saved to the specified file. If the -n option is not given, an encryption key is loaded from the specified file. | -k myapp.key |
| -c <config file> | Specifies the name of an XML configuration file to load Canner options from. The XML format is described later in this document. | -c cannerconfig.xml |

4.3. Command Line Parameters

Any JAR files to be canned should be added after the list of options. For example:

```
canner -t template.can -o application.exe -e 1.4
      main.jar lib1.jar lib2.jar
```

This example creates the output file `application.exe`, building it from the JAR files `main.jar`, `lib1.jar`, `lib2.jar`, and the template file `template.can`. The resulting application will require Java version 1.4 or better in order to run.

4.4. Configuration via XML

Canner options can also be specified with a simple XML configuration file. Although the syntax of this file is described below, you don't need to craft your own configuration by hand. Instead, you can use the GUI to set the options you want, and then save those settings as XML using the **File/Save** menu option. The resulting XML file can then be used over and over again with the command line `-c` parameter or the Ant task `configFile` attribute. For example:

```
canner -c myconfig.xml -o myoutput.exe
```

When you specify a configuration file via the `-c` parameter or the `configFile` attribute, any other configuration settings that you specify take precedence over those in the configuration file. For the example given above, the `-o myoutput.exe` setting would take precedence over any `OutputFile` element in `myconfig.xml`.

The CannerConfiguration element

The configuration file must have a top-level element called `CannerConfiguration`. All other elements must be nested inside it. For example:

```
<CannerConfiguration>
  <!-- other elements can be nested here -->
</CannerConfiguration>
```

The TemplateFile element

The `TemplateFile` element specifies the name of the Canner template file to use. It must have one attribute, called `value`, whose value is the path to the template file. For example:

```
<CannerConfiguration>
  <TemplateFile value="template.can"/>
</CannerConfiguration>
```

The OutputFile element

The `OutputFile` element specifies the name of the executable file that Canner will create. It must have one attribute, called `value`, whose value is the path to the output file. For example:

```
<CannerConfiguration>
  <OutputFile value="application.exe"/>
</CannerConfiguration>
```

The JarFile element

The `JarFile` element specifies the name of a JAR file to include in the canned application. It must have one attribute, called `value`, whose value is the path to the JAR file to include. This element may occur multiple times in the configuration, in order to include multiple JAR files. For example:

```
<CannerConfiguration>
  <JarFile value="main.jar"/>
  <JarFile value="lib1.jar"/>
  <JarFile value="lib2.jar"/>
</CannerConfiguration>
```

The RecurseJars element

The `RecurseJars` element specifies whether the included JAR files should have their manifest Class-Paths scanned recursively for additional JARs to include. It must have one attribute, called `value`, whose value must be either `true` or `false`. If this element is omitted, the option will default to `false`. For example:

```
<CannerConfiguration>
  <RecurseJars value="true"/>
</CannerConfiguration>
```

The IconFile element

The `IconFile` element specifies the name of an icon (`.ico`) file to include. It must have one attribute,

called `value`, whose value is the path to the icon file to include. This element may occur multiple times in the configuration, in order to include multiple icon files. For example:

```
<CannerConfiguration>
  <IconFile value="primary.ico"/>
  <IconFile value="secondary.ico"/>
</CannerConfiguration>
```

The `MainClass` element

The `MainClass` element specifies the name of the application's main class. It must have one attribute, called `value`, whose value is the fully-qualified name of a class that has a `main()` method. This element may be omitted if one of the included JARs has a `Main-Class` attribute in its manifest. For example:

```
<CannerConfiguration>
  <MainClass value="com.mypackage.MainClass"/>
</CannerConfiguration>
```

The `JVMName` element

The `JVMName` element specifies the name of a specific JVM to load. It must have one attribute, called `value`, whose value is the name of the JVM. If this element is omitted the default JVM type (normally `client` or `classic`, depending on the JRE version) will be used. For example:

```
<CannerConfiguration>
  <JVMName value="hotspot"/>
</CannerConfiguration>
```

The `MinimumJREVersion` element

The `MinimumJREVersion` element specifies the minimum version number of the JRE which must be present for the canned application to start. It must have one attribute, called `value`, whose value is the version number of the JRE to check against. If this element is omitted no version number check will be done. For example:

```
<CannerConfiguration>
  <MinimumJREVersion value="1.3.1"/>
</CannerConfiguration>
```

The `ClasspathVariable` element

The `ClasspathVariable` element specifies the name of an environment variable to use to allow additional, external classes to be loaded when the canned application is run. It must have one attribute, called `value`, whose value is the name of the environment variable to use. If this element is omitted no additional classes will be loaded, even if a `CLASSPATH` environment variable is set. For example:

```
<CannerConfiguration>
  <ClasspathVariable value="MYCLASSPATH"/>
</CannerConfiguration>
```

The AllowCtrlBreak element

The AllowCtrlBreak element specifies whether the canned application should respond to the CTRL-BREAK keystroke. Normally, pressing CTRL-BREAK will cause the JVM to output a dump of all running threads along with their current stack backtraces, including class and method names. Disabling this ability will help limit the amount of information available to an attacker. The AllowCtrlBreak element must have one attribute, called value, whose value must be either true or false. If this element is omitted, the option will default to false (i.e. CTRL-BREAK will be disabled). For example:

```
<CannerConfiguration>
  <AllowCtrlBreak value="true"/>
</CannerConfiguration>
```

The JVMArgument element

The JVMArgument element specifies an argument to be passed to the JVM when the canned application is started. It must have one attribute, called value, whose value is the argument to pass to the JVM. This element may appear multiple times if more than one argument is needed. For example:

```
<CannerConfiguration>
  <JVMArgument value="-Dvar1=value1"/>
  <JVMArgument value="-Dvar2=value2"/>
</CannerConfiguration>
```

The GenerateNewKey element

The GenerateNewKey element specifies whether Canner should create a new, random encryption key. It must have one attribute, called value, whose value must be either true or false. If this element is omitted, the option will default to false. For example:

```
<CannerConfiguration>
  <GenerateNewKey value="true"/>
</CannerConfiguration>
```

The EncryptionKeyFile element

The EncryptionKeyFile element specifies the name of a key file. It must have one attribute, called value, whose value must be the path to a key file. If the GenerateNewKey element is present and has the value true, then the newly-generated key is written to the file specified by this element. If the GenerateNewKey element is not present, or is present with the value false, then an encryption key is

read from the file specified by this element. For example:

```
<CannerConfiguration>  
  <EncryptionKeyFile value="mykey.key"/>  
</CannerConfiguration>
```

Chapter 5. Ant Integration

5.1. Ant Task

If you use Apache Ant in your build process, you can easily integrate Canner into your build. The file `<installation directory>/ant/canner-ant.jar` contains an Ant Task which can be used to invoke Canner from within Ant.

5.2. Defining the Task

Before you can use the Canner task, it must be explicitly defined in the Ant build file. To define the tasks, use the following declaration:

```
<taskdef
  resource="com/cinnabarsystems/canner/ant/taskdefs"
  classpath="{canner.installpath}/ant/canner-ant.jar"/>
```

Adjust the definition of the property `canner.installpath` to the appropriate directory for your build environment.

5.3. Invoking the Canner Task

To invoke Canner, use the **canner** task. Your application's JAR files can be specified with a nested `<fileset>` element. Additionally, the task supports the following attributes:

| Attribute Name | Definition |
|--------------------------------|---|
| <code>outputExecutable</code> | The name of the executable file which Canner should create. |
| <code>recurseJars</code> | Set to <code>true</code> to recursively include other JARs referenced by the manifest <code>Class-Path</code> attribute; defaults to <code>false</code> . |
| <code>templateFile</code> | The name of the Canner template file to use. |
| <code>iconFiles</code> | Comma-separated list of icon (<code>.ico</code>) files to include. |
| <code>mainClass</code> | Fully qualified name of the application's main class; can be omitted if one of the included JARs has a <code>Main-Class</code> attribute in its manifest. |
| <code>jvmName</code> | Name of the JVM type to use (e.g. "server" or "hotspot"); if omitted, the default JVM type will be used. |
| <code>minimumJREVersion</code> | Specifies the minimum version number of a JRE which must be present for the canned application to start. |

| Attribute Name | Definition |
|--------------------------------|---|
| <code>classpathVariable</code> | Name of an environment variable to use to allow the canned application to dynamically load classes; if omitted, classes will never be loaded from external sources. |
| <code>allowCtrlBreak</code> | Set to <code>true</code> to enable normal CTRL-BREAK processing; defaults to <code>false</code> , which prevents CTRL-BREAK from causing a full thread and stack dump. |
| <code>jvmArguments</code> | Comma-separated list of arguments to pass through to the JVM. |
| <code>generateNewKey</code> | Set to <code>true</code> to specify that Canner should generate a new random encryption key; defaults to <code>false</code> . |
| <code>keyFile</code> | Specifies the name of a file which key data should either be read from (if <code>generateNewKey</code> is <code>false</code>) or written to (if <code>generateNewKey</code> is <code>true</code>) |
| <code>configFile</code> | Specifies the name of an XML configuration file to load Canner options from. The XML format is described earlier in this document. |
| <code>cannerExecutable</code> | Specifies the full path to the <code>canner.exe</code> file. If omitted, <code>canner.exe</code> must be in the system PATH. |

Example

The following example invokes Canner on the files `main.jar`, `lib1.jar`, and `lib2.jar` to produce the executable file `application.exe`. The template file is `template.can` and the main class is `com.mypackage.Main`. The JVM will be invoked with two arguments, `-Dvar1=value1` and `-Dvar2=value2`. A new encryption key will be generated, but not saved to any file.

```
<canner
  outputExecutable="application.exe"
  templateFile="template.can"
  mainClass="com.mypackage.Main"
  jvmArguments="-Dvar1=value1,-Dvar2=value2"
  generateNewKey="true">
  <fileset>
    <include name="main.jar"/>
    <include name="lib1.jar"/>
    <include name="lib2.jar"/>
  </fileset>
</canner>
```