Version

8.0

PREEMPTIVE SOLUTIONS

DASHO

User Guide

PreEmptive Solutions

Manual Version 8.0-preview
www.preemptive.com

TRADEMARKS

DashO, Overload-Induction, the PreEmptive Solutions logo, and the DashO logo are trademarks of PreEmptive Solutions, LLC

Java™ is a trademark of Oracle, Inc.

.NET™ is a trademark of Microsoft, Inc.

All other trademarks are property of their respective owners.

THIS PUBLICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD CONTAIN TYPOGRAPHIC ERRORS AND/OR TECHNICAL INACCURACIES. UPDATES AND MODIFICATIONS MAY BE MADE TO THIS DOCUMENT AND/OR SUPPORTING SOFTWARE AT ANY TIME.

PreEmptive Solutions, LLC has intellectual property rights relating to technology embodied in this product. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and/or other countries.

This product is distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product may be reproduced in any form by any means without prior written authorization of PreEmptive Solutions, LLC.

Note

If your application utilizes code generation, make sure it works properly with signed jars before adding Tamper Detection. You may need to sign the jars which generate the code with the same certificate.

For Example: In a Spring-based application, you would need to sign `spring-core-4.0.1.RELEASE.jar` (or a similar jar).

Note

The tamper check for Android requires access to the application's context; it expects a `getApplicationContext()` method to exist on the class where it is being injected. If you inject the tamper check into class which extends `android.context.Context`, like an `Application` or `Service` class, you are fine. If not, you will need to add the `getApplicationContext()` method and make sure it returns a proper `Context`.

Tamper Response

Separating the detection and response makes it more difficult for attackers. Having multiple and different responses scattered throughout the application increases the difficulty. Making those responses non-deterministic can make the process maddening. DashO lets you configure your response to a tampered application as simple or as complex as you desire.

The `TamperResponse` annotation adds code that interacts with a `TamperCheck` to separate the detection and response code. You can add one or more TamperResponses to your application.

The `TamperResponse` coordinates with the `TamperCheck` via a `boolean` value. A value set using the `TamperCheck`'s action is retrieved with the `TamperResponse`'s source. If the retrieved value is `true` then the response is executed.

Like the `TamperCheck` the `TamperResponse` can send a message and/or perform a response. In addition the response action can be made conditional based on a probability factor ranging from `0.0` (never) to `1.0` (always) – the default is `1.0`.

```
private static boolean tamperFlag;

@TamperCheck(action="@tamperFlag")
public static void main(final String[] args){

}

@TamperResponse(source="@tamperFlag", sendMessage=true)
private void init() {

}

@TamperResponse(source="@tamperFlag",
response=ResponseType.Exit, probability=0.05f)
private int computeResult(){

}

@TamperResponse(source="@tamperFlag",
response=ResponseType.Error, probability=0.1f)
private FileInputStream readInput(){

}
```

When you are requesting the sending of analytics messages with `TamperResponse`s you may need to provide some additional configuration information. If your application is using analytics and contains an `ApplicationStart` you need no further configuration. If you are using `TamperResponse`s that send messages then you need to supply the company and application IDs using other annotations or provide them on the Instrumentation Properties panel.

Debug Checking and Response

DashO can instrument Android applications to detect if they are being debugged. The debugging check is implemented using Custom Annotations that can be placed in your source code.

Debugging Checking

To detect if an Android app is being debugged, place a `DebuggingCheck` on one or more methods in your application. DashO adds code that performs a runtime check that determines if it is being debugged. If the check determines it is being debugged you can configure one or more ways to respond to it. You can choose one or both of the following at the time the check is performed:

- **Call a method or set a field.**
  You can have the debugging state passed back to your application by invoking a method that takes a single `boolean` or by setting a `boolean` field. When a debugging check succeeds *(being debugged)*, the boolean value is `true`. If the check fails *(not being debugged)* then `false` is used. Your application can act on this information immediately or store it for later interaction with a DebuggingResponse annotation.
- **Perform a response.**
  There are several immediate responses that can be taken: `exit` – exit the application with a randomly non-zero return code; `hang` – cause the current thread to hang; `error` – throw a randomly selected error; `exception` – throw a randomly selected unchecked exception. If the value is not set then the default response of `none` is taken. The randomization of return codes and `Throwable`s is performed at time the check is injected not at run time. Errors and exceptions are thrown with an empty stack trace to conceal their origin.

When you select more than one of these actions they are performed in the order listed above. If you do not request any of these or none are valid the debugging check will be skipped and DashO will produce an error message.

An application can contain any number of `DebuggingChecks` with various configurations. Using more than one check or mixing the responses will hamper attackers.

Examples

```
private static boolean debuggingFlag;

@DebuggingCheck(action="@debuggingFlag")
public void onCreate(Bundle check){

}

@DebuggingCheck(response=ResponseType.Hang)
private int computeResult(){

}
```

Debugging Response

Separating the detection and response makes it more difficult for attackers. Having multiple and different responses scattered throughout the application increases the difficulty. Making those responses non-deterministic can make the process maddening. DashO lets you configure your response to a debugging application as simple or as complex as you desire.

The `DebuggingResponse` annotation adds code that interacts with a `DebuggingCheck` to separate the detection and response code. You can add one or more DebuggingResponses to your application.

The `DebuggingResponse` coordinates with the `DebuggingCheck` via a `boolean` value. A value set using the `DebuggingCheck`'s action is retrieved with the `DebuggingResponse`'s source. If the retrieved value is `true` then the response is executed.

Like `DebuggingCheck, DebuggingResponse` can perform a response. In addition the response action can be made conditional based on a probability factor ranging from `0.0` (never) to `1.0` (always) – the default is `1.0`.

Examples

```
private static boolean debuggingFlag;

@DebuggingCheck(action="@debuggingFlag")
public void onCreate(Bundle state){

}

@DebuggingResponse(source="@debuggingFlag",
response=ResponseType.Exit, probability=0.05f)
private int computeResult(){

}

@DebuggingResponse(source="@debuggingFlag",
response=ResponseType.Error, probability=0.1f)
private FileInputStream readInput(){

}
```

Debug Enabled Check

To detect if an app is setup to be debugged, place a `DebugEnabledCheck` on one or more methods in your application. DashO adds code that performs a runtime check that determines if it is setup to allow debugging. If the check determines it is so setup, you can respond to it in one or more ways. You can choose one or both of the following at the time the check is performed:

- **Call a method or set a field.**
  You can have the debug enabled state passed back to your application by invoking a method that takes a single `boolean` or by setting a `boolean` field. When a debug enabled check succeeds *(setup for debugging)*, the boolean value is `true`. If the check fails *(not setup for debugging)* then `false` is used. Your application can act on this information immediately or store it for later interaction with a DebugEnabledResponse annotation.
- **Perform a response.**
  There are several immediate responses that can be taken: `exit` – exit the application with a randomly non-zero return code; `hang` – cause the current thread to hang; `error` – throw a randomly selected error; `exception` – throw a randomly selected unchecked exception. If the value is not set then the default response of `none` is taken. The randomization of return codes and `Throwable`s is performed at time the check is injected not at run time. Errors and exceptions are thrown with an empty stack trace to conceal their origin.

When you select more than one of these actions they are performed in the order listed above. If you do not request any of these or none are valid the debug enabled check will be skipped and DashO will produce an error message.

An application can contain any number of `DebugEnabledCheck` with various configurations. Using more than one check or mixing the responses will hamper attackers.

Examples

```
private static boolean debuggingFlag;

@DebugEnabledCheck(action="@debuggingFlag")
public void onCreate(Bundle check){

}

@DebugEnabledCheck(response=ResponseType.Hang)
private int computeResult(){

}
```

Note

The debug enabled check for Android requires access to the application's context; it expects a `getApplicationContext()` method to exist on the class where it is being injected. If you inject the debug enabled check into class which extends `android.context.Context`, like an `Application` or `Service` class, you are fine. If not, you will need to add the `getApplicationContext()` method and make sure it returns a proper `Context`.

Debug Enabled Response

Separating the detection and response makes it more difficult for attackers. Having multiple and different responses scattered throughout the application increases the difficulty. Making those responses non-deterministic can make the process

maddening. DashO lets you configure your response to a debug enabled application as simple or as complex as you desire.

The `DebugEnabledResponse` annotation adds code that interacts with a `DebugEnabledCheck` to separate the detection and response code. You can add one or more DebugEnabledResponses to your application.

The `DebugEnabledResponse` coordinates with the `DebugEnabledCheck` via a `boolean` value. A value set using the `DebuggingCheck`'s action is retrieved with the `DebugEnabledResponse`'s source. If the retrieved value is `true` then the response is executed.

Like the `DebugEnabledCheck` the `DebugEnabledResponse` can perform a response. In addition the response action can be made conditional based on a probability factor ranging from `0.0` (never) to `1.0` (always) – the default is `1.0`.

Examples

```
private static boolean debuggingFlag;

@DebugEnabledCheck(action="@debuggingFlag")
public void onCreate(Bundle state){

}

@DebugEnabledResponse(source="@debuggingFlag",
response=ResponseType.Exit, probability=0.05f)
private int computeResult(){

}

@DebugEnabledResponse(source="@debuggingFlag",
response=ResponseType.Error, probability=0.1f)
private FileInputStream readInput(){

}
```

Shelf Life

Shelf Life is an application inventory management function that allows you to add expiration and notification logic to your application. This logic enforces an expiration policy by exiting the application and/or sending an analytics message. For example, a beta application can be made to expire on a particular date. You can schedule an application's expiration for a specific date or a number of days from a starting date and optionally specify a warning period prior to expiration. The expiration information may be placed within your application or can be read from an encrypted external token file. The latter allows you to extend the expiration of the application by issuing a new token file rather than rebuilding your application. Expiration checks can be added to one or more locations in your application.

Activation Key

To start using Shelf Life you must obtain a Shelf Life Activation Key from PreEmptive Solutions. This key is used to generate the tokens that contain the expiration information. PreEmptive will issue you a data file containing the key that generates the tokens and identifies your application. This key is read by DashO when your code is instrumented and can either be specified in the user interface or via a Shelf Life annotation.

Shelf Life Tokens

A Shelf Life Token is an encrypted set of data containing application and expiration information. It can be inserted into your application or stored outside of the application. You can use the DashO user interface or an Ant task to create an externally stored token.

The expiration and warning information for the token is entered via the user interface or via Shelf Life annotations. The annotations can either be added to your source or added with DashO's virtual annotations. Expiration and warning dates can be specified in two different ways:

Absolute Dates – A fixed date for the expiration date or the beginning of the warning period can be specified.

Relative Dates – The expiration period is the number of days from a start date. The warning period is the number of days prior to the expiration date.

You can combine absolute and relative dates - e.g. expire on 1/1/2021 and warn 30 days before expiration. Expiration information is required to create the token, but warning information is optional.

Expiration Check

The `ExpiryCheck` annotation is used to define the location in your application where an expiration check will take place. The `ExpiryCheck` can be added to your source or added with DashO's virtual annotations. If you added the annotation to your source you will need to compile with `dasho-annotations.jar` which is located in the `lib` folder where you installed DashO. By default, DashO removes references to

these annotations; therefore, the jar is not required at application runtime and does not need to be distributed with your application.

If the expiration information is set on the Instrumentation – Shelf Life screen, at the minimum a key file and expiration date, only a single annotation is required to add the expiration check:

Example

```
@ExpiryCheck
public static void main(final String[] args){
    if(args.length == 0){
        System.out.println("Hello no name");
    }else{
        System.out.println("Hello " + args[0]);
    }
}
```

This adds an expiration check to the application at the start of `main()`. You can also specify all the information as annotations:

Example

```
@ExpiryKeyFile("yoyodyne.slkey")
@ExpiryDate("01/01/2021")
@WarningPeriod("30")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}
```

The values for the annotations, dates and periods, are all strings. This allows you to use DashO's properties or environment values to parameterize them:

Example

```
@ExpiryKeyFile("${key_dir}/yoyodyne.slkey")
@ExpiryDate("01/01/${exp_year}")
@WarningPeriod("${warn_period}")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}
```

Relative Expiration Date

Expiration can be specified as a number of days from a dynamic start date. The start date could be something like the install date or the date on which the application is first run. The start date is provided at runtime by your application:

Example

```
@StartDateSource("getInstallDate()")
@ExpiryPeriod("90")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}

private static Date getInstallDate(){
    return new Date(Preferences.userRoot().node("MyApp").
                    getInt("installDate", 0));
}
```

You can use static or instance methods or fields as a source for the start date. See Specifying Sources and Actions for details.

Externally Stored Tokens

In the previous example DashO has embedded the Shelf Life token into your application. The token can also be stored externally as a file or resource and read in at run-time:

Example

```
@ExpiryTokenSource("getToken()")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}

private static Reader getToken(){
    return new
InputStreamReader(HelloWorld.class.getClassLoader().

getResourceAsStream("expiry.dat"));
}
```

The source for the token is a static or instance method that returns a `java.io.Reader` that provides the token data. See Specifying Sources and Actions for details.

Expiration Action

When `ExpiryCheck` is executed, the default action is to print a message to `System.out` and to exit with a non-zero return code:

Example

This application expired on January 1, 2016

If the application is in the warning period a message is printed to `System.out` and execution continues:

Example

This application will expire on December 31, 2016

For a more sophisticated application a custom application action can be specified:

Example

```
@ExpiryCheck(action="check()")
public static void main(final String[] args){
    // ...
}

private static void check(Token token) {
    if(token.isExpired()){
        JOptionPane.showMessageDialog(null,
            "The application expired on " +
token.getExpirationDate(),
            "Expired",
            JOptionPane.ERROR_MESSAGE);
       System.exit(1);
    }
    if(token.isInWarning()){
        JOptionPane.showMessageDialog(null,
            "The application will expire in " +
                token.getDaysTillExpiration() + " days",
            "Expiration Warning",
            JOptionPane.WARNING_MESSAGE);
    }
}
```

The action is passed the Shelf Life token that is then used to determine the action to be taken.

Shelf Life Analytics Messages

Shelf Life can send Analytics messages so you can track applications that have expired or are about to expire. The `ExpiryCheck` has a `sendMessage` property:

Example

```
@ExpiryCheck(sendMessage=true)
public static void main(final String[] args){
    // ...
}
```

If your application already contains an `ApplicationStart` you do not need to add any additional annotations. If you are going to use PreEmptive Analytics for tracking expiration you must add annotations that will identify your application:

Example

```
@ExpiryCheck(sendMessage=true)
@CompanyId("DF29A894-C1AB-5947-E0A2-0D9779CFFB63")
@ApplicationId("F0000FDA-9500-1B92-9564-A9DA3D8C3CF0")
public static void main(final String[] args){
    // ...
}
```

The `ExpiryCheck` will then automatically handle the `ApplicationStart` and `ApplicationStop` to send your expiration messages.

Note

You can also add any of the following annotations with the `ExpiryCheck` to identify your application: `Company`; `CompanyId`; `CompanyName`; `Application`; `ApplicationId`; `ApplicationName`; `ApplicationType`; `ApplicationVersion`; `ApplicationVersionSource`; `ApplicationInstanceIdSource`; `UseSsl`.

Exception Reporting

DashO can instrument applications to report on unhandled exceptions thrown by an application and optionally send a message to a PreEmptive Analytics server. Additionally the application can be instrumented to report on exceptions that are caught, uncaught, or thrown at the method level. The exception reporting is implemented using instrumentation Custom Annotations that can either be placed in your source code or added via Virtual Annotations.

Application and Thread-level Reporting

Unhandled exceptions can be intercepted and reported at either the application level or on a per-thread basis. The unhandled exceptions can be sent directly to a PreEmptive Analytics server for non-GUI applications without user interaction. For GUI applications you can select to have a dialog box presented to your application's user so they may choose to send the report or not. They will also be able to enter information about the activities they were performing prior to the exception as well as some contact information. This information is optional, but if entered is available on the PreEmptive Analytics Workbench or PreEmptive Analytics for TFS along with the exception information.

Application and thread-level reporting is added with the `AddUncaughtExceptionHandler` annotation. Properties of the Annotation determine if the handler is installed as the default handler or only for the current thread and whether a dialog is displayed to the user.

Examples

```
@AddUncaughtExceptionHandler(showDialog=true)
public static void main(final String[] args){
    // ...
}

new Thread() {
    @AddUncaughtExceptionHandler(thread=true)
    public void run() {
        // ...
    }
}.start();
```

Allowing the user to interact with a dialog gives them an opportunity to override the global opt-in setting. If the user chooses to send the report it will override an opt-out from other Analytics messages. Your application will still require the configuration information that will allow it to be identified to a PreEmptive Analytics Server. If the dialog is requested in a non-GUI application the report will only be sent if the user has opted-in to sending messages.

If you choose to send the report without user interaction the report will only be sent if the user has opted-in to sending analytics messages.

Note

These features are available as an API in the `ExceptionHandler` class.

Method-level Reporting

If you require a more fine grained approach to the reporting of exceptions you can use Annotations to track exceptions at the method level. DashO provides three Annotations that are used to add method level exception reporting: `ReportCaughtExceptions`; `ReportThrownExceptions`; `ReportUncaughtExceptions`. All three annotations support the following behaviors:

- **Send an analytics fault message.**
  A fault message will be sent to a PreEmptive Analytics server. This is the `sendMessage` property. The default is to send a message. To send the message your application must contains an `ApplicationStart` and the user must opt-in to the sending of messages.
- **Call a method or set a field.**
  You can have the exception passed back to your application by invoking a method that takes a `Throwable` or by setting a `Throwable` field. This is the `action` property.

If you use both behaviors the sending of the message is performed before the action. The following example show how the reporting of exceptions can be added at the class level so that it applied to all methods in the class. In the example the messages are sent to a PreEmptive Analytics server as well as logged locally using Log4J.

Examples

```
import org.apache.log4j.Logger

@ReportCaughtExceptions(action="@onCatch()")
@ReportThrownExceptions(action="@onThrow()")
class MyClass {
    private final static Logger log =
Logger.getLogger(MyClass.class)

    public void execute(){
        // ...
    }

    private static void onCatch(Throwable t){
        log.info("MyClass caught " + t.getClass().getName(),
t);
    }

    private static void onThrow(Throwable t){
        log.warn("MyClass threw " + t.getClass().getName(),
t);
    }

}
```

In addition to these previously described properties the `ReportUncaughtExceptions` annotation allows you to ignore the unhandled exception. Methods that have a numeric return will return zero when an unhandled exception is ignored. Methods that return objects or arrays will return `null`.

In the following example calling the `div(x, 0)` could cause an `ArithmeticException` to be thrown and printed to `System.err` but the method would return zero.

Example

```
@ReportUncaughtExceptions(sendMessage=false,
action="onErr()",
                          ignore=true)
int div(int num, int denom){
    return num / denom;
}

void onErr(Throwable t){
    t.printStackTrace();
}
```

Getting Version Information

If you use DashO in an automated process you can get version information by calling methods on classes in DashO. The `DashOPro.jar` contains classes that have static methods that return version information. These classes are: `DashOPro`; `DashOProGui`; `Watermarker`.

The classes contain the following static methods:

`static String getVersion()`

The version number in *N.N.N* format, e.g. `6.12.0`

`static int getVersionMajor()`

The major version number, e.g. `6`

`static int getVersionMinor()`

The minor version number, e.g. `12`

`static int getVersionRevision()`

The revision number, e.g. `0`

`static String getFullVersion()`

A human readable version of the version number. This may include text besides the version in *N.N.N* format.

`static String getFileVersion()`

The version number of the DashO project file used by this release, in *N.N.N* format. This may be different from the version returned by `getVersion()`.

Additionally, the `Lucidator.jar` contains the `Lucidator` class which contains all of the above methods except for `getFileVersion()`.

Using Custom Encryption

You can configure DashO to use your own encryption algorithms to deal with the strings found during the string encryption phase. The implementation can be as simple or complex as you desire. Please keep in mind however, that a long running decryption method will ultimately slow down your application. There are two parts to this process: Encryption and Decryption. The encryption method is used when DashO processes the project. The encryption class and method need to be packaged in a separate jar and configured to be used by the project. The decryption method is packaged with the application. The decryption class and method need to be part of the inputs of the project and be configured to be used by the project.

Encryption

The encryption algorithm must be in a public static method that takes a single string, the plaintext, as an argument and returns an array of two non-null strings, the key and the ciphertext.

Example

```
public static String[] encrypt (String plainText) {
  String key = {however you want to determine it};
  String cipherText = {however you want to create it};
  return new String[]{key,cipherText};//The order is
important!
}
```

Decryption

The decryption algorithm must be in a public static method that takes two strings, the key and ciphertext, as arguments and returns a single non-null string, the plaintext. It must be able to properly decrypt the ciphertext created by the encryption method.

Example

```
public static String decrypt (String key, String cipherText)
{
  String plainText = {however you want to determine it};
  return plainText;
}
```

Note

The decryption class can still be renamed, and obfuscated, but it will be excluded from custom string encryption. If your decryption class uses other classes in your input, you may need to manually exclude them from string encryption to avoid an infinite recursive call at runtime. Custom Encryption is not supported in Quick Jar projects.