Hewlett Packard
Enterprise

HPE Security
Fortify Runtime

Software Version: 17.3

Java Edition Designer Guide

Document Release Date: April 2017
Software Release Date: April 2017

Legal Notices

Warranty

The only warranties for Hewlett Packard Enterprise Development products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HPE shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HPE required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The software is restricted to use solely for the purpose of scanning software for security vulnerabilities that is (i) owned by you; (ii) for which you have a valid license to use; or (iii) with the explicit consent of the owner of the software to be scanned, and may not be used for any other purpose.

You shall not install or use the software on any third party or shared (hosted) server without explicit consent from the third party.

Copyright Notice

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number
- Document Release Date, which changes each time the document is updated
- Software Release Date, which indicates the release date of this version of the software

To check for recent updates or to verify that you are using the most recent edition of a document, go to:

https://www.protect724.hpe.com/community/fortify/fortify-product-documentation

You will receive updated or new editions if you subscribe to the appropriate product support service. Contact your HPE sales representative for details.

Contents

Preface

Contacting HPE Security Fortify Support

If you have questions or comments about using this product, contact HPE Security Fortify Technical Support using one of the following options.

To Manage Your Support Cases, Acquire Licenses, and Manage Your Account

https://support.fortify.com

To Email Support

fortifytechsupport@hpe.com

To Call Support

1.844.260.7219

For More Information

For more information about HPE Security software products: http://www.hpenterprisesecurity.com

About the Documentation Set

The HPE Security Fortify Software documentation set contains installation, user, and deployment guides for all HPE Security Fortify Software products and components. In addition, you will find technical notes and release notes that describe new features, known issues, and last-minute updates. You can access the latest versions of these documents from the following HPE Security user community website:

https://www.protect724.hpe.com/community/fortify/fortify-product-documentation

You will need to register for an account.

Change Log

The following table lists changes made to this document. Revisions to this document are published only if the changes made affect product functionality.

Software Release / Document Version	Changes
17.3	Updated: Minor update for 17.3 release; no significant content changes.
16.8	Updated: Minor edits. • Renamed Built-in filters - `StripSensitiveData` is now named `MaskPersonalData` and `CEFFilter` is now `ProcessCEF` . • Increased `syslog protocol restrict` message size - Supports messages up to 65,500 bytes.
16.3	Updated: Minor update for 16.3 release; no significant content changes. HP to HPE rebranding.

Chapter 1: Introduction to this Guide

This document provides content to aid in the configuration and customization of Fortify Runtime for a given application that operates on a Java platform.

Intended Audience

The audience for this guide may be a Fortify Runtime Solution Designer who often creates event handlers and chooses values for settings, sometimes writes rules, and occasionally creates a monitor. The Fortify Runtime Solution Designer must understand both software and security.

Executable Sample Code

Throughout this guide, you will find numerous instances of sample code. The Fortify Runtime SDK is available in the HPE Security Fortify Software ISO.

The following files are available electronically. See the *HPE Security Fortify Software Security Center System Requirements* for details.

- `HPE_Security_Fortify_Runtime_SDK_Dotnet_xx.x.zip`

where `x.xx` represents the current version of Fortify Runtime.

Copy or download the appropriate file and then unzip or untar the file wherever you want.

The directory structure of the unzipped or untared contents is as follows:

- `HPE_Security_Runtime_SDK_Java_x.xx.zip`
  `<unzip_dir>\java_sdk\javadoc`
  `<unzip_dir>\java_sdk\samples`
  `<unzip_dir>\java_sdk\schema`

Related Documents

This topic describes documents that provide information about HPE Security Fortify Runtime.

Note: The Protect724 site location is https://www.protect724.hpe.com/community/fortify/fortify-product-documentation.

All Products

The following documents provide general information for all products.

Document / File Name	Description	Location
HPE Security Fortify Software System Requirements HPE_Sys_Reqs_<*version*>.pdf	This document provides the details about the environments and products supported for this version of HPE Security Fortify Software.	Included with product download and on the Protect724 site
HPE Security Fortify Software Release Notes HPE_FortifySW_RN_ <*version*>.txt	This document provides an overview of the changes made to HPE Security Fortify Software for this release and important information not included elsewhere in the product documentation.	Included on the Protect724 site
What's New in HPE Security Fortify Software <version> HPE_Whats_New_ <*version*>.pdf	This document describes the new features in HPE Security Fortify Software products.	Included on the Protect724 site
HPE Security Fortify Open Source and Third-Party License Agreements HPE_OpenSrc_<*version*>.pdf	This document provides open source and third-party software license agreements for software components used in HPE Security Fortify Software.	Included with product download and on the Protect724 site
HPE Security Fortify Glossary HPE_Glossary.pdf	This document provides definitions for HPE Security Fortify Software terms.	Included with product download and on the Protect724 site

HPE Security Fortify Runtime

The following documents provide information about Fortify Runtime.

Document / File Name	Description	Location
HPE Security Fortify Runtime .NET Edition Designer Guide HPE_RT_DotNet_Design_Guide_<*version*>.pdf PDF only; no help file	This document provides information to aid in the configuration and customization of Fortify Runtime for a given application that operates on a .NET platform. The audience for this guide includes an HPE Security Fortify Runtime Solution Designer who often creates event handlers and chooses values for settings, sometimes writes rules, and occasionally creates a monitor. The HPE Security Fortify Runtime Solution Designer must understand both software and security.	Included with product download and on the Protect724 site
HPE Security Fortify Runtime Java Edition Designer Guide HPE_RT_Java_Design_Guide_<*version*>.pdf PDF only; no help file	This document provides information to aid users in the configuration and customization of Fortify Runtime for a given application that operates on a Java platform. The audience for this guide includes HPE Security Fortify Runtime Solution Designers who often create event handlers and choose values for settings, sometimes write rules, and occasionally create a monitor. The Fortify Runtime Solution Designer must understand both software	Included with product download and on the Protect724 site

Document / File Name	Description	Location
	and security.	
HPE Security Fortify Runtime Application Protection (RTAP) .NET Installation Guide HPE_RTAP_DotNet_Install_ *<version>*.pdf HPE_RTAP_DotNet_Install_ Help_*<version>*	This document describes how to install the Fortify Runtime Agent for applications running under a supported .NET Framework on a supported version of IIS.	Included with product download and on the Protect724 site
HPE Security ArcSight Application View Runtime Agent Installation Guide HPE_AppView_RT_Agent_ Install_*<version>*.pdf HPE_AppView_RT_Agent_ Install_Help_*<version>*	This document describes how to install the Fortify Runtime Agent for applications running under a supported Java Runtime Environment (JRE) on a supported application server or service and applications running under a supported .NET Framework on a supported version of IIS.	Included with product download and on the Protect724 site
HPE Security Fortify Runtime Application Protection Operator Guide HPE_RTAP_Oper_Guide_ *<version>*.pdf HPE_RTAP_Oper_Help_ *<version>*	This document provides information and procedures that enable you to run and monitor the operation of HPE Security Fortify Runtime Application Protection.	Included with product download and on the Protect724 site
HPE Security ArcSight Application View Quick Start HPE_AppView_Quick_Start_ *<version>*.pdf PDF only; no help file	This document provides brief instructions about how to get started with installing and configuring HPE Security ArcSight Application View.	Included with product download and on the Protect724 site
HPE Security Fortify RTAP Rulepack Kit Guide	This document describes the detection capabilities of HPE	Included with product download and on the Protect724 site

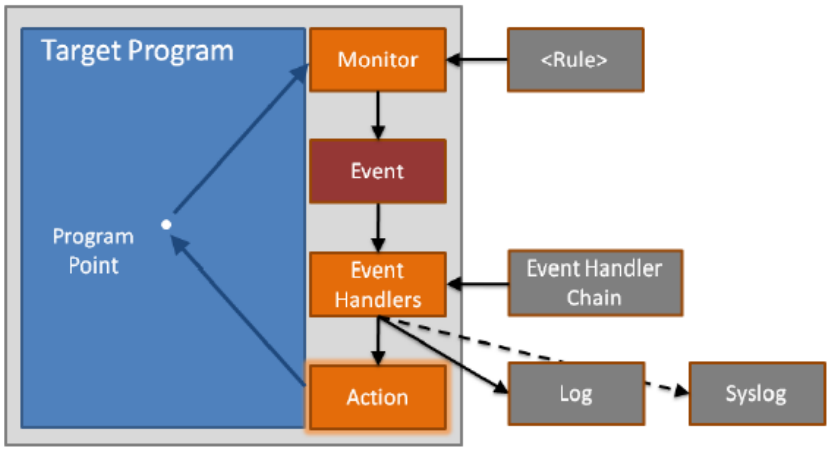
Document / File Name	Description	Location
HPE_RTAP_Rulepack_Kit_ <*version*>.pdf PDF only; no help file	Security Fortify Runtime Application Protection (RTAP) and the HPE Security Fortify RTAP Rulepacks. Each category of attack, vulnerability, or audit event detected by RTAP is described in this document.	
HPE Security Fortify RTAL Rulepack Kit Guide HPE_RTAL_Rulepack_Kit_ <*version*>.pdf PDF only; no help file	This document describes the capabilities of the HPE Security Fortify Runtime Application Logging (RTAL) Rulepack Kit. The HPE Security Fortify RTAL Rulepack is a special Runtime Kit for HPE Security Fortify Runtime. It provides information about web application internal activities to ArcSight analysis servers so that these events can be correlated with other existing ArcSight event information.	Included with product download and on the Protect724 site
HPE Security Fortify Runtime Performance Tuning Guide HPE_RT_Perf_Tuning_ <*version*>.pdf PDF only; no help file	This document recommends ways to address performance bottlenecks a user might encounter in HPE Security Fortify Runtime. It is meant to supplement, not replace, the HPE Fortify Runtime Installation and Configuration guides. It is intended for users who are familiar with and can correctly install and run HPE Security Fortify Runtime.	Included with product download and on the Protect724 site

Chapter 2: Overview of HPE Security Fortify Runtime: Java Edition

This section contains the following topics:

Overview of Fortify Runtime for Java Components

Fortify Runtime for Java functions with programs running under a supported Java Virtual Machine (JVM). The Java program can be a Web application container or any other Java program. The following figure shows the relationship of Fortify Runtime components.



Behavior Is Defined by Rules

Fortify Runtime Behavior is defined by rules. A rulepack serves as a container for one or more rules. HPE Security Fortify supplies Runtime Rulepacks for detecting common types of attacks and vulnerabilities. You can also create your own custom Rulepacks.

A single rule specifies one or more program points that declare where to monitor a target program and one or more monitors that define what to check for at a given program point. When a monitor detects the specified operations in the target program, it creates an event.

Event Handlers Define Response to Events

An event is a collection of attributes. Attributes provide information such as category of problem that has been detected and the location in the code where the problem was detected.

Fortify Runtime evaluates the ongoing stream of events with a set of event handlers. An event handler matches against event attributes and specifies the way Fortify Runtime should respond. A response might include a passive activity such as logging the event or sending out a syslog notification, or it might include an action: a change to the state of the target program. An action could throw an exception or display a special error message to the user.

Event handlers are organized in an event handler chain. By default, Fortify Runtime stops evaluating the event handler chain after it finds the first matching event handler. The event handler chain enables the security designer to organize event handlers into a sequence that provides the optimal action to take to protect the target program.

Overview of Fortify Runtime Modes of Operation

Fortify Runtime has two modes of operation: Standalone Mode and Federated Mode.

Overview of Standalone Mode

In Standalone Mode, Fortify Runtime reads its configuration (including rules, event handlers, and other settings) from disk.

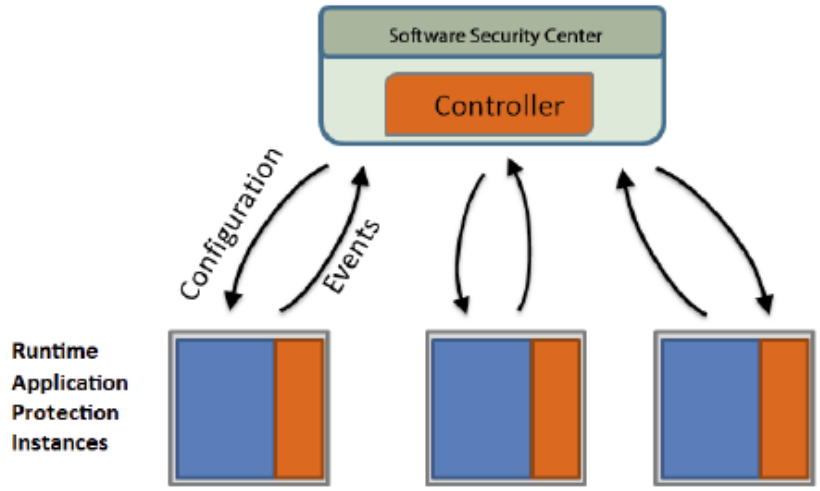
Overview Federated Mode

In Federated Mode multiple computers running HPE Security Fortify Runtime Platform work in concert. They use the network to share a common source of configuration information and common event repository.

In Federated Mode, an instance of Fortify Runtime:

- Operates as a Host—a member of an Fortify Runtime Federation
- Receives its configuration from a Federation Controller
- Transmits security events to its Federation Controller

The figure below shows the relationship of three hosts to an instance of Software Security Center running as those hosts' Federation Controller.



After a Fortify Runtime Host receives a configuration from its Federation Controller, the Host caches the configuration. The Host uses that cached configuration until the Federation Controller sends a new configuration. The Host preserves the most recent cached configuration across program restart. This enables a Fortify Runtime Host running in Federated Mode to resume operation without waiting for the Federation Controller to re-send the Host's configuration.

Fortify Runtime User Roles

Deploying, configuring, customizing, and monitoring Fortify Runtime often involves more than one person within an organization. In this section the three conceptual roles for Fortify Runtime users are described. A single individual may fulfill more than one role, or duties for a single role may be spread across a team.

Conceptual roles may be classified as:

- Fortify Runtime Analyst
- Fortify Runtime Operator
- Fortify Runtime Solution Designer

Fortify Runtime Analyst

This person is responsible for monitoring Fortify Runtime on an ongoing basis and for making limited configuration changes. A Fortify Runtime Analyst looks at the Fortify Runtime output and makes decisions. An analyst might modify event handlers or adjust settings within the structure established by the Fortify Runtime Solution Designer. This is principally a security role.

Fortify Runtime Operator

This person is responsible for installation, basic configuration, and ongoing maintenance of the Fortify Runtime system. A Fortify Runtime Operator expects the Fortify Runtime Solution Designer to provide a configuration for a particular application. With the configuration in hand, the Fortify Runtime Operator can deploy Fortify Runtime. A Fortify Runtime Operator has skills which are similar to a system administrator.

Fortify Runtime Solution Designer

This person is responsible for configuring and customizing Fortify Runtime for a given application. A Fortify Runtime Solution Designer often creates event handlers and chooses values for settings, sometimes writes Rules, and occasionally creates a monitor. The Fortify Runtime Solution Designer must understand both software and security.

Fortify Runtime User Role Examples

These roles may be fulfilled in different ways to meet the needs of different organizations.

Example 1: Business unit in a large enterprise

- Fortify Runtime Analyst: Central security team
- Fortify Runtime Solution Designer: Central security team working with software developers
- Fortify Runtime Operator: Operations team

Example 2: Small team

- Fortify Runtime Analyst: Data center Network Operations Center (NOC)
- Fortify Runtime Solution Designer: Software architect
- Fortify Runtime Operator: Development team

Example 3: Outsourced data center

- Fortify Runtime Analyst: Central security team
- Fortify Runtime Solution Designer: HPE Security Fortify Global Services
- Fortify Runtime Operator: outsourced data center operations

Chapter 3: Writing Event Handlers

This section contains the following topics:

About Event Handlers

When something noteworthy happens in the target program, the Fortify Runtime Platform generates an event. The event handlers determine how events are processed. An event handler can write an event to a log file, report the event using syslog, and/or modify the target program by invoking an action.

Event Handler Example

The following event handler chain that will throw an exception and log any session fixation attack. Events that are not related to session fixation will only be logged.

```
<EventHandlers>
    <EventHandler>
        <Match>
            <MatchAttribute name="category">Session
Fixation</MatchAttribute>
        </Match>
        <Handle>
            <Dispatch name="log"/>
            <Action name="throw"/>
        </Handle>
    </EventHandler>
    <Default>
        <Handle>
            <Dispatch name="log">
                <Setting name="picture">default (no
action)%n%all</Setting>
            </Dispatch>
        </Handle>
```

```
    </Default>
</EventHandlers>
```

The event handlers work in a chain. The handler at the top of the chain sees the event first. The event propagates down the chain until it matches an event handler.

When an event handler matches an event, it handles it and the lower portion of the chain does not see the event. (This behavior is configurable.) The following table lists the attributes for the `<EventHandler>` element.

Event Handler Attributes

Attribute Name	Description
`description`	A description of the event handler.
`enabled`	The event handler is disabled if this attribute is set to `false`. If the event handler is not enabled, an event handler will never match an event. By default `enabled` is `true`.
`label`	A short name for the event handler.
`propagate`	To have event handler match an event but also allow that event to continue down the handler chain, set `propagate` to `true`. By default, `propagate` is `false`.

Picture Strings

A number of places in the definition of an event handler accept picture strings. These strings are similar to printf-style format strings. An event's attributes will be used to perform substitutions on the given picture string. Picture strings interpret attribute names in the form %name or %{name}. %% is treated as a single % character.

In addition to regular event attributes, the following special substitutions are recognized:

- `%all` - All attributes
- `%hostname` - The name of the host computer
- `%location` - The first line of the target program stack where the event was created
- %n - Newline
- `%timestamp` - The timestamp formatted as an ISO8601 string

Matching Events

An event handler may specify any number of `Match` tags. Each Match tag can contain any combination of `MatchAttribute`, `Cluster`, and `MatchCluster` elements. All of the elements in at least one `Match` tag must match the event in order for the event handler to execute.

The `MatchAttribute` element allows an event handler to match against the attributes of an event. The `Cluster` and `MatchCluster` elements allow the event handler to match against a sequence of events over a specified period of time. We will look at `MatchAttribute`, and then discuss `Cluster`, and `MatchCluster`.

MatchAttribute

A `MatchAttribute` has a name and a value. The name of the `MatchAttribute` must match the name of the event attribute. The match is case-insensitive. The value of the `MatchAttribute` is a regular expression that must match the value of the named event attribute. The example below matches any event that has `com.example.me` as part of its stack trace (the `location` attribute of an event).

Example: MatchAttribute element

```
<MatchAttribute name="location">com\.example\.me</MatchAttribute>
```

You may invert the sense of a tag by surrounding it with a `Not` tag. For example, to match all events that are not attacks, you could write something similar to the example below.

Example: Inverting MatchAttribute

```
<Not><MatchAttribute name="attack"/></Not>
```

You may match against more than one event attribute name in a single tag by using a vertical bar (pipe) to separate the names to be matched. The example below shows a tag that matches any event with a non-empty attribute named after one of Snow White's seven dwarfs.

Example: Matching more than one attribute name with MatchAttribute

```
<MatchAttribute
name="Bashful|Doc|Dopey|Grumpy|Happy|Sleepy|Sneezy">.*</MatchAttribute>
```

The following table lists common event attributes. Note that monitors may output any additional fields as appropriate.

Common Event Handler Attributes

Event Attribute Name	Description
`category`	Event Seven Pernicious Kingdoms name
`eventID`	Unique Event ID
`eventType`	Type of event; can be any combination of attack, audit, and/or vulnerability
`kingdom`	Seven Pernicious Kingdom name
`location`	The stack trace for the program point where the event was generated

Common Event Handler Attributes, continued

Event Attribute Name	Description
metaInfo.accuracy	Accuracy of rule
metaInfo.altcategoryCWE	Common Weakness Enumeration (CWE) ID of this category
metaInfo.altcategoryOWASP2004	OWASP 2004 Top 10 mapping
metaInfo.altcategoryOWASP2007	OWASP 2007 Top 10 mapping
metaInfo.altcategoryOWASP2010	OWASP 2010 Top 10 mapping
metaInfo.audience	The audience for the metaInfo category
metaInfo.coveredSCA	Identifies issues that can be found by Fortify Static Code Analyzer (SCA)
metaInfo.impact	Impact of category if exploited
metaInfo.impactBias	Security impact of category
metaInfo.primaryAudience	Specify if the category is a security issue or quality issue
metaInfo.priority	The Fortify Priority Order of this event
metaInfo.probability	The probability that the category being exploited
MonitorID	The identifier of the monitor that generated the event
Reason	The reason why we flag the event (for example, report the event as SQLi)
request.cookies	HTTP request cookies
request.headers	The list of all HTTP headers
request.host	The HTTP request host
request.ip	Client IP
request.method	The HTTP method
request.parameters	HTTP request parameters
request.path	The HTTP request path without query string

Common Event Handler Attributes, continued

Event Attribute Name	Description
`request.port`	The local HTTP port
`request.remote_user`	The username of this HTTP session (if authenticated)
`request.requestIid`	A unique ID per each HTTP request
`request.scheme`	The HTTP scheme of this HTTP connection
`request.sessionId`	HTTP Session ID
`RuleID`	The identifier for the rule that generated the event
`suggestedAction`	The action to be taken
`thrown_from`	The stack trace for the original exception caught by the rule
`timestamp`	Epoch time of the event in ms.
`Trigger`	The trigger of this event

Some event attributes, notably action and dispatch, are added to the event after the event handler chain has been evaluated. Because the attributes do not yet exist when the event handler chain is evaluated, you cannot match against them in the event handler chain.

Cluster and MatchCluster

A cluster is a sequence of similar events that occur within a specified period of time. You might use a cluster to detect a brute-force password guessing attack or to limit the number of suspicious requests allowed before a user's session is terminated.

The `Cluster` tag defines a cluster. The `Cluster` tag shown in the example below will match when 3 events with the same category name occur within a 10 second window. In addition to matching on the 3rd event, the tag will continue to match any event with the same category name for 5 seconds after the 3rd event.

Example: Cluster tag

```
<Cluster id="nextc" n="3" window="10" linger="5" picture="%category"
maxClusters="100"/>
```

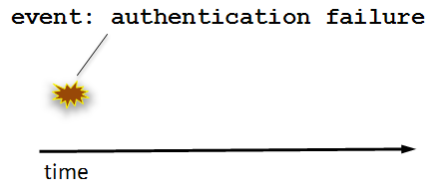
The following table explains each of the cluster attributes. A cluster requires all attributes to be defined.

Cluster Attributes

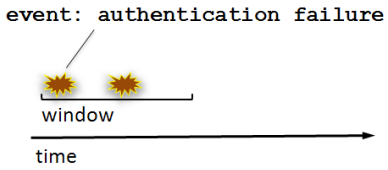
Event Attribute Name	Description
`id`	The identifier for the cluster tag (used by `ClusterMatch` tags).
`linger`	The length of time (in seconds) the tag will continue to match after a cluster is found.
`maxClusters`	The maximum number of clusters the tag will attempt to track. A high value allows the tag to keep track of more events but requires more memory. A low value could cause the tag to miss clusters when they occur.
`n`	The number of events that will cause the cluster to match.
`picture`	The event format string used to group events. Example: the picture `%request.sessionid%category` will create clusters out of the session identifier and the category from the event. You might use such a picture to filter out similar events that occur within the same session in a short period of time.
`window`	The length of time (in seconds) in which the `n` events must occur.

The following series of illustrations show how a `Cluster` tag works. After three events with the category `authentication failure` occur within the period of time defined by `window`, the `Cluster` tag matches for the period of time defined by `linger`.

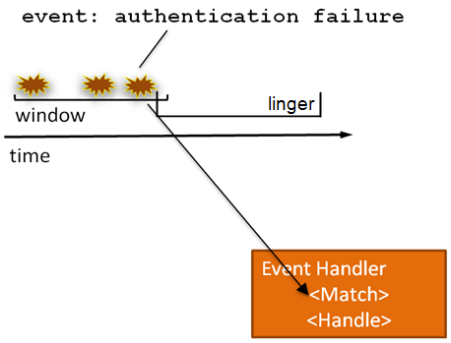
Example: Cluster tag single event



Example: Cluster tag second event



Example: Cluster tag third event



The `MatchCluster` tag refers to the state of a cluster and will match if the cluster matches, but unlike a `Cluster` tag, matching against `MatchCluster` will not change the state of the cluster. The following `MatchCluster` tag refers to `nextc`.

Example: MatchCluster tag referring to nextc

```
<MatchCluster ref="nextc"/>
```

The example in the example below uses a combination of `MatchAttribute`, `Cluster`, and `MatchCluster` tags together to block login attempts for 10 minutes after 3 failed login attempts in 5 minutes. Note that the events in this example are not generated by the default rule set.

Example: Combination of MatchAttribute, Cluster, and MatchCluster tags

```
<EventHandler propagate="true">
    <Match>  <!-- create clusters of failed logins around the failed user
id -->
        <MatchAttribute name="category">Authentication
Failure</MatchAttribute>
        <Cluster id="cluster2" n="3" window="300" linger="600"
                 picture="%userid" maxClusters="100"/>
    </Match>
    <Handle/>
</EventHandler>
<EventHandler>
    <Match> <!-- before performing login, check to see if this user has
                 already failed too often -->
        <MatchAttribute name="category">Authentication
Attempt</MatchAttribute>
        <MatchCluster ref="cluster2"/>
    </Match>
    <Handle>
        <Action name="throw"/>
    </Handle>
</EventHandler>
```

Handling Events

The Handle section of an event handler specifies the operations to carry out when an event is matched. You can add to an event's list of attributes, write the event to the event log file, or communicate the event using the syslog protocol. You can also modify the behavior of the target program.

<Filter>

Filter allows an event handler to invoke a code module which can modify the event being processed. A filter can add, subtract, or modify event attributes or terminate the event. The Filter element specifies the type of filter to use with either an attribute named name or an attribute named class. If a filter declares a name, then the preferred way to use the filter is to reference it by its name using the name attribute. A filter that does not declare a name can be referenced by its fully qualified class name using the class attribute.

The example below shows an event handler that uses two filters in sequence to modify all events by adding the HTTP request being processed when the event occurred and by filling out the trigger attribute that explains the reason for the event.

Example: Filter

```
<EventHandler propagate="true" description="Standard event filters"
              label="Event filters">
  <Match/>
  <Handle>
    <Filter name="AffixHttpRequest"/>
    <Filter name="ProcessTrigger"/>
  </Handle>
</EventHandler>
```

A filter may accept settings, as shown in the following example.

Example: A Filter with Settings

```
<EventHandler propagate="true" description="Run my filter" label="My
Filter">
  <Match/>
  <Handle>
    <Filter name="MyFilter">
      <Setting name=”remove”>.*</Setting>
    </Filter>
  </Handle>
</EventHandler>
```

Fortify Runtime supplies some built-in filters as described in the following table, and users can create their own filters and include them in a Rulepack or a monitor library. See "Writing Filter Classes" on page 80 for information about creating a custom filter.

Built-in Filters

Filter Name	Description	Settings
`AffixHttpRequest`	If an HTTP request is being processed by the thread that created the event, this filter adds attributes that describe the request. Attributes will be created for HTTP parameters and headers, the request path, and other information from the request.	None
`AffixSource`	Adds input source information to events when it is available. Must be used in conjunction with a ruleset that supports gathering of input source information, such as the WebInspect Agent ruleset.	None
`CalculatePriority`	This filter will calculate Fortify Priority Order (FPO) and will create an attribute named priority base on the attribute values of impact, probability and accuracy.	None
`MaskPersonalData`	This filter can be used to remove sensitive data that should not be recorded from event attributes.	Each setting is interpreted as a regular expression. The value of the setting is used as the replacement value for any attribute substring matched by the regular expression.
`MaskSessionId`	This filter will mask out session ID from event attributes. By default, only the first 16 characters of the session ID will be displayed.	Mask: max number of unmasked characters to be displayed, default value is 16

Built-in Filters, continued

Filter Name	Description	Settings
`ProcessCEF`	This filter will create an attribute named `CEF.syslog` by packing attributes that start with "`CEF.*`" according to HPE Security ArcSight Common Event Format (CEF).	None
`ProcessTriggerEx`	If the monitor that created the event is configured with a `TriggerPicture` property, this filter will create an attribute named `Trigger` based on `TriggerPicture`, making substitutions as appropriate. This filter should appear after other filters so that the `Trigger` attribute will include the final values of other event attributes. For example, `ProcessTrigger` should appear after `AffixHttpRequest` so that the `Trigger` attribute can include values from the HTTP request.	None

<Create Event>

`CreateEvent` allows an event handler to create a new event based on a set of attributes defined in a Rulepack. The new event will work its way through the event handler chain the same way all other events are processed. Any actions carried out as a result of processing the new event will be carried out in the context of the call to `CreateEvent`. The `CreateEvent` element requires an attribute named `attributeSet` that specifies the ID of an `<AttributeSet>` defined in a Rulepack. `CreateElement` accepts two optional attributes:

- `inherit` - specifies whether the created event should inherit attributes from the original event (defaults to true)
- `exclude` - specifies a list of attributes that should not be inherited. It defaults to "`RuleID`, `MonitorID`"

In order to prevent a poorly defined event handler chain from causing an infinite cascade of `CreateEvent` invocations, each call to `CreateEvent` increments an event attribute named `generation`. If `generation` is greater than or equal to three, no further invocations of `CreateEvent` are allowed.

The following example creates an event based on an attribute set named `stored_xss`.

```
<CreateEvent attributeSet="stored_xss"/>
```

<SetAttribute>

The `SetAttribute` element requires a name attribute that specifies the name of the attribute to be added to the event. The body of the element is the value of the new attribute. The value can reference event attributes by preceding them with a percent sign (%). You can overwrite existing event attributes. You can specify any number of `SetAttribute` tags.

`SetAttribute` supports an optional attribute named `call_method`. The value of `call_method` must refer to a method in the application or in the Java Fortify Runtime library with one of the following signatures:

- `public static String name()`
- `public static String name(String value)`
- `public static String name(HttpServletRequest req)`
- `public static String name(HttpServletRequest req, String value)`

The specified method will be invoked (with the current request and the value of the `SetAttribute` tag if the method accepts one or both of these arguments), and the value of the event attribute will be set to the value returned by the method.

The following example sets the category attribute to `my vuln` and adds an attribute named `vulnerability`.

Example: SetAttribute

```
<SetAttribute name="category">my vuln</SetAttribute>
<SetAttribute name="vulnerability"/>
```

<Dispatch>

The `Dispatch` element accepts a `name` attribute that specifies where the event is to be sent. You may specify any number of `Dispatch` elements. Valid names for `Dispatch` elements are `log` and `syslog`.

Dispatching to `log` sends the event to the event log file. Dispatching to `syslog` is much like dispatching to `log`, but the resulting message is sent out via the protocol rather than being written to the event log. `syslog` allows a setting named `severity` specifying the severity of the report. If used, the `severity` setting must have one of the following values: `debug`, `info`, `notice`, `warning`, `error`, `critical`, `alert`, or `emergency`.

The example in the example below shows how the `syslog` element sends out emergency notifications.

Example: Syslog

```
<Dispatch name="syslog"><Setting
name="severity">emergency</Setting></Dispatch>
```

Note: The syslog protocol restricts messages to 65,500 bytes. If a picture results in a message longer than 65,500 bytes, the message will be truncated.

A `Dispatch` element can include a `picture` setting to control the way the event is recorded. The picture can reference event attributes by preceding them with a percent sign (%) as shown in the example below.

Example: Dispatch

```
<Dispatch name="syslog"><Setting name="picture">Detected a %category
attack</Setting></Dispatch>
```

The above example format picture could produce the syslog message shown in the example below.

Example: Syslog message

```
Detected a my vuln attack
```

Note: The picture attribute can also be used with `log` dispatch, but using a picture other than the default will make it impossible to later convert the log file into an FPR file.

<Action>

An `Action` changes the target program. An event handler can specify any number of actions. An action can be parameterized using settings. For example, the action shown in the example below throws an exception of type `java.lang.RuntimeException` with the message `Fortify Runtime rule (rule ID)`.

```
<Action name="throw">
  <Setting name="type">java.lang.RuntimeException</Setting>
  <Setting name="message">Fortify Runtime rule %ruleid</Setting>
</Action>
```

The following table describes the supported actions and their settings.

Supported Actions

Action Name	Description
`delay`	Delay the execution of the current thread. Accepts optional `length` setting which specifies the length of time to sleep in milliseconds (default: 3000ms)
`die`	Terminate the JVM. Accepts optional settings `return` (to specify the exit code for the process) and `message` (as a final message to write to the system log.) No additional actions can be executed after this one.
`display`	Ends the current operation by throwing an exception and notifies the user. Accepts optional settings `status_code`, `forward`, `html`, and `text`. These settings are explained further below. No additional actions can be executed after this one.

Supported Actions, continued

Action Name	Description
`ignore`	Do nothing. Equivalent to no action at all.
`invalidate_session`	If the current thread is servicing an HTTP request, this action terminates the session of the user who made the request.
`rewrite`	Changes the value of a variable in the program. The variable and the changes must be specified in the rule that triggers the action. Not all rules support `rewrite`. The documentation supplied with HPE Security Fortify Rulepacks indicates the categories that are compatible with `rewrite`.
`throw`	Throws an exception. Accepts optional settings `type` to specify the name of the exception type to be thrown and message to specify the message the exception should carry. No additional actions can be executed after this one. This action will always throw an exception, even if an exception of the type specified cannot be created.

display Action

The `display` action has four related global settings, all of which can be overridden by any of four optional settings on a specific `display` tag. The global settings for the `display` action are: `DisplayDefaultForwardPath`, `DisplayDefaultHtml`, `DisplayDefaultHttpStatusCode`, and `DisplayDefaultText`. The settings for a specific `display` tag are `status_code`, `forward`, `html`, and `text`.

If the active thread in the target program is not servicing a Web request at the time the `display` action is taken, the `display` action will throw a `RuntimeException` with a message taken from the local `text` setting (or the `DisplayDefaultText` global setting if there is no local `text` setting.)

In the context of a Web request, the precedence (from highest to lowest) and meaning of the `display` options are described in following table.

Display Setting Options

Option	Description
`DisplayDefaultForwardPath` global setting	If this setting has a value, Fortify Runtime will transfer processing of this HTTP request to the given path. Note that this is NOT an HTTP redirect. All processing stays on the server.
`DisplayDefaultHtml` global setting	If this option is set, Fortify Runtime will create an HTTP 200 response and include this value as the body of the response.

Display Setting Options, continued

Option	Description
`DisplayDefaultHttpStatusCode` global setting	If this setting has a value greater than zero, the HTTP status code of the response will be set to this value. If `DisplayDefaultText` is not empty, it will be used as the text of the response.
`DisplayDefaultText` global setting	If this option is set, Fortify Runtime will create an HTTP 200 response and include this value as the body of the response.
`forward` local setting	If this setting has a value, Fortify Runtime will transfer processing of this HTTP request to the given path. Note that this is *not* an HTTP redirect. All processing stays on the server.
`html` local setting	If this option is set, Fortify Runtime will create an HTTP 200 response and include this value as the body of the response.
`status_code` local setting	If this setting has a value greater than zero, the HTTP status code of the response will be set to this value. If the text option is set, it will be used as the text of the response.
`text` local setting	If this option is set, Fortify Runtime will create an HTTP 200 response and include this value as the body of the response.

Event Handler Scenarios

This section provides scenarios that include event handler capability. The format of the scenarios states a common business problem followed by the recommended solution. The scenarios in this section are:

- Add a Custom Message for a 403 Error
- Write Events to a Local Event Log File in Federated Mode
- Preventing Sensitive Information from Being Recorded in Events
- Running behind a Load Balancer

Add a Custom Message for a 403 Error

Problem

The Riches Bank application contains a number of intentionally placed vulnerabilities that allow a demonstration of Fortify Runtime's capabilities. But if someone disables Fortify Runtime protections,

they would leave their computer open to attack. The problem was addressed by limiting network access to the Riches Bank application to the local computer so that it will be okay for the person sitting at the console attack themselves, but no one on another computer could take advantage of the weaknesses in the Riches Bank application.

Tomcat has a facility for solving the access control portion of this problem. Adding the following line to server.xml causes Tomcat to return a status code 403 for any non-local request:

```
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
allow="127.0.0.1,0:0:0:0:0:0:0:1%0"/>
```

However, this creates a new problem: now people will be blocked, but they will not know why. Tomcat does not have any easy way to return an error page rather than blocking the request.

Solution

The default ruleset already generates an event when a 403 occurs, so all that needs to be done to create a special error page is add an event handler that carries out a display action when it sees the 403 coming from a machine other than localhost. The example below demonstrates the event handler used.

Example: Event handler error page

```
<EventHandler description="Informational message for people who get a
403."
                    label="handle 403 for non-localhost">
  <Match>
    <MatchAttribute name="Trigger">403</MatchAttribute>
    <Not>
      <MatchAttribute name="request.ip">127.0.0.1</MatchAttribute>
    </Not>
  </Match>
  <Handle>
    <Action name="display">
      <Setting name="html"><![CDATA[
        <html><p>This tomcat is set to accept connections only from
127.0.0.1.
        It appears you are trying to access it from a different IP
address.
        If necessary, you may disable this security feature by removing
the
        RemoteAddrValve in server.xml</html>
      ]]></Setting>
    </Action>
  </Handle>
</EventHandler>
```

Write Events to a Local Event Log File in Federated Mode

Problem

For creating new rules or debugging in Federated Mode, it can be handy to look at the raw event information being sent to the controller. This scenario addresses how to write events to the local `event.log`.

Solution

There is a system-defined setting for enabling local event logging under System Defined Settings on the Logging tab.

Alternatively, you can write events to the local event log file by adding the line `<Dispatch name="log">` to the same event handler that sends events to the controller. When this is finished, it looks like the results in the following example.

Example: Write events to local log

```
<!-- 3) send to controller -->

<EventHandler propagate="true" description="Send events to the controller"
              label="Event logger">
  <Match/>
  <Handle>
    <Dispatch name="controller"/>
    <Dispatch name="log">
  </Handle>
</EventHandler>
```

Preventing Sensitive Information from Being Recorded in Events

This scenario illustrates one way to prevent sensitive data from ending up in the event log or the controller. See the "Writing Filter Classes" on page 80 chapter for another approach.

Problem

If you log on to the Riches Bank application and drill down on an account, you will receive numerous Privacy Violation events. If you examine the event details, on the **Request** tab you will see the account number from the URL, similar to the following.

```
acctno: 5424000011112221
```

There is a way to prevent the Fortify Runtime operator from seeing this information.

Solution

Add an event handler that removes the sensitive information from the event using a `<SetAttribute>` tag. Imagine that for the Riches Bank application there are numerous web pages that include `acctno` as a request parameter. In order to remove the sensitive information from all events for all pages, add the following event handler to the beginning of the event handler chain.

Example: Event handler to remove sensitive information

```
<!-- 0) remove sensitive info from events -->
<EventHandler description=”Hide the accountno parameter in all events."
label="Hide acctno" propagate="true" >
   <Match>
      <MatchAttribute name="request.parameters.acctno"/>
   </Match>
   <Handle>
      <SetAttribute name="request.parameters.acctno">(hidden by Fortify
Runtime configuration)
      </SetAttribute>
   </Handle>
</EventHandler>
```

If you examine the account again, notice the details for a new privacy violation event. On the Request tab, observe acctno: (hidden by Fortify Runtime configuration).

Running Behind a Load Balancer

Problem

This scenario addresses a configuration whereby you may have multiple application servers behind a load balancer. When examining the details for an event, the IP address for the request is the IP address of the load balancer and not the IP address for the real user. There is a way to make the IP address show up as the IP address of the real user.

Solution

Add an event handler that sets the correct IP address using a `<SetAttribute>` tag. Most load balancers set an HTTP header named `x-forwarded-for` that gives the IP address of the original request. You can set the IP address of the request to be the value of this header by adding the following event handler to the event handler chain just after the event handler which includes the `AffixHttpRequest` filter.

Example: Event handler that sets IP address

```
<!-- 0) set the IP address of the request (compensate for the load
balancer) -->
<EventHandler description="Set the IP address of the request if it has
come
 through a load balancer." label="Fix IP Address" propagate="true" >
  <Match>
    <MatchAttribute name="request.headers.x-forwarded-for"/>
  </Match>
  <Handle>
    <SetAttribute name="request.ip">%{request.headers.x-forwarded-
for}</SetAttribute>
  </Handle>
</EventHandler>
```

Chapter 4: Writing Rules

This section contains the following topics:

Introduction to Fortify Runtime Rules

A Fortify Runtime rule:

- Specifies how Fortify Runtime interacts with a target program
- Specifies one or more program points
- Specifies one or more monitors that connect to the rule's program points
- Provides a configuration for each specified monitor

Multiple Fortify Runtime Platform rules are packaged in a Rulepack.

A sample Rulepack is included in the Fortify Runtime SDK under the directory /sdk/samples. The subdirectories contain the individual Rulepacks which are named similar to my_rules.xml.

Overview of a Complete Fortify Runtime Rule

The following example illustrates a Rulepack that contains a single rule.

In the example rule, the built-in Fortify Runtime monitor `ParameterMonitor` examines HTTP parameters for input that may indicate a command injection attack; for example, the presence of the character sequence `cmd.exe`.

If the `ParameterMonitor` monitor detects a suspicious parameter, then the monitor creates an event of category `Probing`: `Command Injection`; the event also contains value and action information.

Example: Example of a Fortify Runtime rule

```
<?xml version="1.0" encoding="UTF-8"?>
<RulePack xmlns="xmlns://www.fortifysoftware.com/schema/runtime/rules"
```

```
engineVersion="1.0">
  <RulePackID>f101</RulePackID>
  <SKU></SKU>
  <Name>sample rulepack for Java</Name>
  <Version>2010.1.0.0004</Version>
  <Language>java</Language>
  <Description><![CDATA[A sample Java rulepack with a single rule
                       in it.]]></Description>
  <Rules>
  <RuleDefinitions>
    <Rule>
    <RuleID>d129e816-948d-SAMPLE-bed1-1bb993cf908a</RuleID>
    <ProgramPoints>
      <ProgramPoint>
        <Named>Parameter</Named>
        <Capture/>
      </ProgramPoint>
    </ProgramPoints>
    <Monitors>
      <Monitor
class="com.fortify.runtime.containersupport.ParameterMonitor"
               monitorID="E4531ED3">
        <Attributes>
          <Attribute name="category">Probing: Command
Injection</Attribute>
          <Attribute name="suggestedAction">ignore</Attribute>
        </Attributes>
        <Predicate>
          values contains value: { value matches /(?i)(\/bin\/
(ba)?sh)|cmd\.exe/ }
        </Predicate>
        <Configuration>
          <Property name="TriggerPicture">
            <Value>name = %{name}, values = %{values}</Value>
          </Property>
        </Configuration>
         <Bindings>
          <Binding name="name" capture-ref="name"/>
          <Binding name="values" capture-ref="values"/>
         </Bindings>
        </Monitor>
      </Monitors>
```

```
    </Rule>
  </RuleDefinitions>
  </Rules>
</RulePack>
```

Overview of Fortify Runtime Rules

A Fortify Runtime Rulepack is an XML document that conforms to the Fortify Runtime schema specified in `/sdk/schema/rules.xsd`.

In its simplest form, a Rulepack contains a preamble with information about the Rulepack followed by a list of rules.

More typically, a Fortify Runtime Rulepack contains multiple rules, many that share similar characteristics. In order to minimize the amount of duplication between rules, a rule writer can specify program points, monitors, and sets of attributes outside of a particular rule, then reference those definitions from any rule that requires those definitions.

The example below uses pseudo-XML to illustrate the general from of a Fortify Runtime rule.

Example: General form of a Fortify Runtime Rulepack

```
<RulePack>
  …Rulepack header goes here…
  <RuleDefinitions/>
  <MonitorDefinitions/>
  <ProgramPointDefinitions/>
  <AttributeDefinitions/>
</RulePack>
```

Overview of the Fortify Runtime Rule Header

The following table lists Fortify Runtime Rulepack header elements.

Rulepack Header Elements

Name	Required	Description
`Description`	Yes	A longer explanation of the Rulepack contents for display purposes. This element does not control any Fortify Runtime behavior.
`Language`	No	Indicates the language supported by this Rulepack. Fortify Runtime will not load a Rulepack for the wrong language. In other words, it is fine to include Rulepacks for multiple languages in a single configuration. Fortify Runtime will ignore Rulepacks for other

Rulepack Header Elements, continued

Name	Required	Description
		languages.
`Name`	Yes	The display name of the Rulepack. This element does not control any Fortify Runtime behavior.
`RulePackID`	Yes	A unique string that identifies the Rulepack. This element does not control any Fortify Runtime Platform behavior.
`SKU`	Yes	A product identifier. This element does not control any Fortify Runtime behavior.
`Version`	No	Use to differentiate revisions of a Rulepack. This element does not control any Fortify Runtime behavior.

The `Rulepack` element accepts one optional attribute: `engineVersion`, which gives the minimum Fortify Runtime engine version required to run this Rulepack. If Fortify Runtime does not meet the minimum version required by the Rulepack, it will raise an error.

This Rulepack header in the example below specifies an `engineVersion` of 1.0, so it will work with all engine versions. This is equivalent to not specifying the `engineVersion` attribute at all. It specifies all of the required elements and the optional Version and Language elements too.

Example: Rulepack header

```
<RulePack xmlns="xmlns://www.fortifysoftware.com/schema/runtime/rules"
          engineVersion="1.0">
  <RulePackID>f101</RulePackID>
  <SKU></SKU>
  <Name>sample rulepack for Java</Name>
  <Version>2010.1.0.0004</Version>
  <Language>java</Language>
  <Description>
    <![CDATA[A sample Java rulepack with a single rule in it.]]>
  </Description>
```

Program Points

The `ProgramPoints` section of a rule is a sequence containing one or more `ProgramPoint` elements. Each program point defines:

- A set of attributes (optional)
- A method identifier or the name of a built-in program point(required)
- A set of target program values to capture (required)

Specifying Program Point Attributes

A program point can specify a set of program point attributes to be passed on to the monitor. This facility is useful primarily when declaring a program point outside of a rule in the `ProgramPointDefinitions` section of the Rulepack. When used in this way, all rules that use the program point inherit the program point attributes. The program point shown in the example below uses a program point attribute to define the logging level associated with the method `java.util.logging.Logger.info()`. A rule that uses this program point will automatically pass on the `LogLevel` Program Point attribute to its monitors.

Example: Program Point using an attribute

```
<ProgramPoint>
  <Attributes>
    <Attribute name="LogLevel">INFO</Attribute>
  </Attributes>
  <Method>
    <Class subclasses="true">java\.util\.logging\.Logger</Class>
    <Name>info</Name>
  </Method>
 <Capture>
    <This id="logger"/>
    <Argument id="input_text" index="0"/>
  </Capture>
</ProgramPoint>
```

In addition to specifying individual program point attributes, a program point can also make program point attributes out of all of the attributes defined in an attribute set by using a `SetReference` tag as shown in the example below.

Example: Attribute set using a SetReference tag

```
<Attributes>
  <SetReference id="LogAttrs"/>
</Attributes>
```

Specifying the Program Location in a Program Point

There are three ways to specify points in the target program in a ProgramPoint element. Every program point must supply one and only one of the following:

- An identifier for a method
- An identifier for an object initializer
- The name of a built-in program point

Specifying a Method Identifier

A method identifier supplies the class name (including the package name), the name of the method, and (optionally) the return type and parameter types for the method. Element values are interpreted as regular expressions, so it is easy to refer to groups of methods with similar characteristics, but periods and other special characters must be escaped with a backslash to avoid giving them special meaning.

The following table lists the method identifier constraints. Constraints marked as "Negatable" in the table can have their meaning inverted by setting the `negate` attribute to `true`, as demonstrated in the example below the table.

Method Identifier Values

Constraint	Number	Negatable	Description
`Class`	1 or more	Yes	Matches the name of the class in which the method is defined. Setting the attribute subclasses on the `Class` element to `true` causes the method identifier to match against any subclass of the named `Class` or implementation of the named interface.
`ClassMarker`	0 or more	Yes	Matches annotations placed on the class in which the method is defined.
`Flag`	0 or more	Yes	Matches against method flags, such as `public`, `static` or `final`.
`Marker`	0 or more	Yes	Matches annotations placed on the method.

Method Identifier Values, continued

Constraint	Number	Negatable	Description
`Name`	1 or more	Yes	Matches the name of the method.
`Parameters`	0 or 1	No	Matches against the signature (declared parameters) of the method.
`ReturnType`	0 or 1	No	Matches the method return type.

The Method example in the example below matches any implementation of the `warn` method belonging to the `org.apache.log4j.Category` class or any of its subclasses.

Example: Method identifier matching implementation of the warn method

```
<Method>
  <Class subclasses="true">org\.apache\.log4j\.Category</Class>
  <Name>warn</Name>
</Method>
```

The method identifier in the example below matches all methods named `doFilter` that belong to a class implementing the `javax.servlet.FilterChain` interface. In order to match, a method must accept two parameters, the first of type `javax.servlet.ServletRequest` and the second of type `javax.servlet.ServletResponse`.

Example: Method identifier with two parameters

```
<Method>
  <Class subclasses="true">javax\.servlet\.FilterChain</Class>
  <Name>doFilter</Name>
  <Parameters>
    <ParameterType type="javax\.servlet\.ServletRequest"/>
    <ParameterType type="javax\.servlet\.ServletResponse"/>
  </Parameters>
</Method>
```

As with other constraints, `ParameterType` is treated as a regular expression, so be sure to use the correct escaping, especially when matching on array types (`java.lang.String[]` becomes `java.lang.String\[\]`).

The next method identifier shown in the example below matches methods named `authenticate` that extend `org.apache.catalina.authenticator.AuthenticatorBase`. In order to match, a method must return a `Boolean` (as specified by the `ReturnType` tag) and the first two arguments must be of the type `org.apache.catalina.connector.Request` and `org.apache.catalina.connector.Response`. Because the method identifier includes the tag

`Ellipsis` after the two parameter types, it will match against methods that accept only two arguments or methods that take any number of additional arguments with arbitrary types.

Example: Method identifier with matching methods

```
<Method>
  <Class
subclasses="true">org\.apache\.catalina\.authenticator\.AuthenticatorBase<
/Class>
  <Name>authenticate</Name>
  <ReturnType type="boolean"/>
  <Parameters>
    <ParameterType type="org\.apache\.catalina\.connector\.Request"/>
    <ParameterType type="org\.apache\.catalina\.connector\.Response"/>
    <Ellipsis/>
  </Parameters>
</Method>
```

The final `Method` example shown in the example below matches any public, non-static method in any subclass of `javax.servlet.http.HttpServlet` that has an `org.junit.Test` annotation.

Example: Method identifier matching any public, non-static method

```
<Method>
  <Class subclasses="true">javax\.servlet\.http\.HttpServlet</Class>
  <Name>.*</Name>
  <Marker><Name>org\.junit\.Test</Name></Marker>
  <Flag>public</Flag>
  <Flag negate="true">static</Flag>
</Method>
```

Specifying a Built-in Program Point

Fortify Runtime supplies named built-in program points for placing monitors in popular locations or for special purposes. Named program points restrict the behavior of a monitor; the only monitor control point that will be invoked is complete, and the monitor is not allowed to modify program values. A monitor is allowed to throw an exception from a named program point. The names of captured values are an implicit part of a named program point. The following table indicates the named program point that is supported.

Example: Supported named program point

Name	Description	Captured Value
`Parameter`	Fires the first time a new parameter is read for a given request.	`name`: the name of the parameter, and values: the values of the parameter

The program point in the example below uses the Parameter named `ProgramPoint`.

Example: Named program point

```
<ProgramPoints>
  <ProgramPoint>
    <Name>Parameter</Name>
    <Capture/>
  </ProgramPoint>
</ProgramPoints>
```

Capturing Values in a Program Point

Fortify Runtime can capture (and in some cases modify) values from the target program. A program point can specify three kinds of values to be captured:

- A method's return value
- The object invoking the method (the value of this)
- Method arguments individually or as a range of arguments

Regardless of the capture type, every capture must specify an attribute named `id`.

The `id` attribute names the captured value so that it can be wired up to a monitor later in the rule. If multiple capture elements share the same `id`, the values will be combined into an array before being passed to the monitor.

The program point in the example below uses three capture types to examine calls to `javax.servlet.ServletRequest.getParameter()`. It captures the return value of the method and names it `value`. It names the object `request`, and it captures the first argument to the method (`index 0`) and names it name.

Example: Program point capturing a value

```
<ProgramPoint>
  <Method>
    <Class subclasses="true">javax\.servlet\.ServletRequest</Class>
    <Name>getParameter</Name>
  </Method>
  <Capture>
    <Return id="value"/>
    <This id="request"/>
    <Argument id="name" index="0"/>
  </Capture>
</ProgramPoint>
```

In some scenarios it is useful to capture a variable number of values from the target program. This may make it easier to use general purpose monitors in a variety of situations. The example below shows a program point that captures values from a hypothetical set of overloaded concatenate methods on the

`java.lang.String` class. Note that these methods do not exist in the standard implementation of `java.lang.String`.

```
public String concatenate(String a);

public String concatenate(String a, String b);

public String concatenate(String a, String b, String c);
```

In this example, the number of values captured to the capture id “parts” depends on the method in question. For each method, this object is captured along with each method argument. The values will be provided to the monitor as an array (`Object[]`), allowing a single monitor implementation to examine all arguments to the methods of the different variety.

Example: Program point capturing several values to the same id

```
<ProgramPoint>
  <Method>
    <Class subclasses="true">java\.lang\.String</Class>
    <Name>concatenate</Name>
  </Method>
  <Capture>
    <This id="parts"/>
    <Range id="parts" first="0"/>
  </Capture>
</ProgramPoint>
```

The `Range` element functions by capturing multiple arguments starting at the index specified by the attribute `first`. An optional attribute `last` may be specified to indicate where the argument range should end. By default, the range extends to the end of the argument list. The `last` attribute may be specified using a positive number (offset from the start of the argument list beginning at 0) or a negative number (offset from the end of the argument list where -1 represents the last argument). Both `first` and `last` are inclusive.

The values a program point captures affect when the `complete` control point can be executed. If a program point captures a method's return value, then `complete` cannot execute until the method is ready to return. If a program point does not capture a method's return value, then `complete` can execute before the method runs.

All of the program points used by a rule must name the set of capture ids required by the bindings in the monitors for that rule. It is not an error for a monitor to bind to a capture id which is not provided by the program point, but if it does the value of that binding will always be null. In general, any set of program points that are intended to be used together should expose the same set of capture ids.

Defining a Program Point Outside a Rule

Program points can be defined as part of a rule, as the example below shows. Program points can also be defined in the `ProgramPointDefinitions` section of a Rulepack. The format for program points is the same in both cases, but when program points are defined in `ProgramPointDefinitions`, they

must appear under a `ProgramPointSet` element that carries an id attribute so that rules can refer to the set.

The example in the following example illustrates a rule that references a Program Point set named `log4jPPSet0` followed by the definition of that program point set. The set contains two program points.

Example: Rule that references a Program Point

```
<RuleDefinitions>
  <Rule>
    <RuleID>60485088-3895-46c1-a766-4a032492c5b2</RuleID>
    <ProgramPoints>
      <SetReference id="log4jPPSet0"/>
    </ProgramPoints>
    <Monitors>
      …monitor definitions go here…
    </Monitors>
  </Rule>
</RuleDefinitions>
<ProgramPointDefinitions>
  <ProgramPointSet id="log4jPPSet0">
    <ProgramPoint>
      <Attributes>
        <Attribute name="LogLevel">FATAL</Attribute>
      </Attributes>
      <Method>
        <Class subclasses="true">org\.apache\.log4j\.Category</Class>
        <Name>fatal</Name>
      </Method>
      <Capture>
        <This id="logger"/>
        <Argument id="input_text" index="0"/>
      </Capture>
    </ProgramPoint>
    <ProgramPoint>
      <Attributes>
        <Attribute name="LogLevel">ERROR</Attribute>
      </Attributes>
      <Method>
        <Class subclasses="true">org\.apache\.log4j\.Category</Class>
        <Name>error</Name>
      </Method>
      <Capture>
```

```
        <This id="logger"/>
        <Argument id="input_text" index="0"/>
      </Capture>
    </ProgramPoint>
  </ProgramPointSet>
</ProgramPointDefinitions>
```

Overlapping Program Points

A single rule may contain multiple program points which match the same method in the target program. In this case, the behavior at runtime depends on the `Capture` declaration in each program point. If the overlapping program points declare the same `Capture`, each monitor from the rule will be executed only once. If not, each monitor will be executed once for each unique `Capture` declaration from the set of matching program points.

Monitor Specifications

A rule is responsible for declaring one or more Monitor specifications. A Monitor specification names a Monitor class and connects it to the target program through the rule's program points.

Timer Monitor Example

The following Monitor specification in the example below uses the Monitor class `com.fortify.runtime.monitor.Timer` to watch for database methods that take a long time to return.

Example: Monitor with Timer

```
<Monitors>
  <MonitorSpec class="com.fortify.runtime.monitor.Timer"
               reentrant="false" monitorID="8069F9C2-A583-4F4B-89CD-
BF9E7C6AF227">
    <Attributes>
      <Attribute
        name="category">Slow Method Call: Slow Database
Connection</Attribute>
      <Attribute name="suggestedAction">ignore</Attribute>
      <SetReference id="slowMethodCallAttrs"/>
    </Attributes>
    <Configuration>
      <Property name="Threshold">
```

```
        <Value>10000</Value>
      </Property>
    </Configuration>
    <Bindings/>
  </MonitorSpec>
</Monitors>
```

The MonitorSpec Element

The `MonitorSpec` element accepts 4 attributes: `class`, `monitorID`, `reentrant` and `reentrySetID`, as described in the following table.

MonitorSpec element attributes

Name	Required	Description
`class`	Yes	The fully qualified name of the monitor class.
`monitorID`	Yes	A globally unique ID for identifying the monitor in events and log messages.
`reentrant`	No	A Boolean that determines if a monitor will be invoked if a method in its `reentrySet` is already on the call stack. (See `reentrySetID` for more details.) Defaults to `true`.
`reentrySetID`	No	If the `reentrant` attribute is set to `false`, this value names the set of monitors that will be mutually non-reentrant with this monitor. In other words, only one monitor instance from this set can be on the call stack at any given time. Further monitors from the same set will be skipped until the first monitor is popped from the stack. If `reentrant` is `true`, this attribute is ignored. Defaults to the value of the `monitorID` attribute. More than one value may be specified, separated by commas.

The following monitor specification in the example below uses the monitor class `com.fortify.runtime.monitor.Timer`. The rule this monitor comes from is written to handle database driver implementations where a method such as `Statement.executeQuery()` delegates to an internally managed object that implements the same interface. In other words, one `Statement.executeQuery()` might well call another `Statement.executeQuery()`. In order to avoid creating duplicate events in this situation, the monitor sets `reentrant="false"`. The specification does not set the `reentrySetID` attribute, so `reentrySetID` defaults to `8069F9C2-A583-4F4B-89CD-BF9E7C6AF227` (the same value as the `monitorID` attribute).

Example:MonitorSpec

```
<MonitorSpec class="com.fortify.runtime.monitor.Timer"
  reentrant="false" monitorID="8069F9C2-A583-4F4B-89CD-BF9E7C6AF227">
```

Monitor Attributes

Monitor specifications can optionally declare a set of monitor attributes to be reported in any event the Monitor might generate. Monitor attribute declarations work just like program point attributes. Monitor attributes can be declared one at a time or imported as a group from an attribute set. These attributes will appear in any event generated by the Monitor.

Example: Attribute declaration

```
<Attributes>
  <Attribute name="category">Slow Method Call: Slow Database
Connection</Attribute>
  <Attribute name="suggestedAction">ignore</Attribute>
  <SetReference id="slowMethodCallAttrs"/>
</Attributes>
```

A special `noLocation` attribute may be specified to prevent events generated by the monitor from containing stack traces. Collecting stack traces incurs a performance penalty, so disabling that behavior could speed up monitor execution.

Example: noLocation attribute declaration

```
<Attributes>
  <Attribute name="noLocation"/>
</Attributes>
```

Predicate

A popular reason to write a rule is to flag method calls that have particular argument values or that occur in specific contexts. In order to make these rules easy to write without having to create a specialized monitor for each rule, a rule can optionally specify a predicate that must be satisfied before the monitor is invoked. A predicate is evaluated once before the invocation of the method specified by the program point. (The time of predicate evaluation is independent of the control points the monitor watches). If the predicate evaluates to false, none of the monitor's control point methods are invoked. The monitor is effectively canceled.

Predicate syntax is described in the "The Predicate Language" on page 58 section. The following examples demonstrate typical uses for predicates.

Invoke the monitor if the captured value named Input is one of seven numbers as shown in the following example.

Example: Captured value named input

```
<Predicate>
  Input matches /400|403|404|413|414|415|416/
</Predicate>
```

Invoke the Monitor if captured value named name is `jsessionid` (case insensitive) as shown in the following example.

Example: Captured value named “name”

```
<Predicate>
  name matches /(?i)jsessionid/
</Predicate>
```

Do not invoke this Monitor in the context of a web request as shown in the following example.

Example: Web request

```
<Predicate>
  not Request
</Predicate>
```

Invoke the Monitor if the Monitor library method `IsWebOutput` returns `true` for the captured value `out` as shown in the following example.

Example: Captured value “out”

```
<Predicate>
  IsWebOutput(out)
</Predicate>
```

Invoke the monitor if the captured array `values` contain an element that matches the regular expression as shown in the following example.

Example: Captured array values

```
<Predicate>
  values contains value: { value matches /(?i)(\/bin\/(ba)?sh)|cmd\.exe/ }
</Predicate>
```

Configuration

Some Monitors have configurable properties that allow the same Monitor to be used for different purposes by different rules. For example, a rule writer might want to use the timer monitor (`com.fortify.runtime.monitor.Timer`) for two rules to flag long-running database queries.

- Rule 1: flag queries that take longer than 2 second when the database is used in the context of a Web request.
- Rule 2: flag queries that take longer than 10 seconds in the context of a batch processing job.

The timer's `Threshold` property allows different rules to use the same Monitor in different ways, as the following rule excerpts demonstrate.

Example: General construction of a rule that uses the Timer Monitor

```
<Rule>
  <RuleID>Rule1</RuleID>
    ...
  <Configuration>
    <Property name="Threshold">
      <Value>2000</Value>
    </Property>
  </Configuration>
  ...
</Rule>
<Rule>
  <RuleID>Rule2</RuleID>
  ...
  <Configuration>
    <Property name="Threshold">
      <Value>10000</Value>
    </Property>
  </Configuration>
  ...
</Rule>
```

The properties supported by built-in Monitors are described in the "The Default Fortify Runtime Monitor Set" on page 54 section. It is an error to provide an unsupported property to a Monitor as part of a rule.

Bindings

The bindings section connects the values captured in a Program Point to a Monitor's inputs. The bindings section comprises of zero or more `Binding` elements. Each `Binding` element must supply two attributes:

- `name`: the name of the Monitor input.
- `capture-ref`: the id of the capture element declared in the program point.

The Bindings section of a Monitor declaration works closely with the `Capture` section in the program point. The following excerpts from a rule take three values from the target program in the `Capture`section: the object (given `id=”logger”`), the first method argument (given `id=”assert_`

var"), and the second method argument (given id="input_text"). In the Bindings section of the rule these three values are bound to the Monitor inputs Category, Assertvar, and ObjInput.

Example:Bindings

```
<Capture>
  <This id="logger"/>
  <Argument id="assert_var" index="0"/>
  <Argument id="input_text" index="1"/>
</Capture>
...
<Bindings>
  <Binding name="Category" capture-ref="logger"/>
  <Binding name="Assertvar" capture-ref="assert_var"/>
  <Binding name="ObjInput" capture-ref="input_text"/>
</Bindings>
```

It is possible to bind more than one capture element to a single Monitor input field by having multiple Binding entries with the same value for name. In this case, all captured values will be assembled into an array (Object[]) before being passed to the Monitor.

Defining Abstract Monitor Specifications in Monitor Definitions

You can create abstract Monitor specifications in the MonitorDefinitions section of the Rulepack, and then use the abstract specifications inside rules. An abstract Monitor specification is similar in appearance to a regular Monitor specification in a rule except that it requires an id attribute and does not accept the monitorID attribute or the reentrySetID attribute.

The following abstract Monitor specification is set up to look for social security numbers and replace the first 5 digits with XXX-XX.

Example: Abstract monitor

```
<MonitorDefinitions>
  <AbstractMonitorSpec reentrant="false" id="PV_SSN">
    <Predicate>
      outputEnabled and
      Input matches /(?&lt;!\d|-)\d\d\d-\d\d-\d\d\d\d(?!\d|-)/
    </Predicate>
    <Configuration>
      <Property name="ReplaceRegexMap">
        <Map>
          <Entry>
            <Key>(\d\d\d-\d\d-)(\d\d\d\d)</Key>
            <Value>XXX-XX-$2</Value>
```

```
         </Entry>
        </Map>
      </Property>
    </Configuration>
    <Bindings/>
  </AbstractMonitorSpec>
</MonitorDefinitions>
```

You can use an abstract Monitor specification with the Extension element in place of the `Monitor` element inside a rule. The `Extension` element has two required attributes:

- `base`: the id of the abstract Monitor specification to use
- `monitorID`: the Monitor ID for this Monitor specification

An `ExtensionSpec` element accepts all of the same sub-elements and attributes accepted by a `MonitorSpec` element. Any declarations in an `ExtensionSpec` element are added to the declarations in the abstract Monitor specification.

The following rule in the following example uses the abstract Monitor specification declared above.

Example:Rule using abstract Monitor specification

```
<Rule>
  …rule header goes here…
  <Monitors>
    <Extension
        class="com.fortify.runtime.monitor.OrgApacheLog4jGuard"
        base="PV_SSN" monitorID="65AEFB5B-5BC9-438E-BE11-A00C450A56B2">
      <Attributes>
        <Attribute name="category">Privacy: Social Security
Number</Attribute>
        <Attribute name="suggestedAction">rewrite</Attribute>
        <SetReference id="privacyViolationSSNAttrs"/>
      </Attributes>
      <Bindings>
        <Binding name="ObjInput" capture-ref="input_text"/>
        <Binding name="Priority" capture-ref="priority"/>
        <Binding name="Category" capture-ref="logger"/>
      </Bindings>
    </Extension>
  </Monitors>
</Rule>
```

Attribute Sets

Similar rules often use similar sets of attributes. To avoid a great deal of repetition between groups of attributes declared in similar rules, you can create attribute sets in the `AttributeDefinitions` section of the Rulepack, then use a `SetReference` tag to import the attributes as Program Point attributes or monitor attributes.

Example: Attribute Set declaration

```
<AttributeDefinitions>
  <AttributeSet id="privacyViolationCCNAttrs">
    <Attribute name="kingdom">Security Features</Attribute>
    <Attribute
      name="description">/java/privacy_
violation.htm?version=2009.4.0.0016</Attribute>
    <Attribute name="eventType">vulnerability,audit</Attribute>
    <Attribute name="probability">5</Attribute>
    <Attribute name="accuracy">4</Attribute>
    <Attribute name="impact">3</Attribute>
    <Attribute name="impactBias">confidentiality</Attribute>
    <Attribute name="audience">broad,dev</Attribute>
    <Attribute name="primaryAudience">security</Attribute>
    <Attribute name="coveredSCA">no</Attribute>
  </AttributeSet>
</AttributeDefinitions>
```

The Default Fortify Runtime Monitor Set

Fortify Runtime includes three built-in monitors:

- The Fortify Runtime `Guard` monitor type
- The Fortify Runtime `ParameterMonitor` monitor type
- The Fortify Runtime `Timer` monitor type

The Fortify Runtime Guard Monitor Type

The Fortify Runtime monitor class `com.fortify.runtime.monitor.Guard` creates an event when it is triggered. Most rules that use `Guard` create a predicate to determine the conditions under which an event should be created. Guard is able to rewrite its input. "Guard monitor inputs" on the next page table lists guard monitor inputs, while "Guard monitor properties" on the next page table lists guard monitor properties.

Guard monitor inputs

Name	Type	Description
`Input`	`Object`	The single input value captured from the target program.

Guard monitor properties

Name	Type	Description
ReplaceRegexMap	`Map<Regex,String>`	If this property is specified, the regular expressions in this map will be applied to input in the order they are declared. All substrings that match each regular expression will be replaced with the string paired with the regular expression.
TriggerPicture	`String`	A template string which will be used to create an event attribute called `Trigger`, if set. Variables can be specified as `%var` or `%{var}`. Variables may refer to Monitor bindings, event attributes, or Monitor properties.

The following Monitor declaration uses `Guard` to mix up strings that include the substring `dog`. If the input matches `dog`, then `Guard` will create an event. Included in the event will be an attribute named `trigger` with the value of the input. If the rewrite action is triggered on the event, each of the letters `d`, `o`, and `g` will be replaced with a number. Note the syntax for specifying a map as a Monitor property.

A Monitor that uses `Guard` follows in the following example.

Example: Guard monitor

```
</ProgramPoints>
<Monitors>
  <MonitorSpec class="com.fortify.runtime.monitor.Guard" monitorID="m1">
    <Attributes>
      <Attribute name="category">Don't Say SQL</Attribute>
    </Attributes>
    <Predicate><![CDATA[ Input matches /.*dog.*/ ]]></Predicate>
    <Configuration>
      <Property name="ReplaceRegexMap">
        <Map>
          <Entry><Key>d</Key><Value>1</Value></Entry>
          <Entry><Key>o</Key><Value>2</Value></Entry>
          <Entry><Key>g</Key><Value>3</Value></Entry>
        </Map>
      </Property>
```

```
    </Configuration>
    <Bindings>
      <Binding name="Input" capture-ref="x"/>
    </Bindings>
  </MonitorSpec>
```

The Fortify Runtime ParameterMonitor Monitor Type

The Monitor class `com.fortify.runtime.containersupport.ParameterMonitor` provides an easy way to use the named program point parameter to examine HTTP request parameters. The "ParameterMonitor Inputs" below table lists `ParameterMonitor` inputs, while the "ParameterMonitor Monitor Properties" below lists `ParameterMonitor` monitor properties.

ParameterMonitor Inputs

Name	Type	Description
`name`	`String`	The name of the HTTP request parameter.
`values`	`String[]`	The values of the HTTP request parameter.

ParameterMonitor Monitor Properties

Name	Type	Description
`TriggerPicture`	`String`	A template string which will be used to create an event attribute called `Trigger`, if set. Variables can be specified as `%var` or `%{var}`. Variables may refer to Monitor bindings, event attributes, or Monitor properties

This Monitor is from the rule given at the beginning of this document. It uses the `Parameter` named program point to look for command injection attacks.

The following example shows a monitor that uses ParameterMonitor.

Example: ParameterMonitor monitor

```
<MonitorSpec class="com.fortify.runtime.containersupport.ParameterMonitor"
             monitorID="E4531ED3">
  <Attributes>
    <Attribute name="category">Probing: Command Injection</Attribute>
    <Attribute name="triggerHint">name, values</Attribute>
    <Attribute name="suggestedAction">ignore</Attribute>
  </Attributes>
```

```
  <Predicate>
      values contains value: { value matches /(?i)(\/bin\/
(ba)?sh)|cmd\.exe/ }
  </Predicate>
  <Bindings>
    <Binding name="name" capture-ref="name"/>
    <Binding name="values" capture-ref="values"/>
  </Bindings>
</MonitorSpec>
```

The Fortify Runtime Timer Monitor Type

The Monitor class `com.fortify.runtime.monitor.Timer` reports method calls that take a long time to execute.

Timer takes no inputs.

The following table lists timer properties.

Timer Properties

Name	Type	Description
`Threshold`	`int`	The minimum amount of time (given in milliseconds) a method must execute before Timer will generate an event upon method completion. This property is passed through the configuration variable interpreter before it is used, so the rule writer can provide values such as `%MyConfigVar` and the threshold for a rule set in the configuration file.
`TriggerPicture`	`String`	A template string which will be used to create an event attribute called `Trigger`, if set. Variables can be specified as `%var` or `%{var}`. Variables may refer to Monitor bindings, event attributes, Monitor properties, or the special value threshold, which contains the current threshold setting. The default value is "Method call took longer than `%{time}` milliseconds, which exceeds current threshold of `%{threshold}` milliseconds."

The following example illustrates a Fortify Runtime Monitor of class `Timer`.

Example:A MonitorSpec that uses Timer

```
<MonitorSpec class="com.fortify.runtime.monitor.Timer"
  reentrant="false" monitorID="8069F9C2-A583-4F4B-89CD-BF9E7C6AF227">
```

```
  <Attributes>
    <Attribute name="category">Slow Method Call: Slow Database
Connection</Attribute>
    <Attribute name="suggestedAction">ignore</Attribute>
    <SetReference id="slowMethodCallAttrs"/>
  </Attributes>
  <Configuration>
    <Property name="Threshold">
      <Value>${DatabaseConnectionTimeThreshold}</Value>
    </Property>
    <Property name="TriggerAttributeName">
      <Value>Trigger</Value>
    </Property>
  </Configuration>
  <Bindings/>
</MonitorSpec>
```

The Predicate Language

The predicate language gives rule writers easy access to values captured by a rule and functions defined in monitor libraries. It is a simple language. There are no explicit constructs for modifying control flow, no variable declarations (except for iterating over collections), and no user-defined functions or types.

Syntax

The syntax for the predicate language is given in the following example.

Example:BNF Syntax for the Predicate Language

```
pred ::= pred and pred |
       pred or pred |
       not pred |
       expr contains Name ‘:’ ‘{’ pred ‘}’ |
       expr matches Regex |
       ‘(’ pred ‘)’ |
       exp

exp ::= var |
       call |
       String |
       Integer
```

```
var ::= Name |
       Name ‘.’ Name |
       Name‘[’ String ‘]’

call ::= Name ‘(’ { arglist } ‘)’

arglist ::= { exp ‘,’ } exp
```

A `String` is a sequence of characters surrounded by quotation marks (`"this is a string"`).

A `Name` (used for naming variables, fields and functions) is a sequence of one or more letters, numbers, and ampersands. Names are case-insensitive, so the expressions `Request.Path` and `request.path` are equivalent.

A `Regex` is a Java regular expression surrounded by slashes, such as `/[a-zA-Z][a-zA-Z0-9]*/`.

An `Integer` is a positive or negative integer.

Predicates are evaluated from left to right. All predicate operators are short-circuit operators. In other words, a predicate will be evaluated until its value can be determined at which point evaluation will cease. Given the predicate `A` or `B`, if `A` evaluates to `true`, then `B` will not be evaluated.

Predicate operator precedence from highest to lowest is `not`, `and`, `or`.

The `contains` operator iterates over a collection and evaluates to `true` if the inner predicate evaluates to `true` for one of the members of the collection. If `contains` is used with a map, it will iterate over the map values and ignore the map keys.

The `matches` operator interprets its expression as a string and returns `true` if its regular expression matches the string. Note that the regular expression need not match the entire string; to match an entire string, include anchors in the regular expression (for example, `/^test$/`).

The expressions `o.x` and `o["x"]` are equivalent.

Types

Expressions can be one of three types: Boolean, integer or reference. An expression used as a predicate evaluates to `true` if it has the Boolean value true, if it is an integer (of any value), or if it is a non-null reference. For example, the variable `Request` will be null if no HTTP request is available, so the predicate `not Request` creates a rule that will only fire outside the context of a web request.

Example:Using an expression as a Predicate

```
<Predicate> not Request </Predicate>
```

A reference expression can be null or can refer to an object. The built-in reference types are `String` and the collection types `List`, `Map`, and `MultiMap`.

It is illegal to use the `matches` operator with any expression type other than a reference to a string. The `contains` operator must be used with an expression with a collection type. It is only legal to access fields or indices of a reference expression.

Taking a field or index of a null reference results in a null reference. Accessing a field or index that does not exist results in a null reference.

For a reference to a `Map` or `MultiMap`, a special field named `@Keys` returns a collection containing the keys used in the map.

A predicate that performs an illegal operation evaluates to `false`.

Variables

Predicates may use monitor bindings as variables. In the following example, the predicate uses the binding `out` to check to see if an output stream is writing to an HTTP response.

Example:Using a Monitor Binding as a Predicate Variable

```
<MonitorSpec
     class="com.fortify.runtime.monitor.Guard" reentrant="false"
monitorID="MB0D">
  <Attributes/>
  <Predicate>
     IsWebOutput(out)
  </Predicate>
  <Configuration/>
  <Bindings>
    <Binding name="out" capture-ref="captured_out"/>
  </Bindings>
</MonitorSpec>
```

A Special Variable: Request

The special variable `Request` refers to the current HTTP request or is null if no request is currently being processed by the thread that triggered the rule. `Request` fields are described in the following table.

Variable fields for Request variable

Name	Type	Description
`Parameters`	`map`	All of the parameters read by the program thus-far. If the program has not yet requested a parameter, it does not appear in this map.

Variable fields for Request variable, continued

Name	Type	Description
`Cookies`	`map`	A map of all of the cookies sent with the request.
`Headers`	`map`	A map of all of the headers sent with the request.
`Path`	`string`	The path component of the request URL.
`Host`	`string`	The host name of the interface on which the request was received.
`Port`	`string`	The network port on which the request was received.
`QueryString`	`string`	The complete query string for the request.
`Scheme`	`string`	The scheme components of the request URL (`http`, `https`, etc.)
`Session`	`map`	A map representing the contents of the HTTP Session object.
`SessionId`	`string`	The session identifier.
`RemoteUser`	`string`	The name of the remote user as represented in the request.
`RemoteAddress`	`string`	The IP address of the remote user.
`Method`	`string`	The HTTP method of the request (`GET`, `POST`, etc.).

Rules Scenarios

This section provides scenarios that include Rules capability. The format of the scenarios states a common business problem followed by the recommended solution. The scenarios in this section are:

- Viewing and Changing HPE Security Fortify Rules
- Making a Whitelist Rule
- Capturing a Boolean

Viewing and Changing HPE Security Fortify Rules

Problem

The default behavior you get with Fortify Runtime Rulepack is almost what you want, but not quite. You would like to understand how the rule works and maybe edit it a little bit, but the RPR file isn't humanly readable.

Solution

A lot of problems that are specific to a particular program or environment can be remedied with event handlers or other configuration settings, but sometimes writing rules is the only way to go. HPE Security Fortify does not include the XML of its rules with Fortify Runtime distribution because most customers don't require it, but we will provide the XML on an as-needed basis. Customers can then understand our rules in more detail and create their own custom rules based on ours.

Making a Whitelist Rule

Problem

There's a business logic flaw behind a particular web page and you want to use Fortify Runtime to filter out requests that could tickle the flaw.

Solution

Write a rule that only allows known and good input for the parameter in question. The following rule will only allow numbers, letters, period, forward slash, and dash to appear in the parameter path for `downloadLanding.jsp`.

Example: Whitelist Rule

```
<?xml version="1.0"?>
<RulePack xmlns="xmlns://www.fortifysoftware.com/schema/runtime/rules">
  <RulePackID>123</RulePackID>
  <SKU/>
  <Name>custom rules</Name>
  <Version>1</Version>
  <Language>java</Language>
  <Description/>
  <Rules>
    <RuleDefinitions>
      <Rule>
        <RuleID>8EA408CA-77A5-47B4-883C-615E13DE7A53</RuleID>
```

```
        <ProgramPoints>
          <ProgramPoint>
            <Name>Parameter</Name>
            <Capture/>
          </ProgramPoint>
        </ProgramPoints>
        <Monitors>
          <MonitorSpec
class="com.fortify.runtime.containersupport.ParameterMonitor"
                       monitorID="90C859DE-9D2F-4F18-A207-70F193B8744F">
            <Attributes>
              <Attribute name="category">Invalid File Name</Attribute>
              <!-- ... -->
            </Attributes>
            <Predicate>
              name matches /(?i)path/ and
              values contains value: {
                value matches /[^a-zA-Z0-9_\.\/\-]/
              } and
              Request.Path matches /(?i).*downloadLanding.jsp/
            </Predicate>
            <Configuration>
              <Property name="TriggerPicture">
                <Value>name = %{name}, values = %{values}</Value>
              </Property>
            </Configuration>
            <Bindings>
              <Binding name="name" capture-ref="name"/>
              <Binding name="values" capture-ref="values"/>
            </Bindings>
          </MonitorSpec>
        </Monitors>
      </Rule>
    </RuleDefinitions>
  </Rules>
</RulePack>
```

Capturing a Boolean

Problem

You need to monitor a method that returns a `boolean`. There is a way to create one kind of event if the method returns `true` and a different kind of event if it returns `false`.

Solution

The following rule in the example below monitors the method `changePassword()` in `com.fortify.samples.riches.model.AccountService`. The method returns a `boolean`, which the rule captures like the following:

```
<Return id="change"/>
```

The rule then defines two monitors, one to create an event when the method returns `true`, the other to create a different event when the method returns `false`. The first monitor has the predicate `Change`. The second monitor has the predicate `not Change`.

Example: Capture a boolean

```
<?xml version="1.0" encoding="UTF-8"?>
<RulePack xmlns="xmlns://www.fortifysoftware.com/schema/runtime/rules">
  <RulePackID>917F6D48-DFE8-45BB-882B-8E50F89C99FE</RulePackID>
  <SKU/>
  <Name>Log Riches Password Change</Name>
  <Version>2010.2</Version>
  <Language>java</Language>
  <Description/>
  <Rules>
    <RuleDefinitions>
      <Rule>
        <RuleID>4B44CD80-D1D7-4778-AA32-89FB75C07BC2</RuleID>
        <ProgramPoints>
          <ProgramPoint>
            <Method>
              <Class subclasses="true"

>com\.fortify\.samples\.riches\.model\.AccountService</Class>
              <Name>changePassword</Name>
              <Parameters>
                <ParameterType type="java\.lang\.String"/>
                <ParameterType type="java\.lang\.String"/>
                <ParameterType type="java\.lang\.String"/>
```

```
              <ParameterType type="java\.lang\.String"/>
            </Parameters>
          </Method>
          <Capture>
            <Return id="change"/>
            <Argument id="username" index="0"/>
          </Capture>
        </ProgramPoint>
      </ProgramPoints>
      <Monitors>
        <!--  Monitor 1: If the user successfully changed his password,
                         then we want to know about that -->
        <MonitorSpec class="com.fortify.runtime.monitor.Guard"
                     reentrant="false"
                     monitorID="8C8C141C-C874-4E75-8E8C-C7E431906A8F" >
          <Attributes>
            <Attribute name="category">Successful Password
Change</Attribute>
            <Attribute name="suggestedAction">ignore</Attribute>
          </Attributes>
          <Predicate>
            Change
          </Predicate>
          <Configuration>
            <Property name="TriggerPicture">
              <Value>%{Input} successfully changed password</Value>
            </Property>
          </Configuration>
          <Bindings>
            <Binding name="Input" capture-ref="username"/>
            <Binding name="Change" capture-ref="change"/>
          </Bindings>
        </MonitorSpec>
        <!--  Monitor 2: If the user tried to change his password, but
the
                         old password was set incorrectly, then we want
to know
                         about that -->
        <MonitorSpec class="com.fortify.runtime.monitor.Guard"
                     reentrant="false"
                     monitorID="81E799A2-36F0-446A-8CA9-D5D9B60D3845" >
          <Attributes>
```

```
              <Attribute name="category">Failed Password Change
Attempt</Attribute>
              <Attribute name="suggestedAction">ignore</Attribute>
            </Attributes>
            <Predicate>
              not Change
            </Predicate>
            <Configuration>
              <Property name="TriggerPicture">
                <Value>Attempt of user %{Input} to change password,
                       but old password was incorrect</Value>
              </Property>
            </Configuration>
            <Bindings>
              <Binding name="Input" capture-ref="username"/>
              <Binding name="Change" capture-ref="change"/>
            </Bindings>
          </MonitorSpec>
        </Monitors>
      </Rule>
    </RuleDefinitions>
    <MonitorDefinitions/>
    <ProgramPointDefinitions/>
    <AttributeDefinitions/>
  </Rules>
</RulePack>
```

Chapter 5: Writing Monitors

This section contains the following topics:

Introduction to Fortify Runtime Monitors

A monitor is a Java class for observing a target program. Fortify Runtime attaches monitors to the target program as directed by rules. A monitor can create events, modify the state of the target program, modify the control flow of the target program, and call into services provided by Fortify Runtime. Some monitors are general-purpose and can be used in different ways by different rules. Other monitors carry out a specialized operation and can only be used for a single purpose. These specialized monitors often appear in only a single rule.

Fortify Runtime comes with some monitors built in, but you can create your own monitors too. Because most target programs were not written with monitors in mind, monitor writers must take special care to avoid unintended side-effects.

This chapter describes the relationships between rules, Fortify Runtime, and monitors. It explains the frameworks and conventions that govern monitors and documents the supporting classes available from inside a monitor. It assumes the reader is familiar with Fortify Runtime configuration and rule writing.

A sample monitor is included in the Fortify Runtime SDK under the directory

```
/sdk/samples/CustomMonitor
```

Rules, Monitors, and Fortify Runtime

A rule binds a set of Monitors to a set of program points. Fortify Runtime creates a new monitor class at runtime for each binding. When it creates a monitor class, Fortify Runtime writes property and attribute values from the rule into the new monitor class.

When the target program executes the program point specified in a rule, Fortify Runtime will create a new instance of the appropriate monitor class to observe the execution. Fortify Runtime invokes special monitor methods called control methods before or after the execution of the program point. Inside a control method the monitor can examine property and attribute values from the rule and values captured from the target program.

Restrictions and Requirements

A Monitor class must directly extend `java.lang.Object`. It must have the marker annotation for monitor classes, `@com.fortify.runtime.Monitor`. A Monitor class may implement the `com.fortify.runtime.Rewriter` interface, which allows a monitor to carry out the rewrite action.

In order to avoid classloading problems, monitor classes can make use of only methods and objects from Fortify Runtime (`com.fortify.runtime.*`) and from the Java runtime library (`java.*, javax.*`), but not from any other libraries. This restriction does not apply to manipulating objects taken from the execution context of the Monitor; methods and members of those objects can be referenced directly. For example, a Monitor class that watches `log4j` methods can take a `log4j` Logger object as an argument, but no monitor class can ever use `log4j` for the purposes of doing its own logging.

Rule Properties Become Static Member Variables

When program execution begins, the Fortify Runtime Platform automatically binds static Monitor member variables to property values given in a rule. Static variables declared in a Monitor are not shared between monitors for different rules—each rule gets its own clone of the monitor class with its own static variables. If rules `R1` and `R2` both use Monitor class `M`, Fortify Runtime will create two classes from `M: M1` and `M2`. `M1` will have static variables set from the properties in `R1`, and `M2` will have static variables set from the properties in `M2`. Property fields may be declared `final` and must be declared `public`. They must not have initializers; if they do, the value of the initializer will be used at runtime rather than the value of property declared by the rule.

A static field named `Attributes` will receive special treatment—it will be populated from the monitor attributes portion of the rule. It can be declared like this:

```
private final static Map<String,String> Attributes;
```

Static variables `TargetClassName` and `TargetMethodName` are also special cases. If declared, they will be populated with the class and method names of the specific method that the program point attached to.

```
private static String TargetClassName;
```

```
private static String TargetMethodName;
```

These fields must be declared private so that it is not confused with a property field.

An Example Monitor

The example below shows a Monitor that prints properties and attributes from a rule. The Monitor demonstrates the variety of property types a Monitor can declare.

Example: Java program that constructs a Fortify Runtime Monitor

```
import java.util.Map;
import java.util.List;
import java.util.Collection;
import com.fortify.runtime.Monitor;
@Monitor
public class MyMonitor {
  private static Map<String, String> Attributes;

  public static String string;
  public static int I;
  public static long L;
  public static float F;
  public static double D;
  public static short S;
  public static char C;
  public static boolean Z;
  public static byte B;
  public static Object object;
  public static List<String> listString;
  public static Collection<String> collectionString;
  public static String[] arrayString;
  public static Map<String, String> mapString;

  public void before() {
    System.out.println("::hit rule: " + Attributes.get("RuleID"));
    System.out.println("rule attributes:");
    for (Map.Entry<String, String>  e: Attributes.entrySet())
      System.out.println("  " + e.getKey() + " : " + e.getValue());
    System.out.println(string);
    System.out.println(I);
    System.out.println(L);
    System.out.println(F);
    System.out.println(D);
    System.out.println(S);
    System.out.println(C);
    System.out.println(Z);
    System.out.println(B);
    System.out.println(object);
    for (String av : listString)
      System.out.print(av + " ");
    System.out.println();
```

```
    for (String av : collectionString)
      System.out.print(av + " ");
    System.out.println();
    for (String av : arrayString)
      System.out.print(av + " ");
    System.out.println();
    for (Map.Entry<String, String>  e: mapString.entrySet())
      System.out.print(e.getKey() + " : " + e.getValue() + ", ");
    System.out.println();
  }
}
```

A Rule that References the Example Monitor

The following example shows a rule that uses the above illustrated example Monitor.

Example: Fortify Runtime Platform rule that references the Example Monitor

```
<Rule>
  <RuleID>PropertyPrintRule</RuleID>
  <ProgramPoints>
    <ProgramPoint>
      <Method>
        <Class>Main</Class>
        <Name>method0</Name>
      </Method>
      <Capture/>
    </ProgramPoint>
  </ProgramPoints>
  <Monitors>
    <Monitor monitorID="mon1" class="MyMonitor">
      <Attributes/>
      <Configuration>
        <Property name="string">
          <Value>A</Value>
        </Property>
        <Property name="I">
          <Value>1</Value>
        </Property>
        <Property name="L">
          <Value>-2</Value>
        </Property>
```

```
        <Property name="F">
          <Value>3.14159</Value>
        </Property>
        <Property name="D">
          <Value>-5.55555</Value>
        </Property>
        <Property name="S">
          <Value>6</Value>
        </Property>
        <Property name="C">
          <Value>c</Value>
        </Property>
        <Property name="Z">
          <Value>true</Value>
        </Property>
        <Property name="B">
          <Value>127</Value>
        </Property>
        <Property name="object">
          <Value>0</Value>
        </Property>
        <Property name="listString">
          <List>
            <Value>a</Value>
            <Value>b</Value>
            <Value>c</Value>
          </List>
        </Property>
        <Property name="collectionString">
          <List>
            <Value>alpha</Value>
            <Value>bravo</Value>
            <Value>charlie</Value>
          </List>
        </Property>
        <Property name="arrayString">
          <List>
            <Value>z</Value>
            <Value>y</Value>
            <Value>x</Value>
          </List>
        </Property>
```

```
        <Property name="mapString">
          <Map>
            <Entry>
              <Key>one</Key>
              <Value>1</Value>
            </Entry>
            <Entry>
              <Key>two</Key>
              <Value>2</Value>
            </Entry>
            <Entry>
              <Key>three</Key>
              <Value>3</Value>
            </Entry>
          </Map>
        </Property>
      </Configuration>
      <Bindings/>
    </Monitor>
  </Monitors>
</Rule>
```

Rule Bindings Become Member Variables

Before the Fortify Runtime Platform invokes a control point method, it binds Monitor member variables to values from the target program as specified by in the bindings section of the rule. In this way the monitor can observe values from the target program. If a monitor specifies that values from the target program should be modified, the Fortify Runtime Platform will wait until the monitor is finished executing, read values back from member variables, and set the appropriate values in the target program.

An argument field must be declared `public` and must not be declared `final`. The type of the declared field must be assignable from the types that it is bound to in the rules, but the Fortify Runtime Platform will automatically handle boxing/unboxing conventions (for example, `int` to `Integer` or `Integer`to `int`).

Some specially named member fields will be given special treatment:

- A member field named `ProgramPointAttributes` will be populated from the attributes portion of the program point.
- A member field named `Configuration` will be populated with a reference to the `com.fortify.runtime.config.RuntimeConfiguration` under which the Monitor runs. This is the only correct way to get a reference to the `RuntimeConfiguration` object in a Monitor.

These special fields can be declared as follows:

- `private Map<String,String> ProgramPointAttributes;`
- `private RuntimeConfiguration Configuration;`

These fields must be declared `private` to distinguish them from fields which are available for binding in the rule.

The example in the example below uses Monitor member variables for Program Point attributes and to capture values from the target program. A rule that uses the Monitor follows in the next example.

Example:Monitor using member variables for Program Point attributes

```
import java.util.Map;
import java.util.List;
import java.util.Collection;
import com.fortify.runtime.Monitor;

@Monitor
public class MySecondMonitor {
  private Map<String, String> programPointAttributes;
  public String stringArg;
  public int intArg;
```

```
<Rule>
  <RuleID>Method1Rule</RuleID>
  <ProgramPoints>
    <ProgramPoint>
      <Attributes>
        <Attribute name="Jerry">Tom</Attribute>
      </Attributes>
      <Method>
        <Class>Main</Class>
        <Name>method1</Name>
      </Method>
      <Capture>
        <Argument id="firstCapture" index="0"/>
        <Argument id="secondCapture" index="1"/>
      </Capture>
    </ProgramPoint>
  </ProgramPoints>
  <Monitors>
    <Monitor monitorID="mon2" class="MySecondMonitor">
      <Attributes/>
      <Bindings>
        <Binding name="stringArg" capture-ref="firstCapture"/>
        <Binding name="intArg" capture-ref="secondCapture"/>
```

```
      </Bindings>
    </Monitor>
  </Monitors>
</Rule>
```

Watching the Target Program with Control Points

A Monitor can receive control at four points around a method call: before method execution, after method execution, when variable binding is complete, or when an exception is thrown. The Monitor declares interest in receiving control at a particular control point by declaring a method named after the point. A single Monitor can observe more than one control point. The control points are:

- **before** – Called before the method begins executing. Method arguments will be available, but return value of the method is not available because the method has not yet been invoked. A Monitor declares interest in this control point by declaring a method with one of the following two signatures:
  - `public void before()` - Receive control before execution without the ability to alter arguments to the method.
  - `public boolean before()` - Receive control before execution, with the option to alter the argument values delivered to the method. Argument values can be altered by setting the corresponding argument fields and returning `true` from the method.
- **after** – Called after normal return. The method return value is available at this point. A Monitor declares interest in after by declaring a method with one of the following two signatures:
  - `public void after()` - Receive control after execution without the ability to alter the return value from the method.
  - `public boolean after()` - Receive control after execution with the option to alter the return value from the method. The return value will be altered by setting the corresponding argument field and having the method return true.
- **complete** – Called after all Monitor properties declared in a rule have been bound to values from the target program. If the Monitor is not bound to a return value, `complete` will be called before method execution. If it is bound to a return value, `complete` will be called after method execution. Monitors that declare a `complete` control point cannot declare `before` or `after`. When using `complete` and modifying values in the target program, the creator of the Monitor is responsible for ensuring that it is possible to modify that value at the time the control point is executed. For example, it is not possible to alter the parameters passed to a method after the method has already been invoked, so a Monitor watching `complete` should ensure it does is not looking at the return value of a method before it attempts to alter method arguments.
  A Monitor declares interest in complete by declaring a method with one of the following two signatures:
  - `public void complete()` - Receive control after all arguments have been bound.
  - `public boolean complete()` - Receive control after all arguments have been bound, with the option to alter parameter values or the method return value.

- **exception** – Called if the method exits by throwing an exception. A Monitor declares interest in exception by declaring a method with the following signature:
  - `public void exception(java.lang.Throwable t)` - Receive control on abnormal return.

Changing Control Flow in the Target Program

Any exception generated during the execution of a control point method will be caught and handled by Fortify Runtime and will not be propagated to the calling code. This prevents an unexpected exception during execution of a monitor method from unintentionally derailing the target program.

The special class `com.fortify.runtime.MonitorException` allows a control point method to change the flow of control in the target program in one of two ways:

- By causing an exception to be thrown in the target program.
- By causing an early return from the monitored method in the target program.

Because `MonitorException` is a checked exception, a control point method must declare it throws `MonitorException` before it can change the control flow of the target program.

All control points can cause an exception to be thrown in the target program by wrapping the exception inside a `MonitorException` created by the method `MonitorException.mkThrow()`. For example, to cause a `SecurityException` to be thrown in the target program, write the following.

Example: SecurityException

```
SecurityException se = new SecurityException();
throw MonitorException.mkThrow(se);
```

To cause an early return from `before` thereby skipping the normal execution of the monitored method in the target program, throw the return value of `MonitorException.mkReturn()` (for methods that return void) or throw the return value of `MonitorException.mkReturn(value)` (for methods that have a return type). For primitive return types, the value should be an object that can be unboxed to the return type (for example, `Integer` for `int`).

Example: Early return skipping normal execution of the monitored method

```
int rval = -1;
throw MonitorException.mkReturn(rval);
```

If a Monitor implements the exception control point, by default the exception from the target program will continue to propagate up the call stack after the monitor returns control to the target program. To quash the exception and return normally, the `exception()` method can use the method `MonitorException.mkReturn()`, as in:

```
throw MonitorException.mkReturn();
```

Alternately, throwing the value returned by `mkThrow()` from inside the exception control point will cause the new exception to be thrown and the original exception to be ignored.

Classes Provided by the Fortify Runtime Platform

Fortify Runtime provides classes for monitors to do all of the following:

- Create events
- Execute actions
- Write to the system log
- Examine the Fortify Runtime configuration
- Read information from an HTTP request
- Associate information with an HTTP session
- Read and write data tags associated with arbitrary objects.

See the JavaDoc packaged with Fortify Runtime under `/sdk/javadoc` for a detailed description of all classes.

Adding a New Predicate Library

Predicate expressions provide a flexible way for a rule to use a general-purpose Monitor in a specific way. Predicate expressions can evaluate simple checks (such as regular expression matches) against values taken from the program point context to determine if the monitor should be executed or not.

Predicate libraries provide a way to extend the capabilities of predicates. A predicate library is a java class with the special marker annotation `@com.fortify.runtime.PredicateLibrary`. Any public static methods in such a class can be accessed by predicate expressions. For instance, a predicate method might be created to check if an input string is a palindrome (a property that can't be checked with a simple regular expression), as in the following example.

Example:Predicate method to check input string

```
import com.fortify.runtime.PredicateLibrary;
@PredicateLibrary
public final class PalindromeChecker {
  public static boolean IsPalindrome(String input) {
    if (input == null)
      return false;
    for (int i = 0; i < input.length()/2; ++i)
    {
      if (input.charAt(i) != input.charAt(input.length()-1-i))
        return false;
    }
    return true;
  }
}
```

Compiling and Executing a Monitor

Custom Monitors can be included into the Fortify Runtime Platform in two ways. The Monitor classes can be referenced in the configuration file, or monitor classes can be bundled with rules in an RPR file.

Referencing Monitors from a Configuration File

You can use the `<MonitorLibrary>` tag in the rules section of the Fortify Runtime Platform configuration file to make new Monitors available to the Fortify Runtime Platform. A Monitor library can be a Java jar file or a directory of Java classes. Monitor libraries must be specified before rules files. You could add a jar file of monitor classes like this:

```
<MonitorLibrary>my_monitors.jar</MonitorLibrary>
```

Bundling Monitors with Rules in an RPR File

An RPR file is a rule bundle for Fortify Runtime. An RPR bundles three items:

- A java-style properties file - Optional
- An xml Rulepack - Required
- A monitor library file - Optional

The Rulepack and library content may (optionally) be compressed using `gzip`. The properties section is always in plaintext, and is generally used to record metadata about the RPR. RPR files can be created and extracted using the `rpar` utility. Its usage borrows from the tar utility.

Example: rpar utility command line syntax

```
Create rpr file: rpar cf rpr-file [ -p properties-file ] rulepack-file [
lib-file ]
Extract rpr file: rpar xf [ -d outdir ] rpr-file
List rpr file contents: rpar tf rpr-file
```

Without the `f` flag to specify a file, the `rpar` utility reads and writes from `stdout/stdin`.

Example Source Code for a Complete Monitor

Following is the source listing for `com.fortify.runtime.monitor.Guard`. It is an example of a general-purpose monitor that creates events and carries out actions as directed by rules and configuration.

Example: Example of a complete Fortify Runtime Platform Monitor

```
package com.fortify.runtime.monitor;
```

```
import com.fortify.runtime.Rewriter;
import com.fortify.runtime.MonitorException;
import com.fortify.runtime.Monitor;
import com.fortify.runtime.lib.FortifyRuntime;
import com.fortify.runtime.event.Event;
import com.fortify.runtime.logging.Logger;
import com.fortify.runtime.logging.LogFactory;
import com.fortify.runtime.config.RuntimeConfiguration;
import com.fortify.util.Pair;

import java.util.regex.Pattern;
import java.util.Map;
import java.util.List;
/**
 * Guard is a utility monitor which evaluates a single input and fires an
event.
 * It is capable of rewriting its input in response to the event.
 */
@SuppressWarnings({"UnusedDeclaration" or
"MismatchedQueryAndUpdateOfCollection"})
@Monitor
public class Guard implements Rewriter {

  // Attributes
  private static Map<String, String> Attributes;

  // Configuration Properties
  public static Map<String, String> ReplaceRegexMap;
  public static String TriggerPicture;

  // Configuration Handle
  private RuntimeConfiguration Configuration;

  // Local static fields
  private static final Logger Log = LogFactory.getLogger(Guard.class);
  private static final List<Pair<Pattern, String>> Replacements;

  // Runtime Bindings
  public Object input;

  static {
```

```
    // compile all of the keys in ReplaceRegexMap into regex patterns
    Replacements =
      MonitorUtils.CompileRegexMap(ReplaceRegexMap, Attributes.get
("RuleID"));
  }


  public boolean complete() throws MonitorException {
    // This method will be called every time the rule's predicate is
satisfied.
    // Time to create an event
    Event event = FortifyRuntime.createEvent(Attributes);

    // Send off the event.  Because this method might throw an exception,
    // it should be the end of the ctl point method
    return Configuration.dispatch(event, this);
  }

  public boolean rewrite() {
    if (ReplaceRegexMap == null) {
      Log.error("Rewrite cannot proceed without " +
                "a defined set of replacement patterns in
ReplaceRegexMap");
      return false;
    }
    if (input != null) {
      input = MonitorUtils.RewriteValue(input, Replacements);
      return true;
    }
    return false;
  }
}
```

Chapter 6: Writing Filter Classes

This section contains the following topics:

Introduction to Fortify Runtime Filter Class

A filter class is a Java class for modifying an event during processing by the event handler chain. See the "Writing Event Handlers" on page 18 chapter, `<Filter>` section for information about how to use a filter as part of the event handler chain.

Creating a Filter Class

A filter class must implement the interface `com.fortify.runtime.event.EventFilter`. Filter classes can make use of only methods and objects from Fortify Runtime (`com.fortify.runtime.*`) and from the Java runtime library (`java.*`, `javax.*`).

Installing a Filter Class

Filter classes are made available at runtime by including the compiled class files in a monitor library. See Chapter 4 for details on creating a monitor library.

Referencing a Filter Class

If a filter class carries the optional annotation `EventFilterName`, then the value of the annotation can be used to reference the class in the configuration. For example, the annotation `@EventFilterName ("BigFilter")` allows for the tag `<Filter name=”BigFilter”/>`. If a filter does not carry a name annotation, then an event handler can invoke a filter class by referencing the fully qualified class name. The class `com.example.MyFilter` can be invoked with the tag `<Filter class=”com.example.MyFilter”/>`.

An Example Filter Class

The source code for the sample filter given here is also included in the Fortify Runtime SDK under the directory `/sdk/samples/EventFilter`.

The filter class allows an event handler to remove sensitive information from an event so that it isn't logged or further distributed. The example below shows how the filter class can be used on the event handler chain. It is configured to replace attributes that carry credit card numbers with a series of Xs.

Example: Using the Example Filter

```
<Filter class="SensitiveDataFilter">
    <Setting name="\d{16}|\d{4}([\s-])\d{4}\1\d{4}\1\d
{4}">XXXXXXXXXXXXXXXX</Setting>
</Filter>
```

The following example lists the source code for the filter class.

```
import com.fortify.runtime.config.RuntimeConfiguration;
import com.fortify.runtime.event.Event;
import com.fortify.runtime.event.EventFilter;
import com.fortify.runtime.monitor.MonitorInstance;
import com.fortify.runtime.record.*;
import com.fortify.runtime.record.LongValue;
import com.fortify.runtime.record.StringValue;
import com.fortify.runtime.record.ValueList;
import com.fortify.runtime.util.MonitorUtils;

import java.lang.Object;
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/*
 * An event filter that removes sensitive data from events
 * This filter implementation is included in the default runtime platform
filter set
 * and can be accessed via the filter name "StripSensitiveData"
 * The code is reproduced here for demonstration purposes
 */
public class SensitiveDataFilter implements EventFilter {
```

```
  public void configure(Map<String, String> settings) {
    // consider each setting to be a pattern/value pair for replacement
    replacementMap = new HashMap<Pattern, String>();
    for (Map.Entry<String, String> entry : settings.entrySet()) {
      try {
        replacementMap.put(Pattern.compile(entry.getKey()), entry.getValue
());
      } catch (PatternSyntaxException e) {
        throw new UserError(MessageFormat.format(
        "StripSensitiveData filter setting contains an invalid regular
expression:" +
        "'{0}'"   entry.getKey()), e);
      }
    }
  }

  private Map<Pattern, String> replacementMap;

  public Event filterEvent(Event event, MonitorInstance monitor,
                           RuntimeConfiguration configuration) {
    Record attributes = event.attributes();
    // traverse the record structure and run replacement on any string
values
    processRecord(attributes);
    return event;
  }

  private void processRecord(Record r) {
    for (Field field : r.getFields()) {
      processField(field);
    }
  }

  private void processField(Field f) {
// never filter id fields
        String label = f.getLabel().raw();
        if (label.equals("RuleID")) return;
        if (label.equals("MonitorID")) return;
        if (label.equals("eventId")) return;
        if (label.equals("configurationId")) return;
    f.setValue(processValue(f.getValue()));
```

```
  }

  private Value processValue(Value v) {
    if (v instanceof Record) {
      processRecord((Record)v);
      return v;
    }
    if (v instanceof ValueList) {
      return processValueList((ValueList)v);
    }
    if (v instanceof StringValue) {
      return processStringValue((StringValue)v);
    }
    if (v instanceof LongValue) {
      return processLongValue((LongValue)v);
    }
    throw new RuntimeException("Unexpected value type " + v);
  }

  private ValueList processValueList(ValueList l) {
    ValueList result = new ValueList();
    for (Value v : l.getValues()) {
      result.add(processValue(v));
    }
    return result;
  }

  private StringValue processStringValue(StringValue v) {
    String result = v.raw();
    for (Map.Entry<Pattern, String> entry : replacementMap.entrySet()) {
      Pattern p = entry.getKey();
      Matcher m = p.matcher(result);
      if (m.find()) {
        StringBuffer replaced = new StringBuffer();
        do {
          m.appendReplacement(replaced, entry.getValue());
        } while (m.find());
        m.appendTail(replaced);
        result = replaced.toString();
      }
    }
    return new StringValue(result);
```

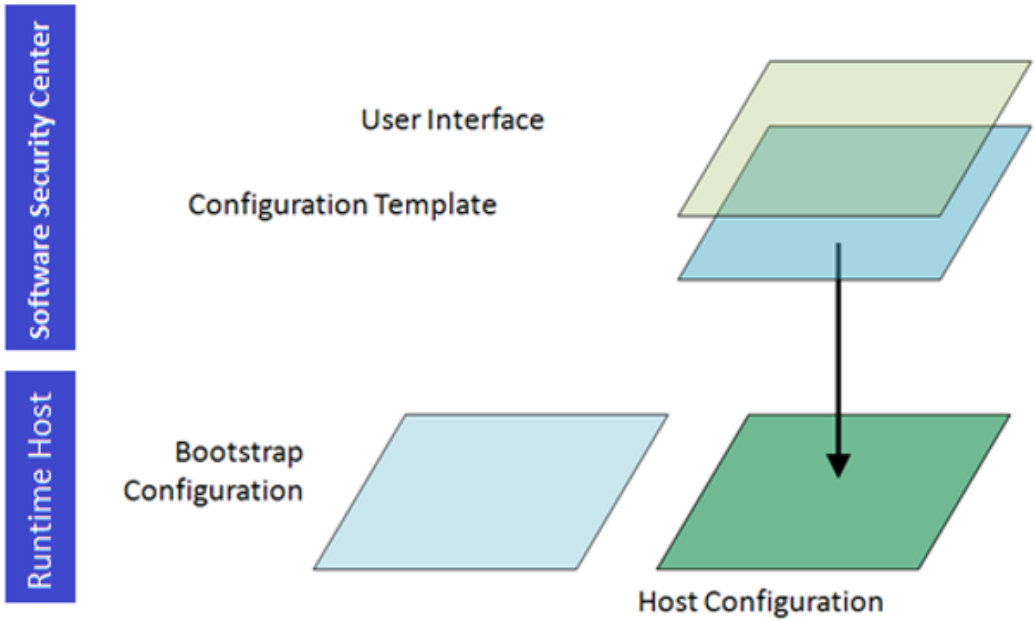
```
  }

  private LongValue processLongValue(LongValue v) {
    // assume that numerical values are not sensitive
    return v;
  }

}
```

Chapter 7: Writing a Federation Configuration Template and Bootstrap Configuration File

This section contains the following topics:

Federation Configuration Templates are baseline Host configurations stored on a Federation controller (HPE Security Fortify Software Security Center).

As illustrated below, when you use the HPE Security Fortify Software Security Center server UI to customize a configuration, the host configuration is the sum of the Configuration Template and the UI settings. The Federation Controller sends the configuration to all the Fortify Runtime Hosts in that Federation.



A Federation Configuration Template is similar to a Standalone Mode configuration, but with some important differences:

- Hosts respect some additional global settings in Federated Mode. A small number of global settings that are used in Standalone Mode are not applicable in Federated Mode.
- Rulepacks are specified through the controller's user interface. Rulepacks cannot be specified directly in a Federation Configuration Template.

Creating a Bootstrap Configuration

In order to join a federation, a host uses a bootstrap configuration to contact the controller. The controller responds with a set of rules and configuration files for the host.

A bootstrap configuration uses the same format as a standalone configuration file, but it used primarily to contact the controller and download the federation configuration.

The following illustrates a simple bootstrap configuration:

Example: Bootstrap configuration

```
<FortifyRuntime
xmlns="xmlns://www.fortifysoftware.com/schema/runtime/configuration"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <GlobalSettings>
    <Setting name="Controller">f360server.example.com</Setting>
  </GlobalSettings>
</FortifyRuntime>
```

A bootstrap configuration can include more than just the name of the controller. All settings available in Standalone Mode are also legal as part of a bootstrap configuration. Values specified in a bootstrap configuration become default values in the federation configuration. If the federation configuration explicitly specifies a setting, then that explicit value will be used, otherwise the value from the bootstrap configuration will be used.

Upon initial execution, Fortify Runtime will delay execution of the target program until it receives a configuration from the controller. Once a host receives a configuration from the controller, it will cache the configuration and use it until the controller sends a new configuration. This configuration cache survives across program restarts, so the host does not have to wait for the controller at startup after it has been configured for the first time.

The following table lists global settings that have special meaning in a bootstrap configuration. These settings are not used in Standalone Mode.

Global Settings for Bootstrap Configuration

Name	Default Value	Description
`AuthDirectory`	`${FortifyHome}/internal/auth`	The directory where the host stores controller

Global Settings for Bootstrap Configuration, continued

Name	Default Value	Description
		authentication credentials
`Controller`	No default. It must be specified in the bootstrap configuration file.	The DNS name or IP address of the federation controller. The presence of this option indicates that the host is to operate in Federated Mode.
`ControllerPort`	10234	The port used to communicate with the controller.
`MaxWaitForInitial Configuration`	30	The number of seconds the host will wait to receive an initial configuration from the controller before giving up.
`ProcessName`	None	A name for the program running in this process. Currently used to assign the process to a federation. If not specified, the controller will assign Fortify Runtime to the default federation for the host. If specified, it must be the name of a federation, and the controller will add Fortify Runtime to named federation.
`RemoteConfig Directory`	`${FortifyHome}/ internal/remote_ config`	The directory where the host stores configuration and rules files sent from the controller.
`StartUsingCached Configuration`	`true`	If true, the host will begin operation using a local cache of configuration information (provided the cache has been established in a previous run.) If false, the host will wait for the latest configuration from the controller before running the target program.

Creating a Federation Template Configuration

The base configuration used by Software Security Server to create a host configuration is called a Federation Template Configuration. Its global settings are listed in the next table. Configuration changes made through the user interface are layered on top of this template and the combined configuration is sent to all hosts in a federation.

A Federation Template Configuration is similar to a standalone configuration, but with some important differences:

- Hosts respect some additional global settings in Federated Mode. A small number of global settings that are used in Standalone Mode are not applicable in Federated Mode.
- Rulepacks are specified through the controller's user interface. Rulepacks cannot be specified directly in a Federation Template Configuration.
- Federated Mode allows an additional dispatch option for hosts: `controller`.
- Event handlers defined through the user interface must be positioned in the host's event handler chain.
- The controller uses a different event handler chain than the hosts. The controller's event handler chain is specified in the Federation Template Configuration.

These differences are described in more detail in the following sections.

Global Settings for a Federation Template Configuration

Name	Default Value	Description
`StatusInterval`	10	The number of seconds between status updates sent by the host.
`ConfigurationId`	(no default, must be specified by the controller)	The identifier for the federation assigned to the host by the controller. This value must be set by the controller. It is an error to specify it in a federation template configuration.
`MaxEventCacheSize`	1000000	The maximum number of events the host will cache on disk if communication with the controller is lost. If the cache grows larger than this value, events will be dropped. A value of zero allows the cache to grow without bound.
`ReconnectDelay`	0, 1, 2, 2, 10, 15, 30, 60	The number of seconds between reconnect attempts, if the connection to the controller is lost. Consecutive failures cause a progression through the array. The default value causes no delay after an initial failure, then a one-second delay, then a two-second delay, and so on.

Some settings that can be used in Standalone Mode are not used in Federated Mode. They are listed in the following table.

Standalone Mode (only) Configuration Setting

Name	Description
ConfigurationPollInterval	The host receives updates from the controller in real time, so no monitor thread exists in Federated Mode, and no polling is needed.

All global settings accept three optional attributes that have special meaning in a graphical user interface: `label`, `description` and `type`. These attributes are legal in a standalone configuration file too, but they are not interpreted by Fortify Runtime, so they are of no benefit in a standalone configuration.

The label attribute gives a display name for the global setting. The description attribute gives a longer explanation of the setting. The type attribute provides a hint to the user interface so that it can perform error checking on user input. By convention, the following types are recognized by Software Security Server: STRING, NUMBER, BOOLEAN, ENUM, and HIDDEN. The ENUM option should be followed by a colon and a comma separated list of the enumeration values. The HIDDEN option causes the setting to not appear in the Software Security Server interface.

The following example shows a global setting declaration that uses `label`, `description`, and `type`.

Example: Global setting declaration

```
<Setting description="The action taken if no event handler specifies
anything more specific"
         name="default_action" label="Default Action"
         type="ENUM:display,throw,ignore">display</Setting>
```

Specifying Rules

Rulepacks are specified through the controller's user interface. Rulepacks cannot be specified directly in a federation template configuration. Instead, use the `ControllerManagedRules` tag to specify where the controller should insert the Rulepacks specified through the user interface. The following example includes controller managed rules and then disables rules for persistent cross-site scripting.

Example: ControllerManagedRules tag

```
<Rules>
  <ControllerManagedRules/>
  <DisableRules>
    <MatchAttribute name="category">Cross-Site Scripting</MatchAttribute>
    <MatchAttribute name="subcategory">Persistent</MatchAttribute>
  </DisableRules>
</Rules>
```

An Additional Host Dispatch Option

In addition to the dispatch options available in Standalone Mode, in Federated Mode a host can send events to the controller using a `controller` dispatch. The event handler in the example below sends all events to the controller.

Example: Controller Dispatch option

```
<EventHandler propagate="true"
              description="Send events to the controller"
              label="Event logger">
  <Match/>  <!-- match all events -->
  <Handle>
    <Dispatch name="controller"/>
  </Handle>
 </EventHandler>
```

Including Event Handlers Defined in the User Interface

The event handler chain for federation hosts is defined in the federation template configuration just as it is in a standalone configuration, but additional event handlers can be defined through the user interface. All of the event handlers defined through the user interface are included in sequence in a host configuration. The tag `ControllerManagedEventHandlers` marks the spot in the event handler chain where they will appear. This tag is not allowed in a standalone configuration. The following event handler chain drops events carrying the "`useless`" attribute, then applies the event handlers from the user interface, then sends events to the controller.

Example: Applying event handler from the user interface

```
<EventHandlers>
  <EventHandler
    <Match>
      <MatchAttribute name="useless"/>
    </Match>
  </EventHandler>

  <!-- event handlers from interface are plugged in here -->
  <ControllerManagedEventHandlers/>

  <EventHandler>
```

```
    <Match/>
    <Handle>
      <Dispatch name="controller"/>
    </Handle>
  </EventHandler>
</EventHandlers>
```

The Controller Event Handler Chain

The controller has its own event handler chain specified at the bottom of the federation default template configuration with the tag `ControllerEventHandlers`. This event handler chain will be evaluated whenever the controller receives an event from a host. Event handlers are specified in much the same way as for hosts, with a few exceptions:

- Event handlers on the controller cannot specify actions. There is no direct means for the controller to change the state of the target program.
- Event handlers on the controller cannot use the dispatch options available in Standalone Mode. Instead, the controller's dispatch calls must be one of the values shown in the following table.

Dispatch Options

Name	Description
`persist`	Persists the event in the controller's database.
`alert`	Sends an alert for the event. (See HPE Security Fortify Software Security Server documentation for more information about alerts.)

As with the event handler chain for hosts, the controller event handler chain must specify where event handlers defined through the user interface will appear. The same tag used for the host event handler chain is also used for the controller event handler chain: `ControllerManagedEventHandlers`.

If the host event handler chain defines clusters, the same clusters should be defined in the controller's event handler chain so that consistent cluster information will be shared across hosts.

The example below illustrates a controller event handler chain.

Example:Controller event handler chain

```
<ControllerEventHandlers>

  <!-- create clusters first -->
  <EventHandler description="Create a cluster for credit card fraud.">
    <Match>
      <MatchAttribute name="kind">credit card transaction
```

```
failure</MatchAttribute>
      <Cluster id="ccfraud" n="5" window="300" linger="18000"
               picture="%userid%req_address" maxClusters="100"/>
    </Match>
  </EventHandler>

  <EventHandler description="persist events" propagate="true">
    <Handle>
      <Dispatch name="persist"/>
    </Handle>
  </EventHandler>

  <!-- event handlers created through the UI go here -->
  <ControllerManagedEventHandlers/>

  <Default>
    <!-- ignore -->
  </Default>
</ControllerEventHandlers>
```

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this computer, click the link above and an email window opens with the following information in the subject line:

Feedback on Java Edition Designer Guide (HPE Security Fortify Runtime 17.3)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to HPFortifyTechpubs@hpe.com.

We appreciate your feedback!