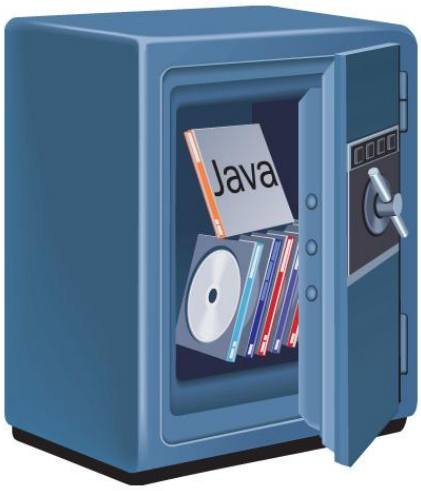
jProductivity LLC

Protect your investments with Protection!



Protection!™

Licensing Toolkit

v4.8

Developers Guide

http://www.jproductivity.com

Revision
271 - 10/24/17

Applicability
This document applies to Protection! v4.8 software.

Contents

Chapter 1

Protection! Licensing Toolkit Concepts

The following topics outline major concepts of Protection! Licensing Toolkit:

1.1 Key Concepts

Protection! is a Licensing Toolkit. The key concept of Protection! is to verify the presence of a valid license required by the product as well as to prevent license tampering. Protection! Licensing Toolkit is a set of Java classes that need to be embedded into the custom application in order to allow only authorized use of the application. Authorized use of a custom application is based on the supplied license. Protection! Licensing Toolkit provides application developers with full control over the license checking and validation mechanism. Protection! classes provide all of the necessary functionality to allow license discovery, license reading and validation, as well as several other utility functions such as the ability to implement and further integrity check for desired classes/resources in order to avoid malicious application patching.

1.1.1 License File

License File represents a way to transfer the appropriate rights to the end-user, allowing the end-user to use the protected application. Protection! License File is implemented as an encoded file that contains information about the product, version, license type, etc. Protection! License File can either be bundled with the application and loaded as a resource, or be located anywhere in the file system. The license File, generated by the Protection! Control Center, is a strongly encrypted file ensuring that virtually no one can break its protection by the simple re-generation of the license file. To meet this goal, a non-symmetric cipher algorithm is used where keys used to encrypt and decrypt the license are different. The key size is 128 bits, which provides enough encryption strength while keeping license file size in acceptable bounds. The Encryption key pair is unique to each particular product, thereby eliminating the ability to utilize the license file generated for a different product. This approach makes it impossible to restore the private key by having knowledge of the public key and vice versa.

1.1.2 Protection! Control Center

Protection! Control Center offers a central location for Product Management, License Generation, License Maintenance, and Creation of the corresponding Code Snippets as ready-to-use Java implementation files, License deployment, and Protection! Web Services application configuration. Protection! Control Center allows the developer to manage an unlimited number of products, product's editions and product's feature sets per each individual product.

1.1.3 Secret Storage

Secret Storage provides a way to persistently and secretly store various and important information about the application. Secret Storage can also be used to store various actions performed by the end-user. The following information is currently stored in Secret Storage:

- First encountered expiration time (to prevent unauthorized use by getting a newer evaluation licenses).
- A flag that specifies whether the user has accepted the terms of the license agreement (this could be used to suppress showing the license agreement dialog at application startup).

Protection! provides simple file and Preferences API based secret storage implementations and with the help of Protection! Control Center the developer is able to generate the appropriate Code Snippets to utilize secret storages. Because Protection! is a licensing framework application developers are free to provide their own implementation of Secret Storage e.g. Windows Registry based. Protection! supports working with several Secret Storage files at the same time in order to provide greater redundancy to protect the data.

1.2 Protection! Process

Protection! Process consists of the following steps:

1.2.1 Reading a License

Protection!'s license support subsystem is responsible for locating, reading and validating the license, providing access to the license file and firing notifications during the license load process (e.g. when the license is valid, missing or invalid).
License reading is performed using the following simple steps:

1. Locating the License. If it is determined that the required license is missing, the developer can call the Resolver mechanism to attempt to resolve the missing license issue. If the Resolver does provide a valid license, then the process of locating the license is repeated.
2. Reading and Decoding the License. If the license is corrupted, the developer can call the Resolver mechanism to attempt to fix the corrupted license issue. If the Resolver does provide a valid license, then the process of license Reading and Decoding is repeated from the start.

1.2.2 Checking License

The following steps are only performed if the license was discovered and read successfully:

1. Verify that the license was issued for the correct product. The default implementation assumes that the license is valid, only if the product's identifier is equal to the product's identifier contained within the license file.
2. Check that the product's major and minor versions contained within the license file are valid for the product. The default implementation assumes that the license is valid if information about the product's major version, which is stored within the license, is equal to the actual product's major version. Minor version is ignored within the default implementation.
3. Check the product's features embedded within the license. The default implementation always assumes that any feature set is valid. However, application developers are always able to add their own checking and verifications.
4. Check that the license has not expired. If the license is an evaluation license, additional checking is performed to verify that the license did not expire earlier using the value obtained from the Secret Storage. If it is determined that the license is expired, the developer can call the Resolver mechanism in order to attempt to resolve an expired license issue. If the Resolver does provide a valid license, then the process is repeated starting from the step of Reading a License.
5. Check that the license is activated. If the license is not activated, the developer can call the Resolver mechanism in order to try to activate the license. If the Resolver

successfully activates the license, then the process is repeated starting from the step of Reading a License.
6. Check that the user has accepted the terms of the License Agreement.
7. Check the number of running copies over the network (Professional Edition only).
8. Save the earliest expiration time for the evaluation license and the License Agreement acceptance flag to the Secret Storage.

During license checking and validation, Protection! Licensing Toolkit provides detailed feedback about the process status and its results back to the application. This feedback is provided by firing an appropriate set of events. It is the responsibility of the application developer to decide which action should be taken in response to a particular Protection! event. The default implementation assumes no actions for any such events provided by Protection! Licensing Toolkit. For example: when it is determined by Protection! that the license file is invalid, the application can either exit or the application could disable all or part of its functionality. Such a decision is the responsibility of the application developer (there is no default behavior provided by Protection! Licensing Toolkit). In addition to the event based analysis/action, it is always possible to get the current status of the license at any time during an application's run to see whether the license is valid.

While the default implementation of the license checking mechanism provides functionality that would be enough for the majority of the applications, developers can easily override/extend any license reading/validating steps stated above to get the desired results.

1.2.3 Checking Integrity

The Integrity verification subsystem allows for checking and validation that the designated key classes, resources or files of the product have not been changed. This type of check is done by comparing the digest of those classes/resources with the original value stored somewhere in the product code, resources or files. The Integrity subsystem significantly increases the time and effort needed to diagnose and locate specific parts of the protection system in order to attempt to break it.

1.2.4 Conclusion

In addition to the core Protection! responsibilities such as license discovery, license reading and validation, Protection! Licensing Toolkit provides developers with various helper classes and with default implementations of the listeners. Default implementation of the listeners provides users with feedback information regarding the Protection! license checking process and its results. Such helper classes and listeners allow the following functionalities:

- Ability to pop-up a message window and/or Licensing Assistant Wizard if the license file was not found, corrupted, invalid or expired
- Ability to prompt the end-user to accept the terms of the license agreement using the License Acceptance dialog
- Ability to show the About Dialog with license information, application name and version, feedback information, a hyperlink to the developer's website, License type, Issue and Expiration Dates, License Number, Licensee information and a hyperlink to view the license agreement.
- Ability to perform an integrity check for desired classes/resources in order to avoid malicious application patching.

Chapter 2

Adding Protection! Support to the Java Applications

To add Protection! support to Java applications the following major steps should be completed:
1. Creation and initialization of Protection! frameworks and its objects.
2. Providing code for handling Protection! events and license checking results.
3. Adding appropriate calls to the license and integrity checking methods.

2.1 Protection! Initialization

Protection! framework and its objects should be properly created and initialized before first use. Such initialization can be done either in the source code or via loading configuration data from the previously generated product's Launcher resource.

2.1.1 Source Code

Initialization via source code is done by implementing all required API calls into the application's source code. Initial implementation of the support class (e.g. `DemoCalcProtectionSupport`) can be based on one of the code snippets provided by the Protection! Control Center; any further changes can be done manually. The following types of generated code snippets are available:

a) *GUI* – provides a template for implementation that can be used to add Protection! support for the GUI-based applications. This implementation includes various Assistants to simplify licensing tasks, the License About Dialog, standard error messages etc.
b) *Headless* - provides a template for implementation that can be used to add Protection! support for the server or console applications.

An instance of support class can easily be created using one of its constructors (for the code snippets based implementation):

```
DemoCalcProtectionSupport support = new DemoCalcProtectionSupport(owner);
```

2.1.2 Launcher

Initialization via generated product's Launcher is done by loading configuration data (Launcher) from the application resource or a file at the runtime. Initial implementation can be based on one of the code snippets provided by the Protection! Control Center; any further changes can be done manually. The following types of generated code snippets are available:

c) *GUI (Launcher)* – provides a template for implementation that can be used to add Protection! support for the GUI-based applications. This implementation includes various Assistants to simplify licensing tasks, the License About Dialog, standard error messages etc.
d) *Headless (Launcher)* - provides a template for implementation which can be used to add Protection! support for the server or console applications.

The Launcher resource can be generated using the Protection! Control Center | Products Screen via the Build | Build Launcher menu item or using the Protection Builder Utility. The Generated Launcher must be bundled and distributed with the application either as an application resource or as a file.

An instance of support class can be created using one of its constructors and then initialized by providing the name of the Launcher resource (for the code snippets based implementation):

```
DemoCalcProtectionSupport support = new DemoCalcProtectionSupport(owner);
support.init(this.getClass(), "DemoCalc.launcher");
```

Using Launcher makes it easy to change and tune the configuration as no source code modifications are required.

Note

It is strongly recommended to provide an owner (if any) attribute to constructors of GUI implementations. This value will be used as owner for all of the dialogs shown during the licensing process. If owner attribute is not specified it may cause incorrect handling of windows Z-order while user switches between different applications.

Tip

In some cases (SWT or console applications) it is not possible to provide a valid owner because there is no frame instance in the components root. In such cases it is recommended to use a `dev.gui.FakeOwnerFrame` instance as owner.

2.2 Handling Protection! Events

A very important part of Protection! initialization is the listening events which are fired during license locating, reading and checking:

```
DefaultLicenseAdapter licenseListener = new DefaultLicenseAdapter(fOwner)
{
  public void featureChecked(LicenseHost aSource, License aLicense, String
                             aFeature, boolean isEnabled)
  {
    if (aFeature.equals("1")) // Basic
    {
      ((DemoCalc) fOwner).setBasicFunctionalityEnabled(isEnabled);
    }
    else if (aFeature.equals("2")) // Scientific
    {
      ((DemoCalc) fOwner).setScientificFunctionalityEnabled(isEnabled);
    }
  }
}
```

2.3 Calling License Check

After all initialization is done the call(s) of `checkLicense()` method should be added somewhere in the application code e.g. during application startup. Optionally the `checkLicense()` calls can be scheduled according to some sequence of events or time frequency (e.g. each hour) to ensure that the license is still OK.

Results of the license check can be used to limit some of the application functionality or can be a reason for application exit (depending on the nature of the application and business needs).

```
if (! support.checkLicense())
{
```

```
  System.out.println("License is not OK");
  System.exit(-1);
}
```

Chapter 3

Adding Protection! Support to C/C++ Applications

Though Protection! is primary intended to add protection and licensing support to Java applications, Protection! can also be integrated with C++ applications.

The following are the most common approaches to integrate Protection! and C/C++ that could be applied to any required platform.

Note

A valid JRE (Java Runtime Environment) is required to run Java code regardless which integration approach is used.

3.1 In-process Integration

In this case both Java and C/C++ code work in the same process. It can be done by one of two ways.

3.1.1 Calling Java from within C/C++

It can be done by in-process instantiation of the Java VM and by letting it execute some Java class responsible for licensing tasks. Results of licensing procedure can be returned back to the caller C/C++ code using any applicable mechanism e.g. system variables, temporary files, clipboard, communication via TCTP/IP etc. The flow would be as follows:

a) Start native C/C++ application.
b) Instantiate Java VM and execute Java code responsible for licensing.
c) Process licensing result if direct communication with Java code is selected (TCP/IP).
d) Quit Java VM.
e) Process licensing results if NO direct communication with Java code is selected.

Note

This approach requires writing platform specific C/C++ code to instantiate Java VM as it seems there is no uniform way to achieve this.

3.1.2 Calling C/C++ from within Java

It can be done by starting Java code as an application launcher and calling your C/C++ application's code via JNI. With this approach Java code would perform all of the licensing tasks, call appropriate C/C++ code to notify about licensing results and then will start C/C++ application (e.g. by calling the main() method). Flow would be as follows:

a) Start Java application.
b) Execute Java code responsible for licensing.
c) Notify C/C++ code about licensing results by calling some methods via JNI.
d) Start core C/C++ application code e.g. by calling main() method via JNI.
e) Quit Java VM when C/C++ main() quits.

3.2 Out-process Integration

In this case Java and C/C++ applications are independent of each other and therefore are run as different processes. When C/C++ need to execute licensing processes it simply starts the Java application responsible for the licensing tasks as a different process and communicates with such Java application remotely e.g. via TCP/IP. When licensing processes are completed, the Java application can quit and the C/C++ application can continue to run or quit according to the licensing results. The flow would be as follows:

a) Start native C/C++ application.
b) Start Java application responsible for licensing as a different process.
c) Communicate with the Java application remotely e.g. via TCP/IP to start licensing and obtain licensing results.
d) Quit Java application.
e) Process licensing results.
f) Continue to run with enabling or disabling some functionality or simply quit if licensing failed.

Chapter 4

License Types

Protection! provides support for several licenses types:
1. Trial:
   - `License.TYPE_EVALUATION` - Evaluation type
   - `License.TYPE_EXTENDED_EVALUATION` - Extended Evaluation type
2. Commercial:
   - `License.TYPE_COMMERCIAL` – Commercial type.

License of each particular type has its own application and behavior.

Trial licenses should be issued for prospective customers who require some time to discover and evaluate product functionality to decide whether they are ready to buy a commercial license for the product.

Commercial licenses should be issued for customers who already purchased/acquired rights to use the product for commercial purpose. It does not matter that both Trial and Commercial license types have the ability to specify some expiration criteria – license of either type should be issued correctly according to the status of the customer (e.g. prospective/evaluation or commercial).

The following functionality depends on license type:
1. Date based expiration.
2. Status of Product features (enabled/disabled) corresponding to an appropriate license state.

In general, licenses of any type are valid when current date is between license's issue and expiration dates (if specified). Trial licenses always have an expiration date as they are temporary by specification. Though there are several cases which can change rules how issue and expiration dates are handled:
1. ANY license type: Optional tracking and check of previous application shutdown date to avoid some tricks with system time.
2. EVALUATION type only: Tracking the date of the license's first use. It is intended to eliminate the ability to prolong the trial period simply by obtaining and applying new (fresh) evaluation licenses.
3. EVALUATION type only: Tracking the fact that the trial license has expired in the past. In this case no new Evaluation licenses can be applied as such licenses would still be considered as expired. If there is a need to allow customer to continue evaluation then a new **Extended** Evaluation license should be issued and applied instead of another Evaluation license.
4. EVALUATION type only: Ability to have a flexible expiration date. In this case the expiration date is recalculated based on the date of the license's first use and the duration of the license's evaluation period. This functionally is useful when the trial license is bundled with the application i.e. for distribution on CD.

To summarize:

1. Evaluation licenses CAN be bundled with the application or CAN be issued and applied by request.
2. Extended Evaluation licenses MUST be issued and applied by request to extend the customer's trial period.
3. Commercial licenses SHOULD be issued and applied by request.

Chapter 5

License Reading

Protection! life-cycle consists of two major steps: obtaining a license and checking the license. Instances of `LicenseReader` are responsible for licenses locating and reading.

Tip

It is recommended to use Code Snippets, which are generated by the Protection! Control Center application, not only to learn about Protection! fundamentals, but also to easily embed Protection! into custom applications.

5.1 License Reader

To get a license, an instance of the `LicenseReader` should be created and properly configured. It can either be done via `LicenseReader's` constructor or via factory method `ProtectionFactory.createLicenseReader()`. After `LicenseReader` is created and properly configured the call of License Reader's `getLicense()` method allows getting a license. License Reader caches an already read license, if any, and always returns the license without trying to read it again. If the license is still not read due to the number of reading attempts or due to the errors during reading, then License Reader continues to try to read the license during any further call of the `getLicense()` method. To force License Reader to re-read the cached license the `getLicense(true)` method call should be used.

To increase efforts required to break protection, none of the License Reader methods are showing any error output or throw any exceptions. It is possible to change this behavior for development and debugging purposes only by calling `setVerbose(true)` method.

Note

It is a sole responsibility of the application developer to remove any `setVerbose(true)` calls before production deployment of the protected application. Failure to do so could significantly compromise application security by allowing malicious user to obtain a wealth of information from License Reader methods output and/or its exceptions.

There are several operations that should be properly and successfully completed during the process of getting a license. These operations are: license locating, license reading and license decoding respectively.

5.2 Locating License

License Reader is capable of locating licenses deployed within the local file system and/or bundled with the application archive. This allows having various deployment options. For example it would be practical to bundle an evaluation license with the application archive to deploy the application on CD to evaluators. This approach allows users to start evaluating the application without a need to visit the vendor's site to register and to get an evaluation license. Later, if the user decides to buy a commercial license, the user simply copies a new license (obtained from the vendor) to the user's *HOME* folder (where user's *HOME* folder is used only to illustrate this example) and the

protected application will make use of the new license overriding the one bundled with the application.

The license file is explicitly identified by its name (e.g. *DemoCalc.key)*. The `setLicenseFileName(…)` and the `getLicenseFileName()` methods of the License Reader can be used to access the name of the license. When the license is located within the local file system it should be placed in the file named according to the license name (e.g. *DemoCalc.key)*. The same rule woks for the bundled licenses. It is impractical to give a license an excessively generic name (e.g. *license.key*) as it can easily conflict with other applications protected by Protection! Licensing Toolkit. As a result, the license name should reflect the product's name that is more or less unique in most of cases. For example, for product *Demo Calculator* license name is *DemoCalc.key*.

By default, License Reader tries to find a license in the user's *HOME* (e.g. "*C:\Documents and Settings\<User Name>*" for Windows) folder first. If a license is not found then License Reader tries to get it from the root of the application archive. The default search sequence can be changed by calling of `setSearchLicenseInFile(false)` method of License Reader.

While the user's *HOME* folder seems to be a reasonably good location for a license file it is possible to specify another desired location of the license by calling `setLicenseFolder(…)` method to let License Reader know exactly where the license should be located. If the specified license folder denotes the relative path to the folder, License Reader treats it as a subfolder of the user's *HOME* folder. It is possible to resolve the relative license folder against the current folder of a running application by calling `setUserHomeRelative(false)` method. This allows, for example, storing the license in the sub folder of the application installation root folder.

Note

Actual location of the user's *HOME* folder could depend on how the application is launched. Windows example: if the application is launched as service the user's *HOME* folder would be *C:*; if the application is launched as the standalone the user's *HOME* folder would be *C:\Documents and Settings\Some User*.

The License file within the application archive can also be placed to some folder that differs from the root of the application archive. The `setLicenseResourceFolder(…)` allows specifying license location in such a case (e.g. `/com/jp/samples/protection/`).

Note

As soon as a license is found, License Reader immediately tries to read and decode the discovered license and even if it encounters problems during reading and decoding it never tries to read the license from the alternate location.

Tip

In some cases there would be a need to load licenses from some other sources (like database) rather than from file. This could be done by subclassing the License Reader and overriding the `getLicenseInputStream()` method. Its implementation should return proper input stream capable of reading licenses from sources such as database.

5.3 Reading and Decoding License

When the license file is located and it is determined that the license can be read, the License Reader reads and decodes the license. Asymmetric cipher is used to encode a license that requires Decryption Key to be specified. It is possible to specify Decryption Key by calling `setDecryptKeyBytes(…)` method and passing Decrypt Key as an array of bytes to the method as a parameter. Application developers should not be concerned with key generation functionality as Protection! Control Center generates and provides

both keys for license reading and writing. Code Snippet available in the Protection! Control Center's Products Screen provides key bytes as well as other code that would be required to embed Protection! into the custom applications.

Control Center generates a unique key pair combination for any particular product. Therefore it is impossible to read the license generated for a different product.

Note

During the creation of the product via the product copy mechanism (e.g. with help of system clipboard), Control Center assigns a new, unique key pair to the copied product, so even if some products look alike they would have completely different key pairs. If you need to move/copy a product to a different Product Storage use the *"File | Import Product"* to achieve it.

Protection! utilizes third-party RSA cipher implementation provided by Bouncy Castle JCE Provider. Though under Java6 Protection! also provides SUN's native RSA cipher implementation making it possible to get rid of Bouncy Castle. To do so:

1. Make sure that SUN's JCE security algorithm is used e.g. "RSA – SunJCE – 512".
2. Instruct Protection! not to use Bouncy Castle either by calling `SecurityProviderFactory.setUseBouncyCastleSecurityProvider(false)` in the application code or by specifying a System Property `com.jp.protection.security.useBouncyCastleSecurityProvider=false`.
3. Remove Bouncy Castle libraries (`bcprov-jdk14-138.jar`) from deployment of the application.

5.4 Listening License Reader Events

With the simplest implementation of the Protection! Licensing Toolkit there is no need to know about any errors that occurred during the license reading. It is enough to call `getLicense()` method, in order to obtain the license, and later, if `getLicense()` method returns the license, the application can just use it. While this approach can work well in some cases it is better to notify the user about errors, especially for GUI applications.

To do so Protection! uses the familiar event based model by notifying registered `LicenseReaderListener` implementation objects about events that occurred during license reading. This listener allows listening the following events:

- When the license is found and is ready to be read;
- When the license is missing;
- When the license is corrupted;
- When the license has been successfully read.

Tip

Since v4.8 it is possible to listen for two additional events by using extended `LicenseReaderListenerExt` listener interface: when the license has been updated and when the license has been removed. Those new methods can be typically called during updating of the license from Licensing Server or other remote service.

Note

All above events are post process and are for informational purposes only. Therefore it is impossible to write code in the event method's implementation to influence license reading.

There is no default implementation of `LicenseReaderListener` itself but Protection! offers two implementations of `LicenseListener` interface (It combines both `LicenseReaderListener` and `LicenseHostListener` interfaces):

- `LicenseAdapter` provides a default implementation that prints appropriate messages to standard output or error streams when it runs in verbose mode. If verbose mode is off, this implementation does nothing.
- `DefaultLicenseAdapter` provides a default implementation suitable for GUI applications as it shows appropriate message dialogs with description of error and vendor contact information. It is possible to create `DefaultLicenseAdapter` via its constructor or via factory method `ProtectionFactory.createLicenseListener(…)`.

5.5 Resolving License Reader Issues

As mentioned above, License Listener does not allow writing of any code to affect license reading and therefore it is not possible to use such listener to fix any issue(s) which could occur during the license reading. To meet this goal of fixing issues that happened during the license reading Protection! provides a simple, yet elegant way by using a special object (resolver) responsible for the resolving of such issues.

Each resolver should implement `LicenseReaderIssueResolver` interface and each License Reader can carry no more than one resolver. This interface introduces the following methods used by the License Reader when issues are discovered:

- `resolveLicenseMissing(…)` – called when License Reader is unable to locate a license;
- `resolveLicenseCorrupted(…)` – called when a license is corrupted.

Tip

Since v4.8 it is possible to implement extended `LicenseReaderIssueResolverExt` interface which provides a new method: `updateLicense(…)` – called to attempt updating license. This method will be called only if License Reader was able to locate and read a license.

If resolver's method is able to resolve an issue then it executes all appropriate actions and returns `true` to indicate that License Reader should start reading the license again. If resolver returns `false` indicating that it is unable to fix the issue then License Reader fires an appropriate event to the registered License Reader Listeners and stops the license reading process.

Actual implementation of resolver methods would perform the following actions:

1. Get a new/updated valid license by communicating with remote backend, Licensing Server or by prompting the user to specify the location of a valid license file etc.
2. Save obtained/updated license to the license file returned by License Reader's `getLicenseFile()` method.
3. Remove the license if it is not available anymore e.g. the user was removed from the allocation in Licensing Server.

Protection! provides two very powerful resolver implementations: GUI and Headless. Both implementations can deal with Protection! Backend or Licensing Server to obtain either evaluation or commercial licenses.

The call of `ProtectionFactory.createLicenseReaderIssueResolver(…)` can be used to create GUI resolver. This resolver shows Licensing Assistant Wizard that aids the user in getting a valid license. It provides the following options:

1. Ability to specify the location of a valid license file;
2. Ability to view information on how to obtain the license offline;
3. Ability to request an evaluation or extended evaluation license from Protection! Backend;
4. Ability to obtain a commercial license from Protection! Backend by providing the Serial Number;
5. Ability to obtain licenses from the Licensing Server.

The `HeadlessLicenseReaderIssueResolver` class provides implementation of headless license reader resolver. This implementation can be used directly in console or server applications as it does not expect any assistance from the user. It automatically tries to get the evaluation license if Serial Number is not specified. If Serial Number is provided, it tries to get a commercial license instead.

If ready to use implementations are not suited for a particular application, it is possible to easily implement any required functionality with the help of `IssueResolverSupport` class. `IssueResolverSupport` class provides all of the functionality required to build custom resolver implementations and hides all communication details with the Protection! backend.

5.6 Upgrading License

Usually the customer would utilize a trial license to learn and evaluate the application at first. Next he/she can purchase a commercial license to allow further use of the application. Or he/she can purchase an upgrade (e.g. from Standard to Professional edition). To simplify the license upgrade process Protection! provides ready to use functionality for GUI applications. The `upgradeLicense()` method of GUI code snippet (generated by the Protection! Control Center) starts the upgrading process by launching the Licensing Assistant Wizard. In this Wizard the user is able to specify the location of a new license, request an extended evaluation license or request a commercial license by Serial Number.

Note: it is the sole responsibility of the application developer to properly implement code that will handle license checking results, to apply new license type, edition or features set. When it is not possible to apply new license without application restart the user should be warned about such case and prompted to restart application to apply the changes. In such case `upgradeLicense()` method may looks like:

```
public void upgradeLicense(Component anOwner)
{
  if (ProtectionFactory.createLicensingWizard(licenseHost.
    getLicenseReader(),
    anOwner, productInfo).executeModal())
  {
    JOptionPane.showMessageDialog(anOwner,
      "Application must be restarted for changes to take effect",
      "Message",
      JOptionPane.OK_OPTION | JOptionPane.INFORMATION_MESSAGE);
  }
}
```

Chapter 6

License Checking

Protection! life-cycle consists of two major steps: obtaining a license and checking the license. Instances of `LicenseHost` are responsible for licenses checking.

Tip

It is recommended to use Code Snippets, which are generated by the Protection! Control Center application, not only to learn about Protection! fundamentals, but to also easily embed Protection! into custom applications.

6.1 License Host

License Host is the heart of the entire Protection! framework as it provides actual checking of a license that has been successfully read by the License Reader. It is possible to create License Host via the constructor but it is recommended to utilize the appropriate factory method `ProtectionFactory.createLicenseHost(…)` as this factory takes care of Protection! Edition and therefore creates instances of appropriate `LicenseHost` implementations. License Host works in conjunction with the License Reader object and uses it to read licenses. As a result a properly configured License Reader needs to be specified for the License Host before its first use. Such an approach simplifies the use of Protection! as the application's code should only deal with the License Host which in turn is able to provide all of the required services and information about environment, license, product etc.

To increase efforts required to break protection all License Host methods neither provide any error output nor throw any exceptions. It is possible to change this behavior for development and debugging purposes only by calling `setVerbose(true)` method.

Note

It is the sole responsibility of the application developer to remove any `setVerbose(true)` calls before production deployment of the protected application. Failure to do so could significantly compromise application security by allowing a malicious user to obtain a wealth of information from License Host methods output and/or its exceptions.

The license can carry text of the License Agreement which can be shown in the About Dialog or used to prompt user to accept the terms and conditions of the License Agreement. It can be specified either during issuing of the license or it can be assigned by License Host during the license check procedure. Assigning text of the License Agreement during issuing of the license is not recommended as it greatly increases the size of the license file. However, this can be appropriate for issuing licenses that hold custom license agreements for some particular customers only. In most cases appropriate License Agreement text should be assigned to the License Host. License Host will use the specified License Agreement text during the license check. It is possible to have different License Agreement text for evaluation and for commercial licenses by using `setLicenseTextEvaluation(…)` and `setLicenseTextCommercial(…)` methods respectively.

The main method of License Host is `checkLicense()`.This method performs all of the work associated with the license checking procedure. `checkLicense()` method uses License Reader to obtain the license needed to be checked. If the license has not been read yet then the license would first be read. However, if the license was already read then a cached license will be used for the checking mechanism. It is possible to force license reading and skip the cached, previously read license, by a call of `checkLicense(true)` method.

In most cases the application should call `checkLicense()` during its startup to determine whether the application can run and if so which application's features or functionalities should be enabled. However, Protection! itself does not impose any limitations on when `checkLicense()` metnod should be called. `checkLicense()` method can be called any number of times during the application's run cycle. For example – a trial version of the server-type application can call `checkLicense()` each new day to ensure that the application's evaluation period is not ended. It is a sole responsibility of the application developer to properly handle license checking results and to make appropriate actions to keep the application's state synchronized with the current license state.

There could be several threads running in the background to allow for the support of the network check and the Protection! Licensing Server connection functionalities. If the application completely exits by calling `System.exit(…)` method it should not be concerned about those threads as they will be correctly terminated during the exit. This approach is not suitable for server applications or components (e.g. Servlets or EJB's) as server typically never exits during the termination of server's application. In this case all the background threads should be terminated explicitly during the application shutdown by releasing the License Host via the call of the `release()` method.

6.2 Analyzing License Checking Results

To analyze results of license checking the developer can obtain license instance from the `LicenseHost` and if license is not `null` then analyze its state and query features enabled for this license instance. The license state can be obtained by a call of `license.getLicenseState()` method. The following license states are supported:

- `License.STATE_UNKNOWN` - Defines an unknown state of the license, which means that license just been read and has not yet been checked or validated;
- `License.STATE_OK` - Defines OK state of the license;
- `License.STATE_EXPIRED` - Defines expired state of the license, which means that the license is already expired;
- `License.STATE_INVALID` - Defines an invalid state of the license, which means that the license is issued for a different product or incompatible product's major version.
- `License.STATE_NOT_ACTIVATED` - Defines not activated state of the license, which means that the license is either a valid commercial license or a not yet expired evaluation that needs to be activated.
- `License.STATE_LOCK_VIOLATED` - Defines that the license is a valid one but it has been activated and locked to a different computer system.
- `License.STATE_NOT_LOCKED` – Defines that the license is a valid one but it needs to be locked using Licensing Server.

It is very common for an application to have several editions. Such an application's editions could offer different feature sets i.e. Standard, Professional and Enterprise. In most cases such applications offer a single installation package as it simplifies build,

deployment, distribution and makes simpler deployments of the application's upgrades. In this case the license file specifies the application edition. Protection! includes support of an application's editions by offering a product edition attribute within the license. Product edition attribute can be analyzed by obtaining its value using `license.getProductEdition()` method and therefore enable/disable a particular application's functionality specific to the application's edition.

Dealing directly with the product edition value may seem enough for simple applications but it can be very restrictive for big and sophisticated applications, especially if such applications assume some flexible licensing model with certain features set for each particular customer. Protection! provides a great and easy solution that can help in such cases. This solution is implemented by the "feature set" functionality of Protection!. "feature set" allows definition of application's features representing some part of the application's functionality. The application's features therefore can be individually enabled for each edition or for a particular license. Therefore, instead (or in addition to checking product edition) it is possible to check whether a particular feature is enabled/disabled for a certain license type and license state combination by calling method `license.isProductFeatureSupported(…)`.

Another, very powerful way to analyze license checking is listening the License Host events.

6.3 Listening License Host Events

Protection! provides the ability to inform interested parties about the license checking process by use of an event based model, notifying registered `LicenseHostListener` implementation objects about events that occurred during the license checking process.

Tip

It is recommended that you combine analysis of the license state and handling of the events fired during the license reading and validation to obtain greater control over licensing process.

This listener allows for listening to the following events:

- When the license is expired;
- When the license is OK;
- When the license is invalid;
- When the license is accepted;
- When the license feature is checked;
- When the license is about to expire;
- When the license activation lock is violated;
- When the number of concurrently running copes is violated;
- When the license is not activated.
- When the license is not locked.

Note

All above events are post process and information events only. Therefore it is impossible to write code in event method's implementation to influence license checking.

There is no default implementation of `LicenseHostListener` itself but Protection! offers two implementations of `LicenseListener` interface (It combines both `LicenseReaderListener` and `LicenseHostListener` interfaces):

- `LicenseAdapter` provides a default implementation that prints appropriate messages to the standard output or standard error streams when runs in verbose mode. If verbose mode is off this implementation does nothing.
- `DefaultLicenseAdapter` provides a default implementation suitable for the GUI applications as it shows appropriate message dialogs with the description of the error and vendor's contact information. It can be created either via constructor or via the factory method `ProtectionFactory.createLicenseListener(…)`.

Using `LicenseListener` is the preferred and recommended way as it allows listening to both License Reader and License Host events, which in turn simplifies application development.

Actual implementation of `LicenseListener` methods should take the appropriate action for each event. For example when "license is invalid" event is received it is practical to show the message to the user (if possible) and then simply exit the application. Protection! does not expect, predict or assume any particular actions so it is up to the application's logic to determine what it will do in each case.

Note

Implementation of `LicenseListener` methods must provide a safe environment for calling such methods as many times as necessary (e.g., calling `LicenseListener` methods during issues resolving and/or license upgrading processes. It is up to the application logic to determine how to handle changes in the license – there are neither requirements nor expectations from the Protection! standpoint.

6.4 License Agreement Acceptance

License Host allows checking whether the user has already accepted the terms and conditions of License Agreement. It can be done via delegating such functionality to `LicenseAcceptanceDelegate` instance, which should be assigned to the License Host. Protection! provides two default implementations of `LicenseAcceptanceDelegate` interface:

- GUI implementation that is able to prompt the user to accept the terms of License Agreement by showing an appropriate dialog. An instance of GUI implementation can be created via a call of:
  `ProtectionFactory.createLicenseAcceptanceDelegate(…)` method;
- Headless implementation that always informs License Host that the term of License Agreement is accepted. It can be created via `SimpleLicenseAcceptanceDelegate` constructor. This implementation can be used for headless applications or when the License Agreement needs to be accepted outside of the application e.g. during installation.

6.5 Working with Secret Storages

Secret Storage provides a way to persistently and secretly store various and important information about the application and primarily to support Protection! licensing models and activities. While it is possible to avoid Secret Storages usage in some cases at least one Secret Storage is strongly required to support the following:

a. Flexible Expiration mode.
b. License Activation and Deactivation.
c. Usage based expiration model.
d. Grace periods support.
e. Analyzing application shutdown time.

Protection! provides two ready to use Secret Storage implementations: file based and Java Preferences API based one. Application developers are free to develop and use custom implementations by making proper implementation of the `SecretStorage` interface.

6.5.1 FileSecretStorage Implementation

This implementation of the `SecretStorage` interface stores its data in a local file using very simple encoding. A new instance of `FileSecretStorage` can be created using one of its constructors: `new FileSecretStorage(".DemoCalc", "DemoCalc.dat", true)`.

The Secret Storage's folder can denote an absolute or relative path to the folder. If a relative path is specified it can be resolved as a subfolder of the user's *HOME* folder or current folder of the running application. This behavior can be controlled via `setUserHomeRelative(…)` method.

Note

Actual location of the user's *HOME* folder could depend on how the application is launched. Windows example: if the application is launched as a service the user's *HOME* folder would be *C:*; if the application is launched as a standalone the user's *HOME* folder would be *C:\Documents and Settings\Some User\.*

It is possible to skip encryption of the Secret Storage file for development and debugging purposes only by calling the `setSkipEncryption(true)` method.

To increase efforts required to break protection no `FileSecretStorage` methods provide any error output nor throw any exceptions. It is possible to change this behavior for development and debugging purposes only by calling `setVerbose(true)` method.

6.5.2 PreferencesSecretStorage Implementation

This implementation of the `SecretStorage` stores its data using the Java Preferences API. A new instance of `PreferencesSecretStorage` can be created using one of its constructors. It is possible to explicitly specify a `Preferences` instance or provide arguments needed to locate proper `Preferences` instance. For example call of constructor `new PreferencesSecretStorage("com.acme.app", true)` creates a new `PreferencesSecretStorage` instance that uses the User Preferences node with `"/com/acme/app"` absolute path.

To increase efforts required to break protection no `PreferencesSecretStorage` methods provide any error output nor throw any exceptions. It is possible to change this behavior for development and debugging purposes only by calling `setVerbose(true)` method.

6.6 Resolving License Host Issues

As mentioned above, License Listener does not allow writing of any code to affect license checking and therefore it is not possible to use License Listener to fix any issue(s) that could occur during the license checking. To meet this goal of fixing issues that happened during the license checking, Protection! provides a simple, yet elegant way by using special object (resolver) responsible for resolving such issues.

Each resolver should implement `LicenseHostIssueResolver` interface and each License Host can carry no more than one resolver. This interface introduces methods used by License Host when it discovers any issues:

- `resolveLicenseInvalid(…)` – called when either the product name or its major version do not match.
- `resolveLicenseMissingActivation(…)` – called when license requires activation but Activation Key is missing from the license;
- `resolveLicenseActivationLockViolation(…)` – called when License Host detects that the license has been locked to a particular computer and is being used on another computer system.
- `resolveLicenseExpired(…)` – called when the license is expired;
- `resolveLicenseNotLocked(…)` – called when the license requires a lock from the Application Server but the LicenseHost was unable to acquire such lock.

If resolver's method is able to resolve an issue then it executes all appropriate actions and returns `true` to indicate that License Host should start reading and checking the license again. If resolver returns `false`, then it indicates that resolver was unable to fix the issue and License Host then fires appropriate events to the registered License Host Listener's and stops license checking.

Actual implementation of the resolver methods would perform the following actions:

1. To resolve invalid or expired licenses:
   a. Get a valid license by communicating with the remote backend, by prompting the user to specify the location of a valid license file or by querying the Licensing Server.
   b. Save the obtained license to the file returned by the License Reader's `getLicenseFile()` method.
2. To resolve license activation issues:
   a. Get an activated license by communicating with remote backend or by prompting the user to specify the location of an activated license file.
   b. Save the obtained license to the file returned by the License Reader's `getLicenseFile()` method.
3. To resolve missing lock issues:
   a. Try again to acquire a lock.

Protection! provides two very powerful resolver implementations: GUI and headless. Both implementations can deal with Protection! Backend to get and activate licenses. GUI implementation also provides the ability to resolve missing lock issues and obtain licenses from the Licensing Server.

To create GUI resolver the call of ProtectionFactory.createLicenseHostIssueResolver(…) can be used. This resolver shows Licensing Assistant Wizard or License Activation Assistant Wizard that aids the user in getting or activating licenses. These Wizards provide the following options:

1. Ability to specify the location of a valid license file;
2. Ability to view information on how to obtain the license offline;
3. Ability to request an evaluation or extended evaluation license using either a Direct or a Proxy Internet connection;
4. Ability to obtain a commercial license by providing the Serial Number.
5. Ability to activate a license online.
6. Ability to view information on how to activate the license offline.
7. Ability to assist in resolving missing license lock issues.
8. Ability to obtain licenses by querying Licensing Server for them;

The `HeadlessLicenseHostIssueResolver` class provides implementation of the headless license host resolver. This implementation can be used directly in console or server applications as it does not expect any assistance from the user. It automatically

tries to get the extended evaluation license (if the license is expired and Serial Number is not specified). If Serial Number is provided it tries to get the commercial license instead.

If ready-to-use implementations are not suited for a particular application it is possible to easily implement any required functionality with the help of `IssueResolverSupport` class. `IssueResolverSupport` class provides all of the functionality required to build a custom resolver implementations and hides all communication details with the Protection! backend.

6.7 License Expiration

Protection! supports two types of license expiration: date based and usage based ones. It is possible to combine both expiration types and the license gets expired whichever comes first.

6.7.1 Date Based License Expiration

Protection! provides the ability to limit license usage by specifying a license expiration date. If the license has an expiration date it is considered OK till the system date is less than or equal to the specified expiration date. If the system date is greater than the specified expiration date then the license is considered as expired and therefore gets `License.STATE_EXPIRED` state.

The first known expiration date is automatically maintained in the Secret Storages to eliminate the ability to prolong the expiration period by tampering with the system date/time. This date is bound to the licenses' major version allowing to evaluate the product only once for a specific major version. However in some cases even the minor releases could be considered significant. In order to allow customers re-evaluate such minor releases the license minor version should be taken into the equation. This could be done by overriding the following method:

```
protected String getPreviousExpirationDatePropertyName()
{
  License license = getLicense();
  return  license != null ?
    PROPERTY_PREVIOUS_EXPIRATION_DATE +"["+
    String.valueOf(license.getProductMajorVersion())+"."+
    String.valueOf(license.getProductMinorVersion())+"]" :
    PROPERTY_PREVIOUS_EXPIRATION_DATE;
}
```

Additionally it is possible to enable checking for the last application shutdown date by calling the

`LicenseHostPro.setCheckPreviousShutdownDate(true)` method.

When license is expired then the corresponding flag is stored in the Secret Storages. If this flag is ON then the license will be considered to be expired regardless of the system values, license issue and its expiration dates. This minimizes the ability to circumvent the expiration sub-system by manipulating system's date/time values. This flag is bound to the licenses' major version allowing to evaluate the product only once for a specific major version. However, in some cases even the minor releases could be considered significant. In order to allow customers re-evaluate such minor releases (when the

license has already expired) the license minor version should be taken into the equation. This could be done by overriding the following method:

```
protected String getAlreadyExpiredPropertyName()
{
  License license = getLicense();
  return  license != null ?
    PROPERTY_ALREADY_EXPIRED +"["+
    String.valueOf(license.getProductMajorVersion())+"."+
    String.valueOf(license.getProductMinorVersion())+"]" :
    PROPERTY_ALREADY_EXPIRED;
}
```

Note

Once Evaluation license is expired it is not possible to simply issue another Evaluation license with a new expiration date as the license still will be considered as expired. Extended Evaluation licenses are intended for this purpose and should be issued instead.

It is possible to bundle a license with the application archive allowing easy distribution of a protected application (e.g. distribution via CD). When the bundled license is an Evaluation license it is possible to overcome license issue date allowing the users to start the evaluation based on the application's installation or the application's first use date. This can be done by allowing flexible expiration date via call of

`setAllowFlexibleExpirationDate(true)` method.

Note

Flexible expiration is functional for Evaluation licenses only; expiration of Extended Evaluation or Commercial licenses is not affected.

Protection! provides support for the grace period approach used for date based expiration. Grace period is the number of days within which the user is eligible to continue working with an application even after reaching the expiration date. During the grace period the license is considered OK and corresponding event `licenseAboutToExpire(…)` is fired to notify interested listeners. After the grace period ends the license is considered as expired. Grace period approach could be an invaluable option for some mission critical application where the application could not be interrupted due to the delay by the purchasing department to obtain a new application license.

6.7.2 Usage Based License Expiration

Protection! provides the ability to limit license usage by specifying a license usage quota. If the license has usage limitation it is considered OK until actual license usage has not reached the usage limit. If the actual license usage is greater than the usage quota then the license is considered as expired and therefore gets `License.STATE_EXPIRED` state.

By default the usage count is automatically incremented during the first successful license check. Therefore each application launch, given that the license is OK, is considered as one license usage. While such an approach is reasonable for GUI or for console utility applications it is definitely not suitable for server based applications. It is up to the particular server application code to determine what should be considered as a usage (e.g. request, session etc.). For server applications usage count can be programmatically incremented by calling the `incUseCount()` method. To eliminate

redundant usage count increment during server application startup the usage auto increment feature can be turned off by calling the `setAutoIncUseCount(false)` method before the license checking procedure.

Protection! provides supports for the grace period approach used for usage based expiration. Usage based grace period is the number of usages within which user is eligible to continue working with an application even after reaching the usage quota. During the usage based grace period the license is considered OK and the corresponding event `licenseUseLimitAboutReach(…)` is fired to notify interested listeners. After the usage based grace period ended license is considered as expired.

6.8 User Licensing Models

User Licensing Model defines how to handle different users working with the same license. The following models are supported:

- `License.USER_LICENSING_UNCOUNTED` – any number of users can work concurrently with the license regardless of the license's number of copies value.
- `License.USER_LICENSING_FLOATING` – only a certain number of users can work concurrently with the license according to the license's number of copies value. Actual usage check is done using network broadcast facility.
- `License.USER_LICENSING_NAMED` - only a certain number of users can work with the license according to the license number of copies value. This is done via activation and/or activation-and-lock facility.
- `License.USER_LICENSING_FLOATING_LS` – only a certain number of users can work concurrently with the license according to the license number of copies value. Actual usage check is done by contacting the Licensing Server and acquiring the license lock.
- `License.USER_LICENSING_NAMED_LS` – only a certain number of users can work with the license according to the license number of copies value. Actual usage check is done by contacting the Licensing Server and acquiring the license lock.

The following table shows the summary of the User Licensing Models applicability and some of the requirements to support them:

Model	# Copies Check	Vendor Involvement	Licensing Server	Native Libraries	LAN	WAN
Uncounted	No	No	No	No	N/A	N/A
Floating	Yes	No	No	No	Yes[1]	No
Named	Yes	Yes	No	Yes[2]	N/A	N/A
Floating – LS	Yes	No	Yes	No	Yes	Yes
Named - LS	Yes	No	Yes	No	Yes	Yes

6.8.1 Named User Licensing Model

Protection! supports the Named User Licensing Model via the Activation concept. The Activation process always requires contacting an application vendor/publisher either off-line (e.g. by sending e-mail to the vendor/publisher sales department) or on-line by using Protection! backend. This allows the application vendor/publisher to:

1. Track actual applications deployments. For example, if a bundled evaluation license requires the application evaluator to provide his/her credentials before using a protected application.

[1] Network check may not work in some network configurations when broadcast messages are blocked by firewalls.
[2] Optionally requires native libraries when lock to the MAC address of the network card is used.

2. Implement Named User licensing model by locking the license to a particular computer system and by enabling usage of only a purchased number of application copies by activating only the exact number of licenses specified in the license's "number of copies" attribute.

Activation assumes having Activation Key in the license. As only the vendor is able to place Activation Key to the license then it should be contacted some way to provide an activated license. It can be done automatically via Protection! Backend or manually by placing an Activation Key to the license using Control Center and sending an activated license back to the customer. Therefore, in a nutshell, licensing where activation is required is a two step process:

1. Getting a license from the vendor.
2. Sending activation information to the vendor and getting back an activated license.

This may look slightly overcomplicated but it is designed this way because only the license is a reasonably safe place to store such important information and only the vendor can safely modify license attributes.

The following activation types are supported:

1. Activation – requirement to have certain Activation Key based on license attributes in the license. This type of activation simply tells vendors actual license usage statistics and allows making some decisions based on it.
2. Activation and Lock - requirement to have certain Activation Key based on the license and local computer attributes in the license. This type of activation extends simple Activation by locking the license to a particular computer and therefore eliminating the ability to move and use the license on other computer systems.

Lock Attributes

It is possible to specify several attributes to which a license could be locked. It can be done via `setActivationLockOptions(options)` method; where options could be any set of the following constants:

- `LicenseHostPro.LOCK_MAC_ADDRESS` - Option specifies that license should be locked to MAC address of network card.
- `LicenseHostPro.LOCK_USER` - Option specifies that license should be locked to the current user.
- `LicenseHostPro.LOCK_IP_ADDRESS` - Option specifies that license should be locked to IP address of the local host.
- `LicenseHostPro.LOCK_HOST` - Option specifies that license should be locked to the computer name.
- `LicenseHostPro.LOCK_CPU_NUMBER` - Option specifies that license should be locked to the number of CPU(s).
- `LicenseHostPro.LOCK_INSTALL_ID` - Option specifies that license should be locked to an internal installation ID (randomly generated and securely saved by Protection! to the Secret Storages during the first application run).

A native library is used to get the MAC address of the network card. The following platforms are supported natively:

- Microsoft Windows - Intel x86
- Free BSD - Intel x86
- Linux - Intel x86
- Mac OS X – PowerPC
- Mac OS X – Intel x86
- Solaris - SPARC

By default native libraries should be located in the `../native_lib/` folder. It is possible to specify a different location by call of `setNativeLibFolder(…)` method.

Note

It is not possible to specify location for native libraries for applications build for Java Web Start. Therefore any calls to `setNativeLibFolder(…)` method will be ignored. Use a `org.safehaus.uuid.NativeInterfaces.setUseStdLibDir(true)` call to allow native libraries to be properly found under the Java Web Start.

The set of supported lock attributes is sufficient for the majority of applications. However, if other hardware attributes must be used then it can be done via:

- Writing the native code which would obtain and return information for the required attributes and using JNI to call such native code.
- Using a third-party wrapper over JNI to directly call native methods of underlying OS to obtain information for the required attributes. Good examples of such third-party wrappers are JNIWrapper (www.jniwrapper.com) or xFunction (www.excelsior-usa.com/xfunction.html).

Java6 Note

It is possible to eliminate use of native libraries to get MAC address on Java6. To turn it ON either:

- Specify `LicenseHostPro.setUseJava6Context(true)` in your code or
- Specify a System Property `com.jp.protection.pub.pro.LicenseHostPro.useJava6Context=true`

Activation Keys Generation

If license activation is required, then License Host generates an Activation Key by calling of `getActivationKey()` method and checking whether such property is already present in the license and, if so, whether it has the same, required value. Method `getActivationKey()` in actuality calls the following methods depending on whether the license needs to be locked to a particular computer system:

1. `getActivationKey(license)` – generates Activation Key when license should not be locked to a particular computer. This method's implementation uses concatenation of the license number hash code plus license activation number (number of particular license activations).
2. `getActivationLockKey(license)` – generates Activation Key when license should be locked to a particular computer system. The current implementation provides the ability to use several attributes, mentioned above, to generate the required value. Custom implementation should override this method to add any other required criteria.

To generate the usual form of Activation Key (e.g. `LZL2P-TXDJE-RE3F1-GKOQ5-Q4TJA`) the `getActivationKey(long)` method should be used. It takes a number of long data type which represents the activation key and encodes it into textual form.

While Protection! already provides several attributes to which the license could be locked to, it is possible to use any other custom attribute(s) for that purpose. The following code snippet illustrates how to employ name of OS to be used as the activation and lock key:

```
public class LicenseHostProEx extends LicenseHostPro
{
```

```
  public String getActivationLockKey(String aLicenseNumber)
  {
    return getActivationKey(composeActivationLockKey(aLicenseNumber,
           System.getProperty("os.name").hashCode()),
           getActivationKeyChars());
  }

protected String[] getActivationLockKeys(String aLicenseNumber)
{
  return new String[]{getActivationLockKey(aLicenseNumber)};
}
```

Note

The pair of `getActivationLockKey(aLicenseNumber)` and `getActivationLockKeys(aLicenseNumber)` methods must always be overridden and properly implemented in order to introduce any custom lock attribute or to change how several attributes are merged. The first method should calculate and return primary activation key (e.g. for primary network interface), the second one should calculate and return all available activation keys (e.g. for all available network interfaces).

In some cases Activation Keys would be generated before delivering of actual licenses. To make it works correctly the custom LicenseHost implementation is needed to eliminate requirement to have any license in order to generate Activation Key:

```
class LicenseHostProEx extends LicenseHostPro
{
  public String getActivationLockKey(String aLicenseNumber)
  {
    return super.getActivationLockKey("");
  }

  protected String[] getActivationLockKeys(String aLicenseNumber)
  {
    return super.getActivationLockKeys("");
  }
}
```

Grace Period

Protection! supports grace period for activation – a number of days within which user is eligible to skip activation and to continue working with the application without any limitations on the application's functionality. If the license is still not activated after the end of the grace period the license will be considered as not activated and get the `License.STATE_NOT_ACTIVATED` state. Activation grace period is stored as a `License.PROPERTY_ACTIVATION_GRACE_PERIOD` license property and can be read to provide further analysis if needed. To provide activation grace period functionality, Protection! tracks the first launch date of the application and optionally the application's shutdown date. To enable this functionality valid Secret Storages with write permissions should be specified and there should be no tricks with the system time.

License Deactivation

License deactivation allows de-authorizing the application to use a certain license to either completely terminate using of said license or to move the license to another computer.

The license can be deactivated only if it is a Commercial, valid and currently activated license. It is possible to test whether the current license can be deactivated by call of `canDeactivateLicense()` method.

Deactivation is a two step process:
1. De-authorizing the application to use a certain license. After this step the license is considered not activated on this computer. Note this step is one way only, therefore it cannot be rolled back.
2. Notifying the vendor about deactivation completion. This allows the vendor to remove the vendor's record about previous license activation and to allow further activation on another computer. Note this step is optional and therefore it may be canceled by the user discretion.

The `deactivateLicense()` method allows deactivation of the current license. It returns an encoded Deactivation Key that holds operation result. This key can be sent to Protection! Backend or manually to the application's vendor to confirm that the customer has actually deactivated the license. The operation is considered to be completed if the deactivation data is successfully stored in the Secret Storages. Therefore Secret Storages must be properly configured and have write permissions enabled.

The `deactivateLicense()` method returns one of the following result codes:
- `LicenseDeactivationWizard.DEACTIVATE_FAIL` – deactivation procedure has failed.
- `LicenseDeactivationWizard.DEACTIVATE_LOCAL` – deactivation procedure has completed on the local computer but vendor was not yet been notified about deactivation results.
- `LicenseDeactivationWizard.DEACTIVATE_LOCAL_VENDOR` – deactivation procedure has completed on the local computer and vendor has been notified about deactivation results.

It is the responsibility of the developer to provide proper handling of activation results if needed. For example the application can be forced to exit or all of the application's functionality can be hidden or disabled.

6.8.2 Floating User Licensing Model

This model allows limiting the number of concurrently running applications with the same license by using the number of copies value stated in the license. License Host does this check by sending broadcast network messages and collecting and analyzing responses. This option may work well inside the network of a small/medium size company or within a workgroup. Even though actual broadcast is firewall friendly, the network check could fail if the network has in place firewalls or routers that have been configured to block network broadcast messages. In this case it is better to use the Licensing Server based approach.

To allow a network check it should be enabled for the License Host instance. The network check is enabled by default for the License Host therefore, if network check should be disabled, a call of `setNetworkCheckEnabled(false)` method should be performed.

Handling Number of Copies Violation Event

To handle number of copies violation the application should listen `LicenseHostListener` event and implement required the code in the `numberCopiesViolation(…)` method's body. It is up to the application how to react to such event. For example: the application

can show some message to the user and stop working; or the application can simply log the event for future analysis, etc.

Note

Only the instance(s) of the application(s) that actually exceeded number of copies limit will be notified via firing the `numberCopiesViolation()` event.

Check Host Policy

It is possible to control how several instances of the application running on the same host will be counted. It can be done by specifying policy value via call of the `setNetworkCheckHostPolicy(policy)` method. Where policy value is represented by one of the constants:

3. `NetworkCheck.POLICY_SAME_HOST` – counts applications running on the same host only.
4. `NetworkCheck.POLICY_DIFFERENT_HOST` - counts applications running on the different host only.
5. `NetworkCheck.POLICY_ANY_HOST` - counts applications running on any hosts.

Note

Specified check host policy can be ignored during the license check if the `License.OPTION_COUNT_ANY_HOSTS` or `License.OPTION_COUNT_DIFF_HOSTS` option is explicitly specified for the license.

Releasing Network Check Resources

The network check uses a background thread to support its functionality. If the application completely exits by calling `System.exit(…)` method it should not be concerned about this thread as it will be correctly terminated during the exit. This approach is not suitable for server applications or components (e.g. Servlets or EJB's) as the server is typically never exits during the stop of the server's application. In this case the network check background thread should be terminated explicitly during the application shutdown via the call of `stopNetworkCheck()` method or by simply releasing the License Host by the call of the `release()` method.

6.8.3 Floating and Named User Licensing Models - Licensing Server

The basis for these models is the requirement that the client application must acquire a lock for the license by contacting the Licensing server. The Licensing Server is fully responsible for deciding whether a certain license lock request can be satisfied based on the number of locks already issued, user and application attributes, administrator rules etc. If the license lock is successfully acquired the license is considered OK; otherwise the license gets the `License.STATE_NOT_LOCKED` state and a corresponding `licenseNotLocked` event is fired.

The Licensing Server holds and maintains a list of license sessions for all acquired license locks. This allows for precise calculation of which licenses are in use and how many license copies are in use or free to be used. The license lock can be explicitly released by the client application (e.g. during application exit) or when the Licensing Server detects that some license lock is expired because communication with the application is lost (e.g. due to network and/or the application failures).

The main behavior difference of Floating and Named User Licensing Models is how license sessions are handled after the corresponding license locks is released:

a. Floating User Licensing Model – license sessions are transient. Such license sessions get immediately removed from the list after the release of license locks making the

locks free. Such an approach allows acquiring available license locks by any of the users.

b. Named User Licensing Model – license sessions are persistent. Such license sessions just get marked as "not in use" and continue to stay in the list after the release of license locks, therefore keeping the locks engaged. Such an approach allows reserving a license lock for a particular named user. If there is need to transfer license rights to a different user and/or computer then a corresponding license session should be manually removed by the Licensing Server administrator.

Adding Licensing Server Support

The application code should be aware of how to connect to the Licensing Server in order to support corresponding Licensing Models. It can be done by registering specific provider implementation:

```
LicensingServiceProviderFactory.getInstance().registerProvider("RMI",
  RmiLicensingServiceProvider.class, "localhost", "", "", true);
```

The Licensing Server address attribute passed to the `registerProvider()` method represents the default address that will be used to locate the Licensing Server when the corresponding attribute is not specified in the license. The login and password attributes are reserved for future use therefore any passed values are not used.

Note

If Protection! Launcher is used there is no need to make any changes in the source code. The desired Licensing Service Provider and its attributes should be specified for the product using the Edit Product | Integration | Licensing Server page in the Products Screen of the Proteciton! Developer Control Center. To apply all the changes the Launcher should be simply re-built.

Listening License Lock Expired Event

The `licenseLockExpired` event is called when the license lock is expired. This issue can occur when connection to the Licensing Server is lost due to network or Licensing Server failure. License lock expiration can be handled by:

- Trying to acquire the license lock again.
- Exiting the application.
- Limiting application functionality.
- Allowing the user to complete and save the work by offering some grace period and then exiting the application.

It is not recommended to completely ignore this event, especially for the Floating User Licensing Model. If expiration is caused by the network failure (not by the Licensing Server failure) the Licensing Server will eventually release license lock after some timeout. This means that different users will be able to acquire license locks while other application(s) with expired license locks are still functional. This allows users to overcome number of copies limitations simply by temporary disabling network interfaces, waiting for license lock expiration on the Licensing Server and allowing new users to start using applications.

It is often practical to provide the user with the ability to acquire the license lock again and to continue working. The easiest way to achieve this is by re-launching the license check procedure by calling the `checkLicense()` method. If the network is restored or the Licensing Server is operational at the time of the license check – all the additional activities would be transparent to the user. Otherwise the License Lock Assistant could be shown to help the user in acquiring the license lock.

Listening for License Lock Revoked Event

The `licenseLockRevoked` event is called when Licensing Server administrator requests license lock revocation. This request can be accepted or declined (ignored) depending on the application's nature and its needs and/or user's input. To decline revocation no special actions should be done. To accept revocation the license lock must be released by the `licenseHost.unlockLicense()` method call. One of the following actions would be performed after the loss of license lock:

- Exiting the application.
- Limiting application functionality.
- Allowing the user to complete and save the work by offering some grace period and then exiting the application.

It is **not recommended** to take no actions to handle the loss of license lock as it allows different users to acquire license locks while other application(s) with revoked license locks are still functional. This allows for the ability to overcome the number of copies limitations simply by sequential revoking of the license locks and letting new users start using applications.

Grace Period

Protection! supports grace period for corresponding Licensing Models – a number of days within which the user is eligible to skip license lock acquisition and to continue working with an application without any limitations on that application's functionality.

If the license lock still could not be acquired after the end of the grace period the license will be considered as not locked and will get the `License.STATE_NOT_LOCKED` state. The grace period is stored as a `License.PROPERTY_LOCK_GRACE_PERIOD` license property and can be read to provide further analysis if needed.

To provide the license lock grace period functionality, Protection! tracks the first launch date of the application and optionally the application's shutdown date. To enable this functionality valid Secret Storages with write permissions should be specified.

Check Host Policy

It is possible to control how several instances of the application with the same license running on the same host will be counted and handled:

a. Floating User Licensing Model
   Either `License.OPTION_COUNT_ANY_HOSTS` or `License.OPTION_COUNT_DIFF_HOSTS` options can be explicitly specified for the license to instruct Licensing Server how to count hosts. If neither option is specified the Licensing Server uses its default policy and counts different hosts only.
b. Named User Licensing Model
   The `License.OPTION_DISALLOW_SEVERAL_COPIES_SAME_HOST` option should be explicitly specified for the license to instruct Licensing Server to disallow running several instances of the application on the same host.

Releasing Licensing Server Resources

A background thread responsible for the Licensing Server connectivity should or should not be terminated depending on the following circumstances. If the application completely exits by calling `System.exit(…)` method then the application should not be concerned about this thread as it will be correctly terminated during the exit. This approach is not suitable for server applications or components (e.g. Servlets or EJB's) as the server is typically never exits when the server's application is terminated. In this

case all the background threads should be terminated explicitly during the application shutdown by releasing the License Host via the call of the `release()` method.

Main/Supplemental License Model

Since v4.8 it is possible to increase number of copies of a license (main license thereafter) using supplemental licenses. To make it working a supplemental license should be issued and imported to Licensing Server. This will immediately increase Number of Copies of the main license by the corresponding value provided by a supplemental license. No main license redistribution or any other additional actions are required. If Number of Copies increase should be temporary – a supplemental license should have expiration date. To support this model no rebuild and update of client applications is required; to support that model just Licensing Server should be updated.

The following are requirements for Supplemental licenses:

- Must be a Commercial license
- Must have same product and product edition as the main license
- May have expiration date
- Must have the `protection.ls.mainLicense` custom property with value having the *License Number* of the main license
- Must have the same User Licensing Model as the main license
- Should have fake Licensing Server attribute to avoid using supplemental licenses in old versions of Licensing Server
- All other properties and attributes will be ignored.

Note: supplemental licenses cannot be allocated to the users; only main licenses can be.

Handling License Updates or Change Users Allocation

By default, license updates or change users allocation in Licensing Server will not be propagated to the clients. So the users should either use the Licensing Assistant to re-obtain license or there should be added some code to perform manual/automatic license updates.

Since v4.8 support of license updates is added to the framework making it possible to follow license or user allocation change. Result of the license update operation can be either:

- *License removal* if no suitable license is available anymore.
- *Updated license* if new or changed license is available.
- *Nothing* if current license is up-to-date.

It is possible to listen for license updates events by utilizing extended `LicenseReaderListenerExt` interface.

To enable automatic license updates - turn ON the `ProtectionLauncher.autoUpdateLicenseFromLicensingServer` property (it is OFF by default to support backward compatibility). If this option is turned ON - license update procedure will be performed after each license read operation (on application startup in the most of the cases). Note: this property will take effect only if Launcher uses an instance of `DefaultLicenseReaderIssueResolver` (GUI) to resolve license reading issues. Otherwise change of this property will be ignored.

If automatic updates cannot be setup for some reason or if license updates need to be done on some event or schedule then the `ProtectionLauncher.updateLicenseFromLicensingServer()` method should be used. If license update

succeeded – the license will be automatically re-read and re-checked by calling `ProtectionLauncher.checkLicense(true)` method.

If all the Protection! initialization is done in source code (no Launcher is used) the `HeadlessLicenseReaderIssueResolver.updateLicenseFromLicensingServer(…)` method should be utilized to perform license updates. Note: license will not be reloaded and rechecked after calling of that method; `LicenseReader.getLicense(…, true)` or `LicenseHost.checkLicense(true)` must be called explicitly to apply that change.

Chapter 7

Working with the Licensing Server

The Licensing Server is an application responsible for distribution of the licenses and for tracking of concurrent use of the licenses.

Note

There are completely independent sub-systems for licenses distribution and for concurrent models support. Therefore:

a) Licenses of any type and any user licensing model can be distributed by the Licensing Server.
b) Licensing Server allows tracking of concurrent use of the licenses regardless how those licenses were obtained.

7.1 Registering Licensing Server Connection

The application code should be aware of how to connect to the Licensing Server in order to use its services. It can be done by registering specific provider implementation:

```
LicensingServiceProviderFactory.getInstance().registerProvider("RMI",
  RmiLicensingServiceProvider.class, "localhost", "", "", true);
```

The Licensing Server address attribute passed to the `registerProvider()` method represents the default address that will be used to locate the Licensing Server for either requesting licenses or for tracking concurrent licenses use. Default address attributes will be used when the corresponding attribute is not specified within the license. The login and password attributes are reserved for future use, therefore any passed values will not be used. It is possible to register several providers.

Note

If Protection! Launcher is used there is no need to make any changes in the source code. The desired Licensing Service Provider and its attributes should be specified for the product using the Edit Product | Integration | Licensing Server page in the Products Screen of Developer Control Center. To apply all the changes the Launcher should simply be re-built.

7.2 Using the LicensingServiceSupport

In most cases there is no need to write any code to use Licensing Server functionality as Protection! already includes everything needed. Most of such functionality is encapsulated in the `LicensingServiceSupport` class. An instance of this class is used by the Licensing Assistant and the LicenseHost providing support for getting licenses and handling of concurrent user models. This instance can be obtained by calling the `getLicensingServiceSupport()` method of the License Host.

7.3 Direct Use of Licensing Service

While `LicensingServiceSupport` class is sufficient in most cases it is possible to use Licensing Services directly to get greater control of the process.

7.3.1 Getting Licensing Service

To get the Licensing Service an instance of the provider should be created by the factory method:

```
LicensingServiceProvider provider = LicensingServiceProviderFactory.
  getInstance().createProvider(null, "localhost", null);
```

It is possible to be notified about various provider events (e.g. when Licensing Service obtained or license locked) by making proper implementation of the `LicensingServiceProviderListener` interface and adding it to the listeners list of provider by the `addLicensingServiceProviderListener(…)` method call.

When an instance of provider is created it can be used to produce a Licensing Service instance that provides remote access to the Licensing Server functionality:

```
LicensingService service = provider.createLicensingService(
  licenseHost.getContext().createAttributes(true));
```

7.3.2 Using Licensing Services

By using the Licensing Service it is possible to:

- Get licenses.
- Acquire license locks.
- Confirm license locks.
- Release license locks.
- Check license locks availability.

Most of the `LicensingService` methods take a `LicenseData.Request` argument that exactly describes a license to get or to lock. It is possible to use custom implementation of this interface or utilize ready-to-use `LicenseDataImpl.RequestImpl` implementation.

All the `LicensingService` methods take a `Map` argument that defines system attributes like the user and host name. Names of such attributes are defined as a set of `LicensingService.ATTR_` constants. It is possible to fill the system attributes using any suitable API or all of the attributes can be easily obtained by the `licenseHost.getContext().createAttributes(true)` call.

The `confirmLicenseLock(…)` and `unlockLicense(…)` methods take a unique ticket argument that identifies the previously obtained license lock.

The `LicensingService` methods return an instance of the `LicensingServiceResult` which provides execution result, message and optional ticket. The following are possible result values:

- `LicensingServiceResult.RESULT_OK` - operation succeeded e.g. the license has been successfully obtained or locked.
- `LicensingFacadeResult.RESULT_ERROR` - operation failed due to some business reason (e.g. when no license is allocated to particular user). In such case the message in the execution result typically contains an explanation or the reason for execution failure. This message may be shown to the user to let him/her know what went wrong.
- `LicensingFacadeResult.RESULT_SYSTEM_ERROR` - operation failed due to some system error. In such a case the message in the execution result contains an error message typically obtained from an exception. It is unnecessary to show this message to the user as it may be meaningless to him/her; general error message

(e.g. "Unable to proceed. Try again or contact your system administrator for assistance") would be used instead.

- `LicensingFacadeResult.RESULT_LICENSE_REVOKED` – indicates that license lock revocation has been requested by the Licensing Server administrator.
- `LicensingFacadeResult.RESULT_SESSION_EXPIRED` – means that the license lock is expired and therefore the application no longer has its license lock.

The ticket property of the `LicensingServiceResult` identifies the obtained license lock and should be passed to `confirmLicenseLock(…)` and `unlockLicense(…)` methods.

The `lockLicense(…)` method allows for the license lock acquisition. The Licensing Server maintains all of the previously obtained locks for the given license and will decide whether a new lock can be obtained based on the license number of copies and according to the host counting policy. If the license lock is obtained then the application may continue to work without limitations. If the license lock is not obtained the application could wait for it; otherwise some of the application's functionality limitations should be applied or the application could simply exit.

The `canLockLicense(…)` method allows for querying the License Server to check that a specific license lock could be obtained. If it succeeded, this method does not reserve any future rights to obtain a license lock therefore sequential call of `lockLicense(…)` could fail if another application obtains the license lock first.

The `confirmLicenseLock(…)` method allows for the Licensing Server notification that the application which holds certain license lock is alive and this license lock is still in use. When default Licensing Service implementation is used the application developer should not be concerned with writing any code to confirm license locks. The default implementation maintains all of the acquired license locks and confirms them as well.

The `unlockLicense(…)` method allows for the Licensing Server notification that certain license lock is no longer needed and therefore it must be released. In most cases the license lock should be released only when applications exit. If an application continues to run after releasing the license lock the application developer should write appropriate code to disable/limit access to the application's functionality.

The `getLicense(…)` method allows for getting a license according to the requested license attributes like product and version. If there is a license allocated to a particular user this method will return an instance of the `LicenseResult` (that sub-classes the `LicensingServiceResult)` and holds the license bytes attributes. If license bytes are not `null` they contain the license that would be saved to the file known to the License Reader (it can be obtained by a call of `licenseReader.getLicenseFile()`).

The following code sample shows how to get a license:

```
LicensingServiceProvider provider = LicensingServiceProviderFactory.
  getInstance().createProvider(null, "localhost", null);
try
{
  Map attrs = licenseHost.getContext().createAttributes(true);
  LicensingService service = provider.createLicensingService(attrs);

  LicenseResult result = (LicenseResult) aLicensingService.
    getLicense(LicensingServiceSupport.createRequest(productInfo), attrs);
  if (result.getResult() == LicensingServiceResult.RESULT_OK)
```

```
  {
    // use result.getResponce().getLicenseBytes() to save license
  }
}
finally
{
  provider.release();
}
```

7.3.3 Releasing Licensing Service

When Licensing Service is no longer needed it should be explicitly released by calling `releaseLicensingService()` method of the provider. During release of the Licensing Service all the resources including license locks acquired and maintained through this service will be released as well.

When all the Licensing Services provided by provider (and provider itself) are no longer needed the provider should be explicitly released by calling its `release()` method. During release of provider all the resources including Licensing Services created by this provider will be released as well.

Chapter 8

Integrity Verification

The Integrity verification subsystem allows for checking and validation that the designated key classes, resources or files of the application have not been changed. This type of check is done by comparing current actual digest of those classes, resources, or files with the original value stored somewhere in the application code, resources or files.

The `IntegrityHostPro` (or `IntegrityHost` for Protection! Std) class provides the `check(…)` and `checkStatic(…)` methods allowing you to verify the application data and code integrity. When an integrity violation is detected (a `check()` method returned `false` value) some actions should be executed to prevent probably unauthorized use of the application. There are no ready-to-use actions provided by Protection! because proper handling of integrity violations is the sole responsibility of the developer and specific to each application.

To start, the Integrity verification subsystem should be properly initialized. The initialization data includes:
1. Classes, resources and files to check.
2. Digest algorithm.
3. Digest.

There are several implementations depending on how initialization data is loaded and handled at runtime. This behavior is controlled by the runtime type attribute (Edit Product | Integrity Check | Runtime Configuration Type).

Tip

The Protection! Developer Control Center generates product specific code snippets to allow for easy implementation of the Integrity verification into the custom applications. Therefore all of the following code examples can be seen and easily saved right from the code snippet section.

8.1 Source Runtime Configuration

When the "Source" Runtime Configuration is selected all the initialization data is explicitly defined in the source code as a set of properly initialized constants.

The following example shows how to check integrity and print error message when violation is detected:

```
final byte[] DIGEST = new byte[]
{-38, 26, -111, 97, 52, -107, 120, -27, 118, -13, 77,
 -28, 23, 34, -48, -39, 104, -99, -66, 111};
final String[] DIGEST_ENTRIES =
{
  "com/jp/samples/protection/DemoCalcProtectionSupport$1.class",
  "com/jp/samples/protection/DemoCalcProtectionSupport.class"
};
final String[] DIGEST_FILE_ENTRIES = null;
final int ALGORITHM = IntegrityHost.ALGORITHM_SHA_1;
if (! IntegrityHostPro.checkStatic(DIGEST_ENTRIES,
```

```
                                        DIGEST_FILE_ENTRIES,
                                        DIGEST,
                                        ALGORITHM)
{
  System.err.println("Integrity violation");
}
```

The following are drawbacks of "Source" Runtime Configuration:

1. Any changes in the initialization data lead to corresponding changes in the source code that in turn would require rebuild of the application.
2. It is almost impossible to support Integrity check for obfuscated code.

Tip

It is recommended to use either the "Resource" or the "File" Runtime Configurations in most implementations. Usage of the "Source" Runtime Configuration could be justified for very specific cases only.

8.2 Resource Runtime Configuration

When the "Resource" Runtime Configuration is selected all the initialization data is explicitly defined in an application resource that could be loaded at runtime.

The following example shows how to load configuration from a resource, check integrity and print error message when violation is detected:

```
IntegrityHostProConfigReader reader = new IntegrityHostProConfigReader();
reader.setSecurityAlgorithm("RSA - 512");
reader.setDecryptKeyBytes("…"); // actual data must be here

if (! IntegrityHostPro.checkStatic(reader, "DemoCalc.digest"))
{
  System.err.println("Integrity violation");
}
```

8.3 File Runtime Configuration

When the "File" Runtime Configuration is selected all the initialization data is explicitly defined in a file that could be loaded at runtime.

The following example shows how to load configuration from a file, check integrity and print an error message when violation is detected:

```
IntegrityHostProConfigReader reader = new IntegrityHostProConfigReader();
reader.setSecurityAlgorithm("RSA - 512");
reader.setDecryptKeyBytes("…"); // actual data must be here

if (! IntegrityHostPro.checkStatic(reader, new File("DemoCalc.digest")))
{
  System.err.println("Integrity violation");
}
```

Both the "Resource" and the "File" Runtime Configurations have a number of advantages over the "Source" configuration:

1. No need to rebuild the application to apply initialization data changes.

2. Ability to use build systems and scripts to automate incorporation of the initialization data changes.
3. Obfuscated code support.

Chapter 9

Licensing Using Protection! Backend

Protection! Professional provides the ability to embed the process of license getting and activation functionality directly into the protected applications. This functionality via remote connectivity deals with the Protection! backend deployed at the vendors/publishers site. Though Protection! currently includes only the Web Services based implementation it is possible to introduce other, different implementations e.g. RMI or raw TCP/IP based. Because most of Protection! client (public) code never deals directly with nor knows about actual implementation it is possible to start with the default implementation and later add any custom one. The `LicensingFacade` interface provides methods required for licenses acquisition and activation.

9.1 Registering the LicensingFacade Implementation

Actual implementation should be registered before the first use of Licensing Façade. The following sample code demonstrates how to register a default Web Services based implementation that will go to the vendor/publisher site and allow licensing of the Demo Calculator application.

```
LicensingFacadeProvider.addProvider("WS", new LicensingFacadeWS.Provider(
  new URL("http://www.vendordomain.com"+
  "/ProtectionWS/services/LicensingFacade"),
  "DemoCalc", "DemoCalc"));
```

It is also suitable for debugging of applications to eliminate remote calls by starting Protection! backend in-process. It can be done by the following call.

```
LicensingFacadeProvider.addProvider("Local",new
  LocalLicensingFacade.Provider(new File("products.dat"),
  "DemoCalc", "DemoCalc"));
```

Note

If Protection! Launcher is used then there is no need to make any changes in the source code. The desired Licensing Façade Provider and its attributes should be specified for the product using the Edit Product | Integration | Façade page in the Products Screen of the Protection! Developer Control Center. To apply all the changes the Launcher must be re-built.

After completion of the registration process the application is able to communicate to the backend to request or activate licenses. For example, the application can now request evaluation licenses using Licensing Assistant Wizard.

Note

If the registration is missing or incomplete the Licensing Assistant and the License Activation Assistant Wizards hide all their functionality that depends on the Licensing Façade.

9.2 Getting the LicensingFacade

The Licensing Facade Provider provides the actual implementation. The following code `LicensingFacadeProvider.getInstance().getLicensingFacade()` allows obtaining the default registered implementation. It is possible to get specific implementation if several implementations are known to the application by calling of `LicensingFacadeProvider.getInstance("WS").getLicensingFacade()` and providing implementation's name e.g. "WS".

9.3 Using the LicensingFacade

By using the Licensing Façade it is possible to:

- Get commercial or evaluation licenses for requested products.
- Get commercial license by Serial Numbers.
- Activate licenses.
- Deactivate licenses (at the same time notifying vendors about deactivation)
- Get version of backend implementation;

Most of `LicensingFacade` methods take as a parameter an instance of the `Customer` that holds the customer's data who is requesting and/or activating the license. Other important parameters are login and password. These parameters provide credentials that may be required by Protection! backend to authorize the customer in order to proceed with the requested operations.

It is possible to send any number of arguments to the backend by putting them to a `HashMap` instance and further passing this instance to the `LicensingFacade` methods as an argument. Note, keys and values which can be sent should be of the `String` type only. By default, current Locale name is passed as `LicensingFacade.ARGUMENT_LOCALE` argument allowing backend to generate localized messages for the client applications. The `LicensingFacade.ARGUMENT_ACTIVATION_LOCK_KEY` and `LicensingFacade.ARGUMENT_ACTIVATION_KEY` arguments are passed by `LicensingWizardPro` implementation during getting licenses to provide a one step activation process.

The `LicensingFacade` methods return instance of `LicensingFacadeResult` that carries execution result, license bytes and an optional message.

If the execution result is `LicensingFacadeResult.RESULT_OK` this means that the license has been successfully obtained or activated. If license bytes are not `null` they contain the license that should be saved to the file known to the License Reader (it can be obtained by a call of `licenseReader.getLicenseFile()`). If license bytes are `null` the license has been delivered to the user using a different way of delivery (e.g. e-mail) and the user is responsible for getting the license himself/herself. In such a case the message in the execution result typically contains instructions that may be shown to the user to aid him/her on how to obtain and apply the license.

If the execution result is `LicensingFacadeResult.RESULT_ERROR` it means that either license getting or activation procedures have failed due to some reason (e.g. when the requested license already has been issued). In such a case the message in the execution result typically contains an explanation or the reason for execution failure. This message may be shown to the user to let him/her know what went wrong.

The following code example shows how to use Licensing Façade to request a license.

```
public boolean obtainLicense(LicenseReader aLicenseReader,
    LicenseInfo aLicense, Customer aCustomer,
    String aLogin, String aPassword)
{
  boolean result = false;
  LicensingFacadeResult requestResult = null;
  try
  {
    LicensingFacade licensingFacade = LicensingFacadeProvider.
      getInstance().getLicensingFacade();
    if (licensingFacade != null)
    {
      requestResult = licensingFacade.getLicense(aLicense, aCustomer,
        aLogin, aPassword, null);
      if (requestResult.getResult() == LicensingFacadeResult.RESULT_OK)
      {
        byte[] licenseBytes = licenseBytes = requestResult.
          getLicenseBytes();
        if (licenseBytes != null)
        {
          /** @todo add code to save license here */
        }
        else
        {
          System.out.println(requestResult.getMessage());
        }
        result = true;
      }
      else
      {
        System.err.println("Unable to get license: "+ requestResult.
            getMessage());
      }
    }
    else
    {
      System.err.println("Unable to get LicensingFacade");
    }
  }
  catch (Exception ex)
  {
    ex.printStackTrace();
  }
  return result;
}
```

9.4 Using the LicensingFacade to Implement Licensing of the Web Applications

Protection! provides ready to use components and Wizards to add the ability to get and activate licenses for any Swing GUI applications. The GUI developers should only be concerned with the proper Protection! classes initialization. Although there is no such

ready-to-use implementation for Web applications it is quite easy to make such an implementation using the following scenario:

1. Add the required code to the headless code snippet generated by the Protection! Control Center to disable or constrain Web application functionality and to eliminate unlicensed use.
2. Implement Web registration form to allow entering user's data (name, e-mail etc.), optional Serial Number or specifying the need to activate a license.
3. Implement code to process Registration Form submission and prepare data required to use Licensing Façade (e.g. create an instance of `Customer` that holds customer information).
4. Use Licensing Façade to get or activate a license and save it to the file known for the License Reader.
5. Use the usual check license sequence to check the license.

When the customer wants to use such a Web application he/she should either place a valid license to the required location or use Registration Form to get an evaluation license, enter Serial Number to get a commercial license or initiate the license activation process.

Chapter 10

Deploying Protected Applications

For best results, use the Archive Builder included with your IDE or provided by JDK to create the .zip or .jar file(s) for use in deploying your target application, Applet, or JavaBean component.

To the extent that you are allowed to redistribute third-party redistributables under the terms of your license, you may redistribute these classes/resources in any of the formats subject to the restrictions in the third-party license agreement. Refer to third-party-license.html for the details regarding redistribution.

The following topics outline redistributables needed to be used to deploy protected applications.

10.1 Required Redistributables

For the purpose of the license(s) included with this product, the redistributables for the Protection! Licensing Toolkit are located in the *<install_dir>/lib* folder and defined as:

1. *Protection.jar* - Protection! core.
2. *bcprov-jdk14-136.jar*- Bouncy Castle Crypto API.

To the extent that you are allowed to redistribute redistributables under the terms of your license, you may redistribute these classes in any of the following formats subject to the restrictions in the license agreement.

1. Protection! Core
   a) As the entire *Protection.jar* file.
   b) As content of *Protection.jar* bundled (and optionally obfuscated) into your application archives.
2. Bouncy Castle Crypto API
   a) As the entire *bcprov-jdk14-136.jar* file.

Note: It is strongly recommended that you include the content of *Protection.jar* into your applications archive and obfuscate it as a whole using some powerful obfuscator to maximize efforts needed to reverse engineer and understand licensing and protection logic.

10.2 Redistributables to Allow Support for Named User Licensing Model (Optional)

For the purpose of the license(s) included with this product, the redistributables for the Named User licensing model is defined as the *jug.jar* file located in the *<install_dir>/lib* folder and the set of native libraries located in the *<install_dir>/native_lib* folder.

Note: By default, native libraries need to be located in the *../native_lib* folder relative to your applications root. However, it is possible to specify another location using `setNativeLibFolder(…)` method of `LicenseHostPro`.

10.3 Redistributables to Work with Protection! Web Services Application (Optional)

For the purpose of the license(s) included with this product, the redistributables to allow for acquisition and/or activation of the licenses via Protection! Web Services Application are defined as the following files located in the <*install_dir*>*/lib* folder:

a) axis.jar
b) axis-ant.jar
c) commons-discovery.jar
d) commons-logging.jar
e) jaxrpc.jar
f) log4j-1.2.8.jar
g) saaj.jar
h) wsdl4j-1.5.1.jar

Chapter 11

Protection! Backend

In addition to the developer's ability to generate licenses using Protection! Control Center it is also possible to employ Protection! API to perform these and other tasks programmatically. This functionality can be used to build Protection! Backend able to process a license request by the client applications and to create applications capable of mass licenses generation, etc.

11.1 Working with Products Storage

The majority of Protection! Backend tasks deal with license generation and therefore require an explicit knowledge of a particular product for which licenses must be generated. The Products Storage provides all of the required information for the above task as it holds products and allows you to query product attributes. Before the first use it should be properly initialized e.g. by loading from a file:

```
ProductsStorage productStorage = new ProductsStorage();
FileInputStream fileInputStream = new FileInputStream("products.dat");
try
{
  productsStorage.load(fileInputStream);
}
finally
{
  fileInputStream.close();
}
```

If read protection is turned on for a particular Products Storage then the valid password must be provided before such product storage could be loaded. This can be done by making a proper implementation of the `PasswordResolver` interface and assigning a new instance of it to the Products Storage:

```
productStorage.setPasswordResolver(new ProductsStorage.PasswordResolver()
{
  public String resolvePassword()
  {
    return "1234";
  }

  public void passwordMismatch()
  {
    System.err.println("Password is invalid");
  }
});
```

11.2 Encoding and Writing Licenses

11.2.1 Licensing Protection! Backend API

To get access to this API a valid Protection! Backend Deployment license should be available at the known location.

By default, Protection! tries to find a Backend Deployment license named *protection.backend.license* in the user's *<HOME>.protection4* (e.g. "*C:\Documents and Settings\<User Name>\.protection4*" for Windows) folder first.

Note

Evaluation or commercial licenses for Protection! Developer allows using Protection! Backend API for development purposes. To allow it the *protection.license* should be copied and renamed to *protection.backend.license.*

If the license is not found then Protection! tries to get it from the root of the application archive or class path. This default search sequence can be changed by setting system property `ProtectionFactory.PROPERTY_LICENSE_WRITER_SEARCH_LICENSE_RESOURCE` to the `true` value:

```
System.setProperty(
  ProtectionFactory.PROPERTY_LICENSE_WRITER_SEARCH_LICENSE_RESOURCE,
  "true");
```

Note

Protection! Web Services Application will always change the default license search sequence to allow first for discovery of a bundled license.

While the user's *HOME* folder seems to be a reasonably good location for a Backend Deployment license it is possible to specify another desired location of the license by setting system property ProtectionFactory.PROPERTY_LICENSE_FOLDER to let Protection! know exactly where the license should be located:

```
System.setProperty(ProtectionFactory.PROPERTY_LICENSE_FOLDER,
  ".demoCalc");
```

Note

The specified license folder always denotes the relative path to the user *HOME* folder.

The Backend Deployment license file within the application archive or class path can also be placed to some folder that differs from the root one. The system property `ProtectionFactory.PROPERTY_LICENSE_RESOURCE_FOLDER` allows specifying the license location in such a case:

```
System.setProperty(ProtectionFactory.PROPERTY_LICENSE_RESOURCE_FOLDER,
 "/com/jp/samples/protection/");
```

It is possible to change the Backend Deployment license name (*protection.license* by default) e.g. to hide the existence of Protection! by specifying a new license name as a value for `ProtectionFactory.PROPERTY_LICENSE_NAME` system property:

```
System.setProperty(ProtectionFactory.PROPERTY_LICENSE_NAME,
 "DemoCalc.bin");
```

11.2.2 Using License Writer

The `LicenseWriter` class provides the ability to encode passed license and then save it to the specified file or output stream. Once created, the instance of `LicenseWriter` should be properly configured by specifying encryption key bytes that can be obtained by reading the appropriate product properties.

The `write(…)` methods should be used to encode and write licenses. These methods take a `License` instance that precisely describes the license to be written as the first parameter and file or output stream as the second one.

It is possible to skip license encryption (for debugging purposes only) during license writing by calling of `setSkipEncryption(true)` method. Certainly `LicenseReader` object in the client application can be instructed to skip encryption too to enable reading of unencrypted licenses.

To increase efforts required to break protection none of the `LicenseWriter` methods show any error output or throw any exceptions. It is possible to change this behavior for development and debugging purposes only by calling `setVerbose(true)` method.

The following code fragment shows how to encode and write a sample license to the *DemoCalc.key* file.

```
// create a license and specify its attributes
LicenseImpl license = new LicenseImpl();
license.setProduct("DemoCalc");
license.setLicenseNumber("1");
license.setLicenseType(License.TYPE_COMMERCIAL);

// create and configure license writer, encode and write license
LicenseWriter licenseWriter = new LicenseWriter();
licenseWriter. setEncryptKeyBytes (new byte[]{…}); // specify license key
bytes here
licenseWriter.writeLicense(license, new File("DemoCalc.key"));
```

The `LicenseWriter` does provide all required low-level API to encode and write licenses. However, it is quite hard to properly configure it as appropriate product should first be located and then read from the corresponding product storage to obtain encryption key bytes. This is why developers should never deal with `LicenseWriter` directly in the majority of cases. The `LicenseWriterFacade` class greatly simplifies the process of `LicenseWriter` configuration and also provides several other very helpful methods.

The `LicenseWriterFacade` object uses associated products storage to locate required products and to create and configure corresponding `LicenseWriter` instances to provide license encoding and saving mechanisms. Other major functionality of `LicenseWriterFacade` is its ability to function as a factory capable of creation of valid licenses for a specified product, product edition, license types etc. While it is possible to fully configure license instances programmatically it is recommended and much better to use `createLicense(…)` methods of `LicenseWriterFacade` as these methods will take care of the proper license initialization (e.g. putting correct features set for specified product and product edition).

The following code fragment shows how easily a developer could achieve the tasks of creation, putting activation key, encoding and writing a sample license to the *DemoCalc.key* file.

```
try
{
  LicenseWriterFacade licenseWriter =
    new LicenseWriterFacade(new File("products.dat"));
  LicenseImpl license = licenseWriter.createLicense("DemoCalc", "1",
```

```
    License.TYPE_COMMERCIAL);
  license.setLicenseOptions(License.OPTION_REQUIRE_ACTIVATION);
  license.putProperty(License.PROPERTY_ACTIVATION_KEY,
    "LZL2P-TXDJE-RE3F1-GKOQ5-Q4TJA");
  licenseWriter.writeLicense(license, new File("DemoCalc.key"));
}
catch (Exception ex)
{
  ex.printStackTrace();
}
```

11.3 Validating Deactivation Result

Result of the deactivation process is represented by the deactivation key which the user should provide to the vendor (along with the product name and the activation key). The deactivation key should be checked at the vendor side to ensure that the deactivation process has actually been completed. This can be done via the communication with the Protection! Backend or with the help of the corresponding Protection! Sales Control Center functionality.

It is also possible to use Protection! API to check the deactivation key:

```
String activationKey =  ...; // provide activation key
String deactivationKey =  ...; // provide deactivation key
Product product = productsStorage.getProduct("DemoCalc");

if (deactivationKey.equals(LocalLicensingFacade.
  getDeactivationKey(product, activationKey, true, true)))
{
  // deactivation successfully completed
}
else if (deactivationKey.equals(LocalLicensingFacade.
  getDeactivationKey(product, activationKey, false, true)))
{
  // deactivation failed
}
else
{
  // error e.g. improper deactivation and/or activation keys
}
```

11.4 Working with Serial Numbers

Protection! backend provides a complete set of API to support Serial Numbers generation and parsing.

Note

At least one License Alias must be created in the Product Storage to allow working with Serial Numbers.

11.4.1 Serial Numbers Generation

A properly initialized Products Storage is required to generate the Serial Numbers. A new instance of Serial Number must be created using the Products Storage:

```
ProductsStorage productsStorage = new ProductsStorage();
productsStorage.load(new FileInputStream(new File("products.dat")));
SerialNumber2 serialNumber = productStorage.getSerialNumber2();
```

The License Number, License Alias identifier and Number of Copies attributes should be specified for Serial Number:

```
serialNumber.setLicenseNumber(12345);

LicenseAlias alias = productStorage.getLicenseAlias("test");
serialNumber.setAlias(Long.parseLong(alias.getAlias()));

serialNumber.setNumberCopies(10);
```

And finally textual representation of Serial Number can be obtained:

```
String textSN = serialNumber.encode();
```

This textual representation can be sent to the customer or printed in the product box.

11.4.2 Serial Numbers Parsing

The main purpose of Serial Numbers is the ability to restore particular licenses using a textual representation of them. It can be easily done using a properly configured LicenseWriterFacade:

```
LicenseWriterFacade writerFacade =
  new LicenseWriterFacade(new File("products.dat"));

LicenseImpl license = writerFacade.
  createLicenseForSerialNumber("BUK3X-9AY4Z-O7NM2-D65WG-UKRXC");
```

Restored license can be delivered to the customer by any suitable way e.g. by e-mail.

11.5 Orders Parsing

Protection! includes license order parsing functionality which allows for extraction of the license and customer information directly from the order notification received from online E-Commerce stores (i.e., Element 5).

The `OrderParserFacade` class represents the entry point to the license orders parsing system. It provides a set of convenient methods for system initialization, order parsing and listening for events that occurred during the process. Properly initialized Products Storage should be passed to the `OrderParserFacade` constructor so it is knowledgeable of the available Products and License Aliases:

```
ProductsStorage productsStorage = new ProductsStorage();
FileInputStream fileInputStream = new FileInputStream("products.dat");
try
{
  productsStorage.load(fileInputStream);
}
finally
{
  fileInputStream.close();
}
```

```
OrderParserFacade facade = new OrderParserFacade(productsStorage);
```

The result of a successful license order parsing process is always represented by an instance of `OrderParserFacade.Result` class per each parsed order. Each `OrderParserFacade.Result` instance holds License Alias, License and Customer that correspond to information obtained from the license order.

It is possible to get notified about progress of license order parsing by making proper implementation of `OrderParserListener` interface and adding it to the list of listeners of `OrderParserFacade` instance.

In addition to listening events the `getErrors()` method allows getting the list of errors which occurred during the last license order parsing operation. This list contains the `OrderParser.Error` instances which contains the error message and its details.

11.5.1 Single Order Parsing

There are several `parse(…)` methods in the `OrderParserFacade` implementation responsible for parsing a single license order.

If several different order formats needs to be supported, then the `parse(InputStream)` method should be used. It accepts license order content as an input stream and walks through all of the registered parsers to find one that is able to parse the supplied order. If no suitable parser is found the method simply returns `null` value. If a parser is found, then it will be used to parse the license order and utilize parsed data to prepare `OrderParserFacade.Result` instance by finding License Alias, creating corresponding License and filling-in the Customer information.

```
FileInputStream inputStream = new FileInputStream("order.txt");
try
{
  OrderParserFacade.Result result = facade.parse(inputStream);
}
finally
{
  fileInputStream.close();
}
```

If the order format is known the `parse(InputStream, String)` or `parse(InputStream, OrderParser)` methods can be used. The first method uses a parser specified by its unique identifier; the second method uses an explicitly specified parser. The following code illustrates how to parse an order by using the generic order format:

```
FileInputStream inputStream = new FileInputStream("order.txt");
try
{
  OrderParserFacade.Result result = facade.parse(inputStream,
    "#Generic License Order#");
}
finally
{
  inputStream.close();
}
```

11.5.2 Bulk Orders Parsing

It is possible to (not recursively) parse all the orders found in some folder. The `parse(File, List)` method provides this ability. It takes a folder to parse orders from as the first parameter and list to hold found errors as the second parameter and returns a list of `OrderParserFacade.Result` instances which represents all the parsed orders.

The following example illustrates how to parse all the orders from the specified folder and dump them to the standard output in comma delimited form:

```
List errors = new ArrayList();
List parseResults = facade.parse(new File("Incoming/Orders"), errors);
StringBuffer stringBuffer = new StringBuffer();
facade.dump(parseResults, stringBuffer);
System.out.println(stringBuffer.toString());
```

11.5.3 Using Generic Order Format

If your current order structure is not yet supported by Protection! order parsing functionality and you would like Protection!'s assistance in order parsing, then you can modify your current order to fit the generic format outlined bellow and then supply it to Protection! for automated parsing.

```
_____________________________

Product              = 1234
Number of licenses   = 1
License number       = 1234

Net sales            = USD   1000.00
Total                = USD   900.00
Payment              = Credit Card: Visa

Salutation           = MR.
Title                =
Last Name            = Doe
First Name           = Joe
Company              = Acme Corporation, Inc.
Street               = 1 Acme way
ZIP                  = 90012
City                 = Acme Industrial City
Country              = USA
State / Province     = California
Phone                = 555-555-1212
E-Mail               = john@acme.com
_____________________________

#Generic License Order#
_____________________________
```

In this format the following tags are very important:

1. The "`Product`" tag specifies License Alias that corresponds to the license that has been sold.
2. The "`#Generic License Order#`" tag specifies name of the generic format.

Note all the generic tags should have exactly the same names as stated above e.g.

"`Product              =`"

but can be placed in other different order sequence.

11.6 Working with Protection! WS Application

Protection provides default implementation of licensing backend as a Web application that exports its functionality via Web Services. This application can be deployed to and hosted on any Web container that supports Servlet 2.3 and JSP 1.2 specifications e.g. Apache Tomcat 5.x.

Use of Web Services provides the ability for third-party developers, vendors or resellers to generate or activate licenses using tools of their choice e.g. Borland Delphi or Visual Basic .NET. They should use the appropriate WSDL descriptor that describes Licensing Façade interface. Developers should also generate proper implementation or support code to access the Licensing Façade functionality. This descriptor can be easy obtained using ?wsdl command. For example the following URL allows for obtaining WSDL descriptor for Licensing Façade deployed at jProductivity Web site:

http://services.jproductivity.net:8080/ProtectionWS/services/LicensingFacade?wsdl

While it is possible to use virtually any language or tool to access the Protection! backend, Java applications that use Protection! for licensing should not directly deal with Protection! Web Services. This is because all of the necessary code is already included in the Protection! framework where it is very easy to request or activate licenses without the need to know or understand Web Services.

The only required step is getting a Licensing Façade instance by call of `LicensingFacadeProvider.getInstance().getLicensingFacade()` method and further utilization of the `LicensingFacade` methods to get or activate licenses.

11.6.1 Configuring Protection! WS Application

Though Protection! WS Application represents a valid and functional Web application it lacks several things required for proper license generation and activation:

- *Valid Protection! Backend Deployment license*. This license is required in order to provide full commercial licensed use of the Protection! backend. Protection! Professional Edition does include a Protection! Backend Developer License that can be used for development purposes only. To enable full production deployment and usage of Protection! Backend a Protection! Backend Deployment License must be purchased.
- *Products Storage file.* This file is required in order to let the Protection! backend know which licenses can be generated for which products and to supply proper keys for licenses encoding.
- *Backend configuration options* to fine tune its functionality.

Protection! Control Center provides familiar GUI front-end allowing configuration of Protection! Web Services Application without the need to understand Web development practices and approaches. This configuration can be done via Licensing Façade Configuration Dialog. This dialog can be invoked using the Edit | Licensing Façade Config menu item at the Products Screen.

Note

Although there is an ability to specify database connection attributes there, these attributes can only be used by specific plug-in implementation. These attributes provide necessary how-to for database connection for the plug-in. It is sole responsibility of application developers to design actual code to get a database connection and to utilize database to handle any licensing tasks.

Please consult "Protection! User Guide" for more information about the process of configuring Protection! Web Services Application.

11.6.2 Deploying Protection! Web Services Application to Compatible Container

When Protection! Web Services Application is properly configured with the help of Licensing Façade Configuration Dialog it should be deployed to the compatible Web container.

The first step is getting a ready-to-deploy Web application (.war file). This can be done either:

- In Protection! Control Center by invoking the File | Generate WS menu item at the Products Screen.
- By using the command-line Protection! Builder Utility. This utility can be also used in various build systems or scripts to automate the build process.

Built Protection Web Services Application includes the following parts:

1. Protection! Backend Deployment license.
2. Products Storage.
3. Backend and Web Services Configuration.
4. Core libraries.

The next step is deploying the configured application to a compatible Web container e.g. to Apache Tomcat 5.x. Please refer to documentation of the corresponding Web container to learn how to deploy Web applications. When Protection! Web Services Application has been deployed and is up and running it is ready to serve clients by allowing them to get or activate licenses.

The simple test that allows developers to see that Protection! Web Services Application is up and running is done by opening the Licensing Façade URL with the ?wsdl command in your Web browser. For example the following link:

http://services.jproductivity.net:8080/ProtectionWS/services/LicensingFacade?wsdl

shows WSDL descriptor of running Licensing Façade at jProductivity Web site.

The current implementation does not provide any ability to configure an already deployed Protection! Web Services Application. To do so the application should be reconfigured and deployed/re-deployed again. This allows publishing changes of the product storage (e.g. products addition, removal or modifications) or changes of any other options.

11.6.3 Extending Protection! Web Services Application

Protection! Web Services Application provides all of the functionality required to quickly start custom application development. It is possible to prepare and deploy a Web application capable of license generation and activation for custom applications in several mouse clicks and in just a few minutes.

However, it is also possible to extend functionality of Protection! Web Services Application by writing a plug-in which could add specific business logic. Such business logic can include, but is not limited to the following:

- Having multiple accounts identified by "login" parameter. Each account can define access permissions for a particular product.
- Logging all successful and failed attempts to get or activate licenses.
- Deciding whether it is possible to get or activate a certain license for each particular customer.
- Deciding how to deliver generated or activated licenses.

There are two types of plug-ins supported by Protection! – extension and delegate. Writing any types of plug-ins does not require knowledge or understanding of the Web or Web Services development and therefore greatly simplifies plug-ins implementation.

Writing of delegate plug-in requires the entire implementation of the `LicensingFacade` interface. Even though the delegate plug-in allows full control over the process of issuing licenses and their activation, in most of the cases it is impractical to develop such plug-ins.

Writing Extension Plug-in

Extension plug-in should implement the `LicensingFacadeExtension` interface. It allows a quite easy extension of Protection! Web Services Application by allowing contribution to certain required steps of licenses generation and activation processes only.

Default implementation calls extension's methods in the following sequence during the license acquisition:

1. `login(…)` – this method should authenticate and authorize the incoming request.
2. `checkGetLicenseEnabled(…)` – this method allows deciding whether it is possible to get a certain type of license.
3. `onGetLicense(…)` – this method allows deciding whether it is possible to push this license to a particular customer, modifying license attributes before it will be delivered to the customer and logging such an attempt.
4. `deliver(…)` – this methods allows deciding how to deliver the generated license to the customer.

Default implementation calls extension's methods in the following sequence during the license activation:

1. `login(…)` – this method should authenticate and authorize incoming request.
2. `checkActivateLicenseEnabled(…)` – this method allows deciding whether it is possible to activate a certain type of license.
3. `onActivateLicense(…)` – this method allows deciding whether it is possible to activate this license for a particular customer, modifying license attributes before it will be delivered to the customer and logging the attempt.
4. `deliver(…)` – this methods allows deciding how to deliver activated license to the customer.

Note execution of the sequence above will be stopped if any method calls are failed.

login(…) Method Implementation

This method authenticates and authorizes the attempt to get or activate a license. Caller credentials (login and password) should be checked to ensure that the caller is known to the system. License information can be used to ensure that the caller has rights to get or activate a certain type of license.

It should return a session object if the caller is authorized to make the requested operation or `null` otherwise. Therefore the session object by default holds the specified login, arguments and operation. Custom implementations can return custom session objects that could carry out any additional information.

Simple method implementation may rely on login name and password passed in configuration. In this case the code may look like the following:

```
public LicensingFacadeSession login(LicenseBase aLicenseInfo,
  String aLogin, String aPassword, Map anArguments, int anOperation)
{
  // obtain façade passed to the init(…) method
  LocalLicensingFacade f = getLocalLicensingFacade();

  return f.isLoginEnabled(aLicenseInfo, aLogin, aPassword) ?
    newLicensingFacadeSession(aLogin, f.getSerialNumber(aLicenseInfo),
      anArguments, anOperation) : null;
}
```

The real-life implementation probably should deal with some persistent storage e.g. database to get authentication and authorization data.

checkGetLicenseEnabled(…) Method Implementation

This method checks whether it is possible to get licenses according to the specified license information and operation code (stored in the session). Simple method implementation may rely on permissions passed in configuration. In this case the code may look like:

```
public boolean checkGetLicenseEnabled(LicensingFacadeSession aSession,
  LicenseBase aLicenseInfo)
{
  // obtain façade passed to the init(…) method
  LocalLicensingFacade f = getLocalLicensingFacade();

  return f.isGetLicenseEnabled(aSession, aLicenseInfo);
}
```

The real-life implementation probably should analyze some authorization data stored in the session during execution of `login()` method.

checkActivateLicenseEnabled (…) Method Implementation

This method checks whether it is possible to activate licenses according to the specified license information. Simple method implementation may rely on permissions passed in configuration. In this case the code may look like the following:

```
public boolean checkActivateLicenseEnabled(
  LicensingFacadeSession aSession)
{
  // obtain façade passed to the init(…) method
  LocalLicensingFacade f = getLocalLicensingFacade();

  return f.isActivateLicenseEnabled();
}
```

The real-life implementation probably should analyze some authorization data stored in the session during execution of `login()` method.

checkDeactivateLicenseEnabled (…) Method Implementation

This method checks whether it is possible to deactivate licenses according to the specified license information. Simple method implementation may rely on permissions passed in configuration. In this case the code may look like the following:

```
public boolean checkDeactivateLicenseEnabled(
  LicensingFacadeSession aSession, String aLicenseNumber,
  String aProductID)
{
  // obtain façade passed to the init(…) method
  LocalLicensingFacade f = getLocalLicensingFacade();

  return f.isDeactivateLicenseEnabled();
}
```

The real-life implementation probably should analyze some authorization data stored in the session during execution of `login()` method.

onGetLicense() Method Implementation

This method allows inclusion into the process of the license generation. Implementation of the method can:

- Change any license attributes.
- Break license generation by throwing an exception due to system error or misconfiguration.
- Decide whether it is possible to continue with the license generation process or should the process be broken by providing a meaningful message and/or the `LicensingFacadeResultBase.RESULT_ERROR` for the method result.
- Specify how the license should be delivered by identifying delivery type for the method result.

The following code snippet shows how this method would be implemented. It adds code to check whether a license has not been already issued for a particular customer and to log the issued license on success. This can be done by introducing `isLicenseAlreadyIssued(…)` method that can inspect previously issued and logged licenses, by querying database and by calling this method as part of `onGetLicense(…)` method implementation, for example.

```
protected void logIssuedLicense(LicenseImpl aLicense, Customer aCustomer)
{
  /** @todo add code to log issued license to database */
}

protected boolean isLicenseAlreadyIssued(LicenseImpl aLicense,
  Customer aCustomer)
{
  /** @todo add code to check database for license issuing */
  return false;
}

public LicensingFacadeResultBase onGetLicense(LicensingFacadeSession
  aSession, LicenseImpl aLicense, Customer aCustomer)
{
  if (isLicenseAlreadyIssued(aLicense, aCustomer))
```

```
  {
    return new LicensingFacadeResultBase(
      LicensingFacadeResultBase.RESULT_ERROR,
      "The requested license has already been issued");
  }
  else
  {
    logIssuedLicense(aLicense, aCustomer);
    return new LicensingFacadeResultBase(LicensingFacadeResultBase.
      RESULT_OK);
  }
}
```

onActivateLicense(…) Method Implementation

This method allows inclusion into the process of license activation. Implementation of the method can:

- Change license attributes.
- Break license activation by throwing an exception due to system error or misconfiguration.
- Decide whether it is possible to continue with the license activation process or should the process be broken by providing meaningful message and/or the `LicensingFacadeResultBase.RESULT_ERROR` for the method result.
- Specify how the license should be delivered by identifying delivery type for the method result.

The following code snippet shows how this method would be implemented. It checks if the license was not activated before or if the number of activations does not exceed the number of copies allowed and finally logs activated license on success.

```
protected void logActivatedLicense(LicenseImpl aLicense,
  Customer aCustomer)
{
  /** @todo add code to log activated license to database */
}

protected boolean isLicenseAlreadyActivated(LicenseImpl aLicense,
  Customer aCustomer)
{
  /** @todo add code to check database for license activation records */
  return false;
}

public LicensingFacadeResultBase onActivateLicense(
  LicensingFacadeSession aSession, LicenseImpl aLicense,
  Customer aCustomer, String anActivationKey)
{
  if (isLicenseAlreadyActivated(aLicense, aCustomer))
  {
    return new LicensingFacadeResultBase(
      LicensingFacadeResultBase.RESULT_ERROR,
      "The license has already been activated");
  }
  else
  {
    logActivatedLicense (aLicense, aCustomer);
    return new LicensingFacadeResultBase(LicensingFacadeResultBase.
      RESULT_OK);
```

```
  }
}
```

deliver() Method Implementation

This method provides support to deliver issued or activated licenses to the customer. The current implementation provides two types of delivery:

- *Online* – when the license is returned to the caller as an array of bytes.
- *E-mail* – when the license is sent to the customer's specified e-mail address.

Method implementation should write the license to an array of bytes and assign it to the method's result for an online delivery type. For other delivery types the license should be distributed to the user by some other methods (e.g. by sending the license as an attachment to the e-mail message). In such a case a meaningful message should be specified for the method's result to instruct the user on how to find the delivered license (e.g. "Please check your mailbox for a license").

Let's add sample code to include a custom subject and text for e-mail messages used for delivery of licenses by e-mail. This can be done by adding code for sending licenses by e-mail with a custom subject and text.

```
protected String getLicenseMessageSubject()
{
  /** @todo add actual implementation here */
  return "License is here";
}

protected String getLicenseMessageText()
{
  /** @todo add actual implementation here */
  return "Please see your license attached";
}

public LicensingFacadeResult deliver(LicensingFacadeSession aSession,
  int aDeliveryType, License aLicense, Customer aCustomer)
{
  LicensingFacadeResult result = null;

  LicensingFacadeDelivery delivery = getLocalLicensingFacade().
    getDelivery(aDeliveryType);
  if (delivery == null)
    throw new IllegalStateException(
      "Unable to obtain delivery for delivery type: "+aDeliveryType);

  if (delivery instanceof EmailLicensingFacadeDelivery)
  {
    EmailLicensingFacadeDelivery emailDelivery =
     (EmailLicensingFacadeDelivery) delivery;
    try
    {
      if (emailDelivery.sendLicense(aLicense, aCustomer,
          getLicenseMessageSubject(), getLicenseMessageText()))
        result = new LicensingFacadeResult(LicensingFacadeResultBase.
          RESULT_OK, null,
          emailDelivery.getSentMessage(aCustomer));
    }
    catch (Exception ex)
    {
```

```
      ex.printStackTrace();
    }
  }
  else
  {
    result = delivery.deliver(aLicense, aCustomer);
  }

  return result;
}
```

Extending LicensingFacadeExtensionSupport

While it is possible to directly implement the `LicensingFacadeExtension` interface it is more practical and is recommended to extend `LicensingFacadeExtensionSupport` class instead. `LicensingFacadeExtensionSupport` provides default implementation for most of the plug-in methods in the way similar to the ones described in the previous topics and as a result allowing overriding only required abstract methods.

Logging Licensing Activities

Protection! Backend logs licensing attempts, methods calls and results etc. by writing them to files, console or to other supported destination.

Note that such logging capability is not considered to be a part of custom business logic. Rather, it provides information for system administrators about what, where and when occurred in the backend. In most of the cases such information can be used to find and analyze various problems.

It is possible to add custom logging to specific plug-in implementation by calling suitable methods of `ProtectionLogger` interface. Default instance of `ProtectionLogger` can be obtained using the `LocalLicensingFacade.getLogger()` method.

By default the Java Logging API is used in implementation of `ProtectionLogger` interface. Custom plug-in implementation can obtain default logger by `LocalLicensingFacade.LOGGER_NAME` name and configure it by specifying level, handlers etc. For example:

```
Logger logger = Logger.getLogger(LocalLicensingFacade.LOGGER_NAME);
Logger.setLevel(Level.ALL);
```

However, developers are free to make their own implementations (e.g. by employing Log4J framework). In such a case a new `ProtectionLogger` instance should be specified to the Protection! backend by call of `LocalLicensingFacade.setLogger(…)` method.

Deploying Plug-in Implementation

To deploy plug-in implementation to Protection! Web Services Application the factory class responsible for plug-in instances creation should be implemented. The factory class should implement `LicensingFacadePluginFactory` interface as listed in the sample code below.

```
public class LicensingFacadePluginFactoryImpl
  implements LicensingFacadePluginFactory
{
  public LicensingFacadePlugin create(LocalLicensingFacade
    aLicensingFacade)
  {
```

```
    LicensingFacadeExtensionImpl result =
      new LicensingFacadeExtensionImpl();
    result.init(aLicensingFacade);
    return result;
  }

  public void release(LicensingFacadePlugin aLicensingFacadePlugin)
  {
  }

  public void init()
  {
  }

  public void close()
  {
  }
}
```

When the plug-in implementation and factory are ready, they should be compiled and deployed to a .jar file. Now the plug-in is ready to be included in the Protection! Web Services Application. This can be done via specifying plug-in factory class, plug-in library (.jar) and any other libraries required by plug-in implementation using the Plug-In page of the Licensing Façade Configuration dialog in Protection! Control Center. After specifying the plug-in properties, Protection! Web Services Application should be exported to a .war file and deployed to a Web Container as described in the previous topics.

11.7 Deploying Protection! Backend Applications

To deploy Protection! Backend (Server) application, a deployment license must be purchased. For more information on purchasing a Protection! Backend (per CPU) deployment license, see http://www.jproductivity.com/products/protection/buy.htm

To the extent that you are allowed to redistribute third-party redistributables under the terms of your license, you may redistribute these classes/resources in any of the formats subject to the restrictions in the third-party license agreement. Refer third-party-license.html for the details regarding redistribution.

11.7.1 Protection! Library Redistributables

Only the following libraries may be deployed with your application when you purchase a Protection! Server/CPU deployment license:

1. *Protection.jar* - Protection! core.
2. *ProtectionPriv.jar* - Protection! backend core.
3. *bcprov-jdk14-136.jar*–Bouncy Castle Crypto API.

To the extent that you are allowed to redistribute redistributables under the terms of your license, you may redistribute these classes in any of the following formats subject to the restrictions in the license agreement.

1. Protection! API
   c) As the entire *Protection.jar* and *ProtectionPriv.jar* file.
   d) As content of *Protection.jar* and *ProtectionPriv.jar* bundled (and optionally obfuscated) into your application archives.
2. Bouncy Castle Crypto API

b) As the entire *bcprov-jdk14-136.jar* file.

11.7.2 Redistributables to Allow Support for License Delivery via E-mail (Optional)

For the purpose of the license(s) included with this product, the redistributables to allow support for license delivery via e-mail mechanism and support for e-mail template processing is defined as the activation.jar, mail.jar, and freemarker.jar files located in the /<install_dir>/lib directory.

To the extent that you are allowed to redistribute redistributables under the terms of your license, you may redistribute activation.jar and mail.jar in any of the following formats subject to the restrictions in the license agreement:

a) As the entire activation.jar and mail.jar file

To the extent that you are allowed to redistribute redistributables under the terms of your license, you may redistribute freemarker.jar in any of the formats subject to the restrictions in the third-party license agreement.

Chapter 12

Frequently Asked Questions

General questions

How do I implement Protection! with applications running on a GUI-less, server environment?

It is quite easy to embed Protection! into any application. The developer can use the following steps:

1. Create a new product in Control Center using the Edit Product Dialog, and specify the product's editions, features and other attributes.
2. Specify the resource folder attribute for your product using the Edit Product Dialog | Locations | Resource Folder edit box to let Protection! know where to find the license in your application archive (e.g. /com/acme/app)
3. Use "Headless" code snippet to get a template of the class that should be used to embed Protection! in your server application.
4. Modify the generated code to execute actions appropriate for certain events. Here is a simple example:

```
private LicenseAdapter licenseListener = new LicenseAdapter()
{
  public void licenseOk(LicenseHost aSource, License aLicense);
  {
    enableMyFunctionality(true);
  }
  public void licenseExpired(LicenseHost aSource, License aLicense)
  {
    enableMyFunctionality(false);
  }
};
```

In this example, you have some functionality disabled by default and you enable it only when the license is valid and is not expired.

5. Add a call of `checkLicense()` method somewhere in your code during application startup, and schedule a call of it each new day (hour, minute...) to ensure that the license is still OK.
6. Generate a new evaluation license, make an application archive (e.g. .jar, .war, etc...) and include application classes and license file into it.

That's it! Now you can deploy your application on your Server, and it will work until the date stated in the bundled license. When your customer purchases a commercial license he/she should simply copy it to the HOME folder to get the application licensed!

Is activation with lock limited to the platforms that have native libraries provided?

If you use the default implementation provided by Protection! - then Yes. However, you are free to provide your own implementation. Protection! Professional edition does provide developers with the ability to lock a license to a specific system. The default implementation allows you to lock the license to a network card MAC address. However, the developer/publisher is free to lock the license to any other user-definable attribute.

For example: You can lock the license to the user's host address by making a subclass of `LicenseHostPro` and overriding the `getActivationLockKey()` method to employ any other methods/lock mechanisms/etc...
The following code snippet illustrates how to employ a node's IP address to be used as activation and lock key:

```
public class LicenseHostProEx extends LicenseHostPro
{
  protected String getActivationLockKey(LicenseImpl aLicense)
  {
    return getActivationKey(InetAddress.getLocalHost().getHostAddress().
      hashCode());
  }
}
```

How do floating licenses work?
Protection! sends a UDP broadcast (Firewall friendly) message on a specific port, and with a specific signature. At the same time, Protection! listens and collects responses. Response will contain the license number and other important license attributes. As responses are collected, Protection! counts how many licenses are allowed and how many licenses are in use. If the number of licenses in use is greater than allowed by the floating user license, then Protection! fires an appropriate event.

You should override the `numberCopiesViolation(…)` method in your implementation of the `LicenseListener` interface to listen and properly handle such situations.

Please note: Protection! is a framework, and it will provide you with all necessary mechanisms and sometimes their default implementations. As a developer, however, you need to tell in your implementation code what should happen with your application when Protection! tells you that certain events/conditions have occurred.

What if on-line activation is not possible?
Please see the following simple code snippet that allows generation of the activation key. This snippet can be embedded or used to generate the activation key. The generated activation key can then be emailed back to the original developer/publisher, along with the original license for further generation of the license with the "activation and lock" option.

```
public class ActivationKeyGenerator
{
  // introduced subclass of license host. Note it MUST be used for both
  // activation key generation
  // and usual use to provide licensing
  public static class LicenseHostProExt extends LicenseHostPro
  {
    // overriden to always return MAC address based activation key
    public String getActivationKey()
    {
      return getActivationLockKey("");
    }
  }

  // simply prints activation key
  public static void main(String[] args)
```

```
  {
    LicenseHostProExt licenseHost = new LicenseHostProExt();
    System.out.println(licenseHost.getActivationKey());
  }
}
```

Please note: The above code snippet is suitable for "headless" environment. For GUI based applications, the Protection! License Activation Wizard should be used to resolve license
activation issues.

How do I implement Protection! with a product that has SDK and Runtime libraries?

Q. *We are deploying our products as:*

2. SDK - The customer develops applications which are linking with our libraries
3. Runtime - The customer purchases a runtime license to run the applications they created with our SDK
4. Applications - The applications that we created with our software base
5. How we going to differentiate Runtime and SDK distributions?

A. The schema could work as follows:

- For your SDK - you could create a product in the Protection! Control Center with 2 (two) editions - API and Runtime. You would then generate a license for the API edition and would implement license reading and validation somewhere in either/or both - initialization method, a specific API call, etc. In your API license, you would set some limitations like expiration date, number of copies, etc. In your Runtime/Deployment license, such limitations would go away.
- For your Applications - if they are using your SDK, then it is probably a good idea to make your application see the SDK license. This way you would not need to generate two or more licenses for an application. However, you can also have two or more licenses if desired - one for the SDK and one for the application.

Can Protection! be integrated with online automated systems?

Protection! would allow you very smooth and seamless integration with almost any backend system by either using Protection! Web Services (default implementation) or by writing your own backend implementation (RMI, EJB, etc). One of the major points that separates Protection! from its competitors is that Protection! is a framework and therefore gives you as a developer/publisher all the necessary tools and methodologies, while not imposing any limitations or having any "underwater stones". You can use and embed Protection! into your application in as little as 5-10 minutes (via automatically generated for your product and ready to use java implementation files) or extend Protection! to fit your specific needs.

There are many security models that could be implemented with the aid of Protection! These models could be either manual (require manual license generation) and/or automated where a license can be generated by the Protection! backend automatically. With automatic license generation, the license can either be deployed to your customer immediately or can be stored in some queue for further review and approval by sales/management before distribution). You can also tie Protection! backend to your sales-force application that processes your orders, so that the backend will query if a valid purchase was recorded before the commercial license is activated. As you can see, these models could be as complex or as simple as you want them to be. Protection! gives you 100% flexibility to implement any of them by being a Framework, and by not imposing any of the models on you as developer/publisher.

In general, for a GUI application you could follow these steps:

1. Vendor generates a license that requires activation and lock
2. Vendor provides this license to a customer
3. Customer copies the license file to the right place manually, or starts the application and uses the Licensing Wizard to specify the license file location
4. Customer gets License Activation Wizard prompting him/her to activate the license If offline activation is chosen, then he/she should provide the Activation Key shown in the Wizard back to the vendor by e-mail or phone
5. Vendor re-generates the license, turning on activation and lock and places supplied by customer Activation Key in the "Activation Key" box in the Protection! Control Center
6. Vendor provides the activated license to a customer.

If Protection! Backend is used, then steps 5-6 will be handled by Protection! Backend automatically.

How can I implement an efficient "Secret Storage"?

Protection! provides a file-based implementation of secret storage that has no native code dependencies, and therefore will work on any platform. You can use several instances placed in different locations e.g. one based in the user's HOME folder, another one in some unique system folder, etc. Having several secret storage files can be enough in most of the cases. As an alternative, you can implement your secret storage using standard Preferences API to hold data in a more-or-less secret location (e.g. in Registry under Windows).

Actually, the only time you should be seriously concerned about efficient secret storage design/implementation is when you provide a bundled evaluation license, and therefore allow a flexible expiration date (the flexible expiration date is set based on the first usage of the application). In this case, if customer locates and removes all secret storages of your application, he can continue to use the evaluation indefinitely.

If no bundled license is used, and therefore you will provide evaluation licenses personally to each of your customers, the only way to trick your application is by playing with the system time. Certainly it is possible in some rare situations, but it is completely inappropriate for server and enterprise based applications as it can cause a series of other server applications to behave incorrectly.

How does Protection! recognizes the expiration date if the application is already running?

If your protected application is currently running and/or in constant running state (i.e., server based application) you should perform license checking procedures at some predetermined/scheduled time interval.

How does Protection! handle various events fired during the license reading/validation processes?

During license checking and validation, Protection! Licensing Toolkit provides detailed feedback about the process status and its results back to the application. This feedback is provided by firing an appropriate set of events. It is the responsibility of the application developer to decide which action should be taken in response to a particular Protection! event. The default implementation assumes no actions for any such events provided by Protection! Licensing Toolkit. For example: when it is determined by Protection! that the license file is invalid, the application can either exit or the application could disable all or part of its functionality. Such a decision is the responsibility of the application developer (there is no default behavior provided by Protection! Licensing Toolkit). In addition to the event based analysis/action, it is always possible to get the current status of the license at anytime during an application's run/lifecycle, to see whether the license is valid.

While the default implementation of the license checking mechanism provides functionality that would be enough for the majority of applications, developers can easily override/extend any license reading/validating procedures to get the desired results.

Would Protection! detect a "number of copies violation" if the application is running on two different networks?
Probably not! It depends on the network settings/configuration. If firewall(s) or router(s) are instructed to filter broadcasting, then this approach will not work. While the current default implementation of the "number of copies violation" detection mechanism has some well known disadvantages, it requires zero-administration and eliminates additional costs of buying and hosting a licensing server.

Can Protection! be integrated into an app server environment?
Yes, Protection! can be easily integrated into an application server environment. Protection! Pro also provides:

- Powerful default back-end implementation that gives you a foundation for building custom back-office system integration
- Plug-in support to allow easy extension of default back-end implementation
- Ready-to-deploy Protection! Web Services application

Please look at the "Licensing Facade Configuration Dialog" section in the Protection! User Guide for more information.

Does the secret storage file get deleted during uninstall of the protected application?
This depends on how you build your installation. If you deploy your secret storage during the installation and you use Protection! file-based default implementation, then probably every installer package on the market today would allow you to exclude certain files from the uninstaller script, so the file would remain on the user's system. This said, you can generate secret storage dynamically after the installation and during the first usage of the application (see the sample DemoCalc that comes with Protection! for an example of such an implementation).

Is Protection! secret storage only file or Java Preferences API based?
No. The secret storage is a mechanism which used to provide a way to persistently store various information about the application and any other information (if necessary). The default secret storage implementation is file or Java Preferences API based. However it is up to you as a developer to implement a desired secret storage mechanism of your own, if so desired (e.g. Windows Registry based).

Can someone else who bought your product generate licenses for our products?
Without having product storage there is no such possibility for anyone who also has Protection! to generate licenses for someone else's products. Similarly, you would not be able to generate license for the Protection! Control Center yourself.

Is Protection! compatible with obfuscation products?
Protection! co-exists very happily side-by-side with code obfuscation.
In general, you should embed all Protection! redistributable classes inside your application archive, which in turn should be obfuscated using your favorite Java byte code obfuscator.

Will the Encryption Key Bytes ever change once the product has been created?
No. Once the product has been created in Protection! Control Center the Key Bytes will not change for a given product under any circumstances.

Is it possible to define the product editions, features, etc. in a relational database?
In theory, it is possible to define the product editions, features, etc. in the relational database, along with the key used to generate the licenses, and use the API to pass that information in to generate the license.
You can obtain products from the products storage and save their properties to a some database. Later, when you'd like to read or generate licenses you can create a product instance, fill its properties from the database, and use it. Please check classes in the `com.jp.protection.priv` and `com.jp.protection.priv.products` packages to get full access to the required low-level API.
However, the most simple way to do this is by using BLOB's in your database (if supported). In this case, you can simply save and load products storage to and from DB using a BLOB field. `ProductsStorage` class provides the following methods suitable for such a task:

```
public boolean save(OutputStream anOutputStream) throws IOException
public boolean load(InputStream anInputStream) throws IOException
```

Is it possible to generate a commercial license with an expiration period?
You can generate a commercial license for you customers with the specified expiration period. At a time of license reading and validation, you would check for an expiration period and would (or would not) check for the application version.
This allows some software publishers to implement "rental" licensing mechanism. In this case, your client is renting your application/library for a mutually agreed renting/leasing period (this would indicate license expiration period span).

Where can I find Javadocs for Protection! API?
You can find Javadocs in *<install_dir>/doc/ProtectionDoc.jar*

Why doesn't Protection! provide any error messages?
Suppressing error messages is by design, as having any output during license reading and validation can effectively reduce efforts required to break the protection. That's why there are no exceptions shown during the process. But you can change this behavior for debugging purposes by calling `setVerbose(true)` methods for license reader and host.

Can Protection! be integrated with the installation tools
You can take the following approach:
1. Use GUI code snippet that is generated by Protection! Control Center.
2. Build it and add it to your Installation tool of choice to be executed as custom code (see your installation tool documentation on how to include/execute custom code).
3. Call it (ProtectionSupport code) from within your Installation tool where you think it is appropriate.
4. Call to ProtectionSupport class would initiate standard Protection! Licensing Wizards and will aid your users in obtaining the license.
5. When your user would launch your application, the license would already be obtained (previous step).

What is the encryption size used for a Protection! license?
Protection! uses 256 or 512-bit RSA encryption.

Are you using RSA key pair?
Yes

Do you generate a key pair for each product?
Yes

Does Protection! support a license with a flexible expiration date?
Yes, Protection! will allow you to generate a license key having a flexible expiration date - meaning that the actual expiration date would be set when the user launches your application for the first time. The expiration date would be set for a number of days (specified by you) from the first usage of your application.

Protection! allows you to bundle a license with your application. You should specify the license location options using the License Location group in the Location tab of the Edit Product dialog. You should then specify license file name, folder to find the license in the local file system and resource folder (e.g. */com/jp/samples/protection/*) to find the license within your application archive. By default, Protection! tries to find license in the local file system. If the license is not found, then Protection! tries to find the license in the application archive. This allows for bundling evaluation licenses with your application (e.g. for CD distribution). When a customer buys a commercial license, he/she should place it into the specified folder or use the help of the Protection! Licensing Assistant to do so.

Please use the following method `setAllowFlexibleExpirationDate()` to specify whether the expiration date of a license should be adjusted according to the current date and duration of the evaluation period.

Does Protection! prevent the end-user from decompiling the Java byte code?
It is not the responsibility of Protection! to prevent de-compilation. However, Protection! does prevent potential malicious application patching via the Protection! Integrity support module.

Do you consider Protection! to be un-crackable?
One of the major things that differentiates Protection! from a small number of competitive products is that Protection! is a framework. Therefore, you, as a developer/publisher not only have *full* control of design and implementation of your licensing solution, but it is also totally up to you on how well and/or strongly you will protect your application. Please keep in mind that there is always a fine-gray-line between strong protection and user-friendliness of your application. On a different note - We do not think anyone can honestly say (and if anyone would - it would be a lie) that something is un-crackable. However, this is a more theoretical and philosophical question for debate :)

How much time would it take to implement Protection! in my application?
Protection! has an extremely fast learning curve. With the help of the Protection! tutorials and provided examples, a developer would be able to write protected application in a matter of hours.

Can a License be created from the remote location?
Yes - this could be done with the aid of Protection! Backend via WebServces (default implementation) and/or EJB, RMI, etc.

Why are activation, serial number, and other things missing from the License Activation Wizard dialog?
You probably did not create and register `LicensingFacadeProvider`. Therefore, Protection! does not know how to get a license from a remote host, etc...

Please look at the DemoCalculator (that comes with Protection!) source code

<install_dir>\samples\src\com\jp\samples\protection\DemoCalcProtectionSupport.java

for `initLicensingFacade()` method.

How can I enhance the license checking procedure?
Please look in your *ProtectionSupport.java* file (generated code snippet):

```
private DefaultLicenseAdapter
  licenseListener = new DefaultLicenseAdapter(owner) {…}
```

has a lot of methods such as `licenseCorrupted(…)`, `licenseExpired(…)` etc. You can override any of these methods to handle your situations. For example if the license is expired you can invoke `System.exit()`, etc.

What is the Custom License Agreement option?
Please note that this section is reserved for *TRULY* custom License agreements. In normal usage, you would specify the License agreement in your code through the following methods (this is also generated by Protection! Code snippets):

```
licenseHost.setLicenseTextCommercial("Commercial license agreement text");
licenseHost.setLicenseTextEvaluation("Evaluation license agreement text");
```

You should use the custom license agreement in situations when you need to create a special agreement for one customer in some special case only!

Please note: Custom License agreement is encoded into the license key. Therefore if your custom license agreement is large - your generated license key would also be increased in its size.

Why am I getting an expired license message when trying to run the Demo application?
Possibly the license expired at some point in the past. If this is the case, then the expiration flag was written into the secret storage - this prevents your users from downloading evaluation licenses and using them after the expiration date. Either generate an extended evaluation license, or delete your secret storage and run your application again with a regular evaluation license.

How does the extended evaluation license work?
Extended evaluation ignores any previously set expiration time, and therefore it allows you to extend the evaluation period for eligible customers. If and how many times you and/or your backend will issue extended evaluation license is all up to you as a developer/publisher. A user could request extended evaluation license ad infinitum, but you (if you issue licenses manually) and/or your backend implementation would need to make a decision based on your business practices if you allow extended evaluation to be requested at all/more then once/how many times/ etc...

What is the file size overhead for embedding Protection! libraries into our application?
This is depends on the Protection! Licensing Toolkit functionality that you are planning to utilize in you application. For a description of the deployment libraries please see *<install_dir>\lib\deploy.html.* In case you are only deploying Protection! Library Redistributables, you would need to deploy Protection.jar and bcprov-jdk14-136.jar only (~2.5 MB).

Are native calls always used even when I do not check the MAC address?

No. You need to provide native libraries only if you are planning to utilize the "named-user" licensing model with the ability to lock a license to a specific computer system using MAC address of network card a one of the lock attributes.

Is it possible to provide default user details with the application?
You can do this using the following scenario:
On Publisher Side: put customer details into the license properties e.g. via specifying customer attributes for the license using the License Screen.

On Client Side:
1. Read the license.
2. Create an instance of the Customer class and specify its properties based on the properties you put into the license.
3. Write customer data using

```
customer.toPreferences(LicenseUtils.getCustomerPreferences(license.
  getProduct()));
```

Now, customer data will be shown in the Protection! About Dialog.

Why does the license checking process not fully complete sometimes?
It seems the issue is caused by missing native libraries. Such libraries are required to obtain the network card address used to lock a license to particular computer. To see any exceptions, you should turn on verbose mode of `LicenseHost` by calling `setVerbose(true)` method. Please consult the *<install_dir>lib/deploy.htm* for more information.

What part does the FileSecretStorage.setDirty() play?
It allows you to set Secret Storage mode when changes were made since the last load or creation. If secret storage is not modified (not dirty) the `save()` method simply does nothing.

Can multiple Secret Storage files be kept in-sync?
Yes, it is possible by using the `LicenseHost.saveSecretStorageProperty(…)` method.

When should I call the SecretStorage.save() method?
Q. *Should I manually call `SecretStorage.save()` after a call to `SecretStorage.setProperty()` or is it called automatically after a property has changed?*
A. You should manually save Secret Storage by call `SecretStorage.save()` when you think it is appropriate.

Should I implement the code for checking the number of copies myself?

You should override `numberCopiesViolation(…)` method in your `LicenseListener` implementation and write code that will make the appropriate actions to handle this case.

Please note: Protection! is a framework - it will provide you with all necessary mechanisms and sometimes their default implementations. However, you as a developer need to tell in your implementation code what should happen with your application when Protection! tells you that certain events/conditions have occurred.

Does Protection! provide the ability for a product to be locked to a specific system?

Yes, Protection! Professional edition does provide developers with the ability to lock a license to a specific system. Protection! provides support for a variety of licensing models. The "named-user" licensing model will allow you to lock the license to a network card MAC address (default implementation) or other, user-definable attribute.

How would Protection! deal with re-installation of a protected application and evaluation license?
This is controlled by the Secret Storage file(s). Because the Secret Storage file is being generated in the first use of your application, the uninstaller would never remove this file because the uninstaller does not know anything about this file. It is up to you where and how you implement Secret Storage. The simplest way is to put it in user HOME directory. However, you can also modify the Registry, create multiple secret storage files, etc, etc.

Is it possible to use Serial Numbers without contacting software developer / publisher?
No, this is not possible as Serial Numbers represent a very small subset of the license and it is designed so you can deploy your application and supply a Serial Number on a CD.

How can I test numberCopiesViolation?
Such test needs to be performed on multiple copies on two or more different systems. If you are testing multiple copies on same system - this would never work because of the same computer.

How do I specify custom properties for a given license?
The easiest way is to create custom properties in Control Center (see Properties Tab - Control Center License Screen). Give this property some descriptive name like "Connection Number" (or whatever you want it to be) and set it to a desired value. In your code, after you read a license, use `getProperty(…)` method to read your property and its value. Then you can specify how your application would behave if the property value is outside of its expected range.

How can I include line breaks in the license agreement text?
Q. *I cannot seem to include new lines in the license agreement text. I tried \n and \r\n but only get a continuous string that wraps.*
A. Because the License Agreement is treated as HTML, you should use HTML tags to brake your text (e.g.,
</br>; , etc). Using HTML, you can also implement better formatting of your text vs. plain ASCII text.

Is Protection! multi-platform?
Protection! Licensing Toolkit is multi-platform. Protection! is tested and verified for Microsoft Windows NT/2000/XP/Vista, Solaris/x86, Linux (Red Hat/SuSE) and Mac OS X. However, it should work on any Java-Enabled platforms.

Does Protection! show exceptions, if any, during license reading and validation?
No. To see exceptions, if any, you should turn on verbose mode of `LicenseHost` and/or `LicenseReader` by calling `setVerbose(true)` method.
Please note: This should be set for testing purposes only and should be turned off for production.

How can I specify various environmental settings within the license?

Protection! provides you as a developer/publisher with ability to specify an unlimited number of custom properties (key/value pairs). Such properties could be used at any point of license reading/validation in order to allow/disallow some (or all) of you application functionality.

Why am I getting java.lang.NoClassDefFoundError exception?

The following exception:

java.lang.NoClassDefFoundError:
org/doomdark/uuid/NativeInterfaces ...

is caused by missing libraries needed to support "named-user" licensing model. You need to provide *<install_dir>/lib/jug.jar* and native libraries if you are planning to utilize the "name-user" licensing model with the ability to lock a license to a specific computer system. For description of the deployment libraries please see: *<install_dir>/lib/deploy.html.*

Protection! Back-end and Web Services

How do I manage and track generated licenses?

Protection! is not responsible for such tasks. However, you can achieve this by writing a plug-in to Protection! Web Service default implementation.

Protection! provides default implementation of `LicensingFacade` as a Web Services application. Certainly you are free to make any other implementation suitable to your requirements e.g. RMI or EJB based. Licensing Facade Config Dialog provides the ability to configure Protection! WS application so you don't need to play with WS or servlets directly. You should just specify all the options and deploy them along with the product's storage to the Web application using the File | Deploy WS menu in the Products Screen. As a result, you'll get ready to deploy Web application capable of license generation and activation through the use of Web Services. This allows you to quickly and easily start development of a protected application.

While the default implementation provides a good start, you probably need to extend it to have the ability to log issued licenses and/or decide when it is possible to issue or activate a particular license. It is possible by writing a plug-in to the default implementation. Plug-ins should be implementations of either: `LicensingFacadeDelegate` or `LicensingFacadeExtension` interfaces. You should also provide implementation of `LicensingFacadePluginFactory` responsible of plug-in creation. Then you should pack all your classes to the general use archive and use the Plug-in page of the Licensing Facade Config Dialog to specify factory class, plug-in archive and all other libraries required by plug-in to work (e.g. JDBC driver). Finally, you should put this config to the WS application and deploy to the Web container. It is also possible to specify database options for convenience of database plug-ins development.

Does Protection! Web Service perform any kind of logging?

Yes, Protection! Web Service provides logging using default Log4J based implementation.

Why do I get an exception when trying to connect to Protection! Web Service?

The following exception:

```
Exception occurred during event dispatching:
java.lang.NoClassDefFoundError: org/apache/axis/AxisFault
 at
```

```
com.jp.protection.pub.pro.integration.ws.LicensingFacadeWS
...
```
indicates that Apache Axis libraries are missing. You need to add Apache Axis libraries to your deployment. For description and usage of the deployment libraries please see *<install_dir>\lib\deploy.html*.

With the deployed back-end service, is Protection! able to respond to commercial license requests?
Absolutely!

Why am I receiving a corrupted license from Protection! Web Service?
This could be caused if an incorrect or missing license file is specified in Licensing Facade Config Dialog | General Tab | Deployment License. You must specify the location of your commercial or evaluation Protection! License key file (protection.key)

What container can host Protection! Web Service?
Any available web container (free or commercial) supporting Servlet API 2.3 and JSP 1.2 (if you will use JSP) would work.

License specific questions

When a license is generated with the 'activation and lock', why don't I see the License Activation dialog?

It seems the issue is caused by missing native libraries. Such libraries are required to obtain the network card address used to lock license to particular computer. To see exceptions, if any, you should turn on verbose mode of `LicenseHost` by calling `setVerbose(true)` method.

Why does the Protection! About Dialog shows "Unlicensed" for a license value?

The About Dialog shows "Unlicensed" state when no license is available, or when the license has been read but not yet checked. You can get such a result when you call the About Dialog before checking the license. Most probably you are using different LicenseHost's for license checking, and for showing the About Dialog (this is not the best implementation pattern).

What is the general license activation sequence for a GUI application?
1. Vendor generates a license that requires activation and lock.
2. Vendor provides this license to a customer
3. Customer copies the license file to the right place manually, or starts the application and uses the Licensing Wizard to specify the license file location
4. Customer gets the License Activation Wizard prompting him/her to activate the license If offline activation is chosen, then he/she should provide the Activation Key shown in the Wizard back to the vendor by e-mail or phone
5. Vendor re-generates the license turning on activation and lock and places supplied by customer Activation Key in the "Activation Key" box in the Protection! Control Center
6. Vendor provides the activated license to the customer

If Protection! Backend is used steps 5-6 will be handled by Protection! Backend automatically.

How do I 'lock' the license to a Network card's MAC address?

You would use the "Activation and Lock" option when generating the license that you would supply to your customer(s) (or embed into the application). When the customer runs your protected application with this license he/she would see License Activation Assistant Wizard and would be able to provide you with a generated activation key. This key is being generated/calculated based on the customer's MAC address.
If you do not use Protection! backend, then you can generate and provide to your customer an activated license by specifying the received from the customer activation key in the "Activation Key" box in the Protection! Control Center.
The activation process always requires contacting the vendor/publisher either off-line (e.g. by calling/e-mailing sales) or on-line by using Protection! backend. This allows vendors to:

- Track actual application deployments. For example, if a bundled evaluation license requires an activation, then each evaluator would have to provide his/her credentials before using the application
- Implement the Floating User licensing model by enabling usage of only the purchased number of copies by activating only that number of licenses that are specified in the license's number of copies attribute
- Implement the Named User licensing model by locking a license to a particular computer

How does Protection! know that a license file is locked to a MAC address and therefore cannot be reused?
Because a License that is being generated is based on the activation key from your customer. The activation key is unique to the customer's system (calculated based on the customer MAC address - default implementation).
If license activation is required, License Host (Pro Edition only) generates Activation Key by calling `getActivationKey()` method and checks whether such property is already present in the license and if so whether it has the same value. Method getActivationKey`()` actually calls the following methods depending on whether license needs to be locked:

1. `getActivationKey(license)` - generates Activation Key when the license should not be locked. Method's implementation uses concatenation of product and license number hash codes
2. `getActivationLockKey(license)` - generates Activation Key when the license should be locked. The current implementation uses MAC address of the network card to generate this value

If license is moved to a different system, then License host method `licenseLockViolation(…)` can notify listener that the license is activated for use on another computer.

Can Protection! generate specialized keys for our subscribers automatically?
Yes, Protection! can generate specialized license keys. Such license keys could be generated and distributed to application subscribers in several ways:

- Protection! Web Services application could generate a license automatically upon request from the user, and distribute the license either via Direct Internet connection (with support for Proxy) or via e-mail - all of these are controlled by you as a developer
- Protection! could generate either one or multiple Serial Numbers for a given license (e.g. SR84D-VBESQ-GK3RC-F9HM5-ESQGU) You as a developer can also choose if the license, which you provide to the user, needs to be either:
  a) Activated (this would generate a unique activation key for a given license)
  b) Activated and locked (this would lock the license to a specific computer system via the network card MAC address, or some other, specified by your attribute)

c) You can also specify that the user must provide user's info (name, company, e-mail, etc) during product activation

- Protection! has built-in sophisticated and highly customizable Licensing and Activation Assistant Wizards that would aid your users in the process of obtaining new/evaluation/extended evaluation/commercial license and/or activating such a license (if necessary)
- And, of course, you can generate a license manually using Protection! Control Center - with all of the above mentioned functionality and attributes

What is the procedure for sending an activated commercial license?

Q. *What exactly is the procedure? Surely you don't want to send out an activated commercial license without having approved the customer's purchase order.*

A. Probably not, but this is again all up to you as a developer/publisher. There are many models that could be implemented on how and when you would decide to distribute an activated commercial license. These models could be manual (require manual license generation and/or license can be generated by backend automatically, but stored in some queue for further review and approval by sales/management before distribution). You can also tie backend to the sales-force application that processes your orders in order for backend to query if a valid purchase was recorded before the commercial license is activated. As you can see, these models could be as complex or as simple as you want them to be. Protection! gives you 100% flexibility to implement any of them by being a Framework, and not imposing any of the models on you as developer/publisher.

In general, the steps could be as follows - for GUI application:

1. Vendor generates a license that requires activation and lock
2. Vendor provides this license to a customer
3. Customer copies the license file to the right place manually, or starts the application and uses the Licensing Wizard to specify the license file location
4. Customer gets License Activation Wizard prompting him/her to activate the license If offline activation is chosen, then he/she should provide the Activation Key shown in the Wizard back to the vendor by e-mail or phone
5. Vendor re-generates the license turning on activation and lock and places supplied by customer Activation Key in the "Activation Key" box in the Protection! Control Center
6. Vendor provides the activated license to the customer

If Protection! Backend is used steps 5-6 will be handled by Protection! Backend automatically.

Does the "floating" model require that a separate licensing server to be running?

There are two floating models: network broadcast based and Licensing Server based models. The first model does not require presence of the Licensing Server as it simply utilizes local network resources to find other applications running within network.

How does Activation work at a high-level?

With Activation and Lock: Your user, in order to be able to use your application, would have to provide you an activation key that is generated by Protection! and is *specific* to the user's system. When you generate the license key for this user - this license key would only be able to work on this specific user's system. If license is transferred to another system it would throw necessary events during license reading/validation stage such as (in this case):

```
licenseLockViolation(LicenseHost aSource, License aLicense)
```

In this case, the License host calls this method to notify the listener (your application) that the license is activated for use on another computer. You, as a developer, will need to build some logic into your application to handle (this or other license

reading/validation) events. For example, if such event occurs, you can completely disable the application features, or enable only some of them, etc...
Of course, all of the above is applicable if you will use Protection! Web Service to generate and distribute a license key file to your users without your (publisher) interaction.

I like the idea of authenticating the license to a certain machine, but...

***Q**. I like the idea of authenticating the license to a certain machine. However, what happens when the machine is replaced, due to failure or old age? What is the pathway for the user to obtain a valid license for the replacement machine?*

A. This could be handled several ways - however, as you would probably agree, in the majority of the cases this is an administrative (your) decision to issue a new license.
In any case, you can create a generic license that would require activation-and-lock. This license could be either evaluation or limited functionality, with a flexible expiration date (when the expiration date is set based on the first run of your application) - whatever. Then, you embed this license inside your application archive.
In brief - you can embed the license into the application archive as follows: You can Specify the resource folder attribute for your product using the Edit Product Dialog | Locations | Resource Folder edit box to let Protection! know where to find the license in your application archive (e.g. /com/acme/app)
Use Protection! code snippet to get a template of the class that should be used to embed Protection! in your application. Generate a new license; make an application archive containing your application classes and the license file.
That's it! Now you can deploy your application with the bundled license. Please note - you can use both licenses (embedded and located within the file system). This way, you can deploy your application with a bundled evaluation license and when your customer purchases a commercial license, he/she should simply copy their license to your <application_root>/license/ folder, for example.
Back to the question: Let's assume that the user reinstalls his system, changes hardware, etc, etc. - in this case, the user either would need to reinstall your application in which case it would be using an embedded license until the user requests a full commercial one. Or, another example would be if you lock your license to a network card MAC address and the user changes the network card (or if license is transferred to another system). In this case Protection! would throw necessary events during license reading/validation stage such as (in this case):

```
- licenseLockViolation(LicenseHost aSource, License aLicense)
```

Protection! License host calls this method to notify listener (your application) that the license is activated for use on another computer. You, as a developer, will need to build some logic into your application to handle (this one and/or other license reading/validation) events. For example if such an event occurs, you can completely disable the application's features, or enable only some of them, or call Protection! Licensing Assistant Wizard to aid the user on resolving this issue, etc, etc, etc...
Demo Calculator (a sample application that comes with the Protection!) illustrates this and many other examples.

What prevents a user from reusing license keys?

***Q.** What prevents a user from reusing license keys, to install and use an unlimited number of copies of the product?*

- Protection! has several built-in mechanisms to prevent this: Number of copies violation - where you indicate how many copies of your product would be able to run on the network (floating license model).
- Activation and/or activation-and-lock mechanism - where you lock the license to a specific computer system (named-user licensing model)

And of course - because Protection! is a framework you can mix-and-match existing mechanisms, extend them, or implement your own.

Can online licensing be run on a Linux server?
Protection! is 100% pure Java so it will run on any platform that supports Java. We have successful implementations on various flavors of UNIX and Linux, MAC OS X and of course Windows.
The one and only time where there are some platform dependencies is when the "activation-and-lock" mechanism is chosen. Protection! default implementation is able to lock a license to a network card MAC address, and therefore uses some native libraries (provided with Protection!). Of course, if other lock mechanisms are chosen by the developer, then native support may or may not be necessary (i.e. locking license to an IP address of the host will not require native support and therefore would be platform independent).

How fast is the Protection! back-end response to user request?
Q. *With respect to a license file or a serial number that requires activation, if the user decides to 'activate online' will the back-end respond immediately and send out an activated license file/sn?*
A. This would generally depend on when/how/where your back-end is deployed. Even though in general I would answer yes, but you would agree, that there are a lot of variables that could delay distribution of an activated license file. All of these variables are 100% in your control, however. These variable could be your backend business logic (what would happen when the customer requests a license file, would you do this automatically or would you require management/sales approval, would you tie Protection! backend with your sales-force, CRM, ERP applications, etc, etc, etc...) and of course variables like backend server horsepower, network connections, etc, etc, etc...)

Is it possible to transfer a license to another computer system after it has been activated and locked?
No. The license key file would be locked to a specific user system. Default implementation locks the license to a network card MAC address of the user's system. However other, user-definable attributes could be used by you, as a developer, as well.

Is it possible to embed the license file inside an application archive jar file?
You can Specify a resource folder attribute for your product using the Edit Product Dialog | Locations | Resource Folder edit box to let Protection! know where to find the license in your application archive (e.g. */com/acme/app*)
Use Protection! code snippet to get a template of the class that should be used to embed Protection! in your application.
Generate a new license; make an application archive containing your application classes and the license file.
That's it! Now you can deploy your application with the bundled license. Please note - you can use both licenses (embedded and located within the file system). This way you can deploy your application with a bundled evaluation license (for example) and when your customer purchases a commercial license, he/she should simply copy their license to your *<application_root>/license/* folder, for example.

I don't really understand the concept of "activation"
Q. *I don't really understand the concept of "activation" in the license generation panel: what is the benefit of the activation key if finally you have to send a second license to your customer?*
A. The reason for such an approach is to provide secure storage for the activation key and to prevent customers from saving it manually. Everything on the customer's computer needs to be considered insecure (e.g. file system or registry). Therefore, it is very easy to use a special application to track activity of the protected application - find

what it saves, when and where. Our approach of placing the activation key into the license and sending it back to the customer is very safe as the client side is only capable of decoding licenses, therefore making it impossible to place the activation key into the license. This process only looks complicated, however. With the help of Protection! Activation wizards and Protection! Backend it is completely transparent to the customers and vendors.
Activation process always requires contacting the vendor/publisher either off-line (e.g. by calling/e-mailing sales) or on-line by using Protection! backend. This allows vendors to:

- Track actual application deployments. For example, if a bundled evaluation license requires activation, then each evaluator would have to provide his/her credentials before using your application
- Implement Floating User licensing model by enabling usage of only the purchased number of copies by activating only that number of licenses that is specified in the license's number of copies attribute
- Implements Named User licensing model by locking the license to a particular computer

Why does the license return an UNKNOWN_STATE?
This could happen if the developer is simply using License Reader to get a license. License Reader is responsible only for discovery and decoding of the license. Developers should be using License Host in order to read and check a license. License Host uses License Reader, so developers do not need to use (even though they can) License Reader directly.
Please see `licenseHost.checkLicense(...)` method and also demo application that comes with Protection (samples directory) - specifically DemoCalcProtectionSupport.java.

How can I implement the automatic license generation and activation mechanism?
This would require use of the Protection! Backend.

Integrity Check Support

Problems with Integrity checking when packaging is changed
Integrity check uses fully qualified class/resource names:

```
class DemoCalcIntegritySupport
{
  private static final String[] DIGEST_ENTRIES =
  {
    "com/jp/samples/protection/DemoCalcProtectionSupport.class",
    "com/jp/samples/protection/DemoCalcProtectionSupport$1.class"
  };

  public static boolean checkCRC()
  {
      return IntegrityHost.checkStatic(DIGEST_ENTRIES , DIGEST);
  }
}
```

And of course if packaging is changed the code snippet needs to be modified accordingly.

Why did Integrity checking fail after classes obfuscation?

Because obfuscation changes names and packages of the classes in most cases it is not possible to find them and so check their integrity. Protection! provides build-in support of obfuscated code by allowing to specify obfuscator changes log file and use it to find actual classes/resources to check.

How can I prohibit possible tampering with application code?

Developers can prohibit tempering with a protected application by utilizing the Integrity verification mechanism. During the product creation, you can assign Integrity checking to any number of resources in your application. During runtime, the developer can verify if Integrity values for any of these resources were changed - which would indicate a potential tampering with the application code. At this point, the developer can apply his/her own logic, indicating how the developer wants to handle this situation.

Here is the snippet from Protection! User Guide: "...Integrity verification subsystem allows for the checking and validation that the designated key classes of the product have not changed. This type of check is done by comparing the Integrity of these classes with the original value stored somewhere in the product code, resources or files. This subsystem significantly increases time and effort needed to diagnose and locate this part of the protection system in order to attempt to break it...."