Micro Focus
Fortify Static Code Analyzer

Software Version: 23.1.0

Custom Rules Guide

Document Release Date: May 2023
Software Release Date: May 2023

Legal Notices

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Copyright Notice

Copyright 2003 - 2023 Open Text.

The only warranties for products and services of Open Text and its affiliates and licensors (“Open Text”) are as may be set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Open Text shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Trademark Notices

“OpenText” and other Open Text trademarks and service marks are the property of Open Text or its affiliates. All other trademarks or service marks are the property of their respective owners.

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

This document was produced on April 13, 2023. To check for recent updates or to verify that you are using the most recent edition of a document, go to:

https://www.microfocus.com/support/documentation

Contents

Preface 8
- Contacting Micro Focus Fortify Customer Support 8
- For More Information 8
- About the Documentation Set 8
- Fortify Product Feature Videos 8

Change Log 9

Chapter 1: Introduction 10
- Intended Audience 10
- Document Structure 10
- Additional Custom Rules Documentation and Sample Application 11
- Related Documents 11
  - All Products 12
  - Fortify Static Code Analyzer 13

Chapter 2: Custom Rules Overview 14
- Fortify Secure Coding Rulepacks 14
- Custom Rules 14
- Custom Rules and User Roles 15
  - Individual Auditor 15
  - Central Security Team 15
  - Development Team 16
- Rulepacks and Common Rule XML Elements 16
  - RulePack Element 17
  - Rules Element 18
  - Common Rule Elements 20
    - FunctionIdentifier Element 21
    - Parameters Element 24
    - Modifiers Element 25
    - Conditional Elements 26

Custom Descriptions 28

Adding Custom Descriptions to Fortify Rules 28

Identify Rules to Modify 30

Adding Fortify Descriptions to Custom Rules 30

Chapter 3: Structural Analyzer Rules 32

Structural Analyzer and Custom Rules 32

Structural Tree 32

Structural Tree Query Language 33

Structural Tree Examples 33

XML Representation of Structural Analyzer Rules 38

Custom Structural Rule Scenarios 38

Leftover Debug 39

Dangerous Function Calls 40

Overly Broad Catch Blocks 42

Password in Comments 45

Poor Logging Practice 46

Empty Catch Block 47

Chapter 4: Dataflow Analyzer Rules 49

Dataflow Analyzer and Custom Rules 49

Dataflow Analyzer and Custom Rules Concepts 50

Taint Source 51

Taint Write 51

Taint Entrypoint 51

Taint Sink 52

Taint Passthrough / Transfer 52

Taint Cleanse 52

Taint Flags 53

Taint Flag Types 53

Taint Flag Behavior 54

Taint Path 54

Validation Constructs 54

Types of Dataflow Analyzer Rules 55

XML Representation of Dataflow Analyzer Rules 55

Source Rules 57

Sink Rules 60

Passthrough Rules 62
Entrypoint Rules 64
Cleanse Rules 65
Custom Dataflow Analyzer Rule Scenarios 67
SQL Injection and Access Control 67
Persistent Cross-Site Scripting 72
Path Manipulation 78
Command Injection 81
Validation Construct Examples 85

Chapter 5: Control Flow Analyzer Rules 91
Control Flow Analyzer and Custom Rules 91
Control Flow Analyzer and Custom Rule Concepts 93
Rule Pattern 93
Rule Variable 93
Rule Binding 93
XML Representation of Control Flow Analyzer Rules 94
Custom Control Flow Rule Scenarios 96
Resource Leak 97
Null Pointer Check 103

Chapter 6: Content and Configuration Analyzer Rules 107
Content Analyzer and Custom Rules 107
XML Representation of Content Analyzer Rules 107
Configuration Analyzer and Custom Rules 108
XML Representation of Configuration Analyzer Rules 108
Configuration Rules 108
Regular Expression Rules 111
Custom Configuration Rule Scenarios 112
Property File 112
Tomcat File 114
Authentication Tokens in Files 115

Chapter 7: Manipulation Rules 117
Suppression Rules 117

XML Representation of Suppression Rules 119
Alias Rules 119
XML Representation of Alias Rules 120
Result Filter Rules 120
XML Representation of Result Filter Rules 121

Chapter 8: Custom Vulnerability Category Mapping 124
Mapping Fortify Categories to Alternative External Categories 124
External Metadata XML Structure 125
ExternalMetadataPack Element 125
PackInfo Element 127
ExternalList Element 128
ExternalListExtension Element 129
ExternalCategoryDefinition Element 130
Mapping Element 131
XML Skeleton 131
Example Mappings 132

Appendix A: Taint Flag Reference 134
General Taint Flags 134
Specific Taint Flags 136
Neutral Taint Flags 139

Appendix B: Structural Rules Language Reference 143
Structural Syntax and Grammar 143
Types 144
Properties 145
Reference Resolution 146
Null Resolutions 146
Relations 147
Results Reporting 148
Call Graph Reachability 149

Appendix C: Control Flow Rule Reference 151

Syntax and Grammar 151

Control Flow Rules 152

Control Flow Rule Identifiers 152

Control Flow Rule Format 152

Declarations 153

Transitions 153

Function Calls 156

Send Documentation Feedback 157

Preface

Contacting Micro Focus Fortify Customer Support

Visit the Support website to:

- Manage licenses and entitlements
- Create and manage technical assistance requests
- Browse documentation and knowledge articles
- Download software
- Explore the Community

https://www.microfocus.com/support

For More Information

For more information about Fortify software products:

https://www.microfocus.com/cyberres/application-security

About the Documentation Set

The Fortify Software documentation set contains installation, user, and deployment guides for all 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 Micro Focus Product Documentation website:

https://www.microfocus.com/support/documentation

To be notified of documentation updates between releases, subscribe to Fortify Product Announcements on the Micro Focus Community:

https://community.microfocus.com/cyberres/fortify/w/fortify-product-announcements

Fortify Product Feature Videos

You can find videos that highlight Fortify products and features on the Fortify Unplugged YouTube channel:

https://www.youtube.com/c/FortifyUnplugged

Change Log

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

Software Release / Document Version	**Change**
23.1.0	Updated: Release date and version
22.2.0	Updated: • Added new modifier `suspend` for use with Kotlin only (see "Modifiers Element" on page 25)
22.1.0	Updated: • Clarified how the language attribute can apply to multiple programming languages (see "Rules Element" on page 18) • Added a new rule type: RegexRule (see "Configuration Analyzer and Custom Rules" on page 108)
21.2.0	Updated: Release date and version

Chapter 1: Introduction

Fortify Static Code Analyzer provides a set of analyzers that detect potential security vulnerabilities in source code. It is important, when you analyze a project, that the Fortify Static Code Analyzer translation phase completes without errors and that all relevant source code is included to ensure that the necessary artifacts are part of the scanned model. After the source code is translated, the Fortify Static Code Analyzer analyzers can use both Fortify Secure Coding Rulepacks and customer-specific security rules (custom rules) to identify vulnerabilities.

This document provides the information that you need to create custom rules for Fortify Static Code Analyzer. This includes both conceptual content that focuses on customizing topics and several examples that apply rule-writing concepts to real-world problems.

Intended Audience

This document is intended for people who are experienced with both security and programming. Some content in this guide might be difficult to understand without programming experience.

Document Structure

The following chapters describe how Fortify Static Code Analyzer works with specific analyzers to discover vulnerabilities in code and how to write custom rules to influence the results produced:

- "Custom Rules Overview" on page 14—Describes Fortify Secure Coding Rulepacks, custom rules, and introduces the XML representation for rules
- "Structural Analyzer Rules" on page 32—Describes how to write custom rules to detect issues by identifying certain patterns of code
- "Dataflow Analyzer Rules" on page 49—Describes how to write custom rules to detect security issues that involve tainted data (user-controlled input) that is put to potentially dangerous use
- "Control Flow Analyzer Rules" on page 91—Describes how to write custom rules to detect issues in programs that have insecure sequences of operations
- "Content and Configuration Analyzer Rules" on page 107—Describes how to write custom rules to detect issues in HTML content and application configuration files
- Custom Rule Scenarios—The specific analyzer chapters include scenarios for the sample application called Riches Wealth Online (RWO). This application enables users to perform the following online banking operations: transfer money, view account statements, and receive messages. The RWO application demonstrates the diverse range of application security vulnerabilities that are typically encountered in real-world applications that provide functionality similar to RWO. The application is built with JavaScript, Struts 2, Hibernate 2, and Java Enterprise Edition.

Each scenario highlights specific vulnerabilities in RWO and demonstrates how to identify them using custom rules. Each scenario shows how an attacker can exploit vulnerabilities in RWO source code. The scenario, where applicable, highlights how Fortify Static Code Analyzer and the Secure Coding Rulepacks detect the vulnerability. The scenario then explains the type of custom rules necessary to detect the vulnerability and provides examples.

You can reproduce the results by analyzing RWO with either Secure Coding Rulepacks or by using the provided custom rules. To use the provided custom rules, you must first disable Secure Coding Rulepacks.

- The "Custom Vulnerability Category Mapping" on page 124 chapter describes how to create mappings from alternative taxonomies and standards to the Fortify Taxonomy.

The following appendices provide reference information you need to write custom rules:

- "Taint Flag Reference" on page 134—Describes the taint flags included with the Fortify Secure Coding Rulepacks
- "Structural Rules Language Reference" on page 143—Provides syntax and grammar for structural rules
- "Control Flow Rule Reference" on page 151—Provides syntax and grammar for control flow rules

Additional Custom Rules Documentation and Sample Application

The following additional information is included in the ZIP file from which you extracted this document:

- *Fortify Static Code Analyzer Rules Schema*—This HTML content provides the Fortify XML schema, including valid attributes and elements, child and parent relationships between elements, whether an element is empty or can include text, element data types, as well as element and attribute default and fixed values.
- *Fortify Structural Type and Properties Reference*—This HTML content provides type and properties reference for structural rules. Use this content when creating custom structural rules.
- Riches Wealth Online (RWO)—Sample application (`riches.zip`) that contains the vulnerabilities described in the custom rule scenarios in this guide and scenario custom rules.

Related Documents

This topic describes documents that provide information about Micro Focus Fortify software products.

> **Note:** You can find the Fortify Product Documentation at https://www.microfocus.com/support/documentation. Most guides are available in both PDF and HTML formats. Product help is available within the Fortify LIM product.

All Products

The following documents provide general information for all products. Unless otherwise noted, these documents are available on the Micro Focus Product Documentation website.

Document / File Name	Description
About Fortify Product Software Documentation About_Fortify_Docs_<*version*>.pdf	This paper provides information about how to access Fortify product documentation. **Note:** This document is included only with the product download.
Fortify License and Infrastructure Manager Installation and Usage Guide LIM_Guide_<*version*>.pdf	This document describes how to install, configure, and use the Fortify License and Infrastructure Manager (LIM), which is available for installation on a local Windows server and as a container image on the Docker platform.
Fortify Software System Requirements Fortify_Sys_Reqs_<*version*>.pdf	This document provides the details about the environments and products supported for this version of Fortify Software.
Fortify Software Release Notes FortifySW_RN_<*version*>.pdf	This document provides an overview of the changes made to Fortify Software for this release and important information not included elsewhere in the product documentation.
What's New in Fortify Software <version> Fortify_Whats_New_<*version*>.pdf	This document describes the new features in Fortify Software products.

Fortify Static Code Analyzer

The following documents provide information about Fortify Static Code Analyzer. Unless otherwise noted, these documents are available on the Micro Focus Product Documentation website at https://www.microfocus.com/documentation/fortify-static-code.

Document / File Name	Description
Fortify Static Code Analyzer User Guide SCA_Guide_<*version*>.pdf	This document describes how to install and use Fortify Static Code Analyzer to scan code on many of the major programming platforms. It is intended for people responsible for security audits and secure coding.
Fortify Static Code Analyzer Applications and Tools Guide SCA_Apps_Tools_<*version*>.pdf	This document describes how to install Fortify Static Code Analyzer applications and tools. It provides an overview of the applications and command-line tools that enable you to scan your code with Fortify Static Code Analyzer, review analysis results, work with analysis results files, and more.
Fortify Static Code Analyzer Custom Rules Guide SCA_Cust_Rules_Guide_<*version*>.zip	This document provides the information that you need to create custom rules for Fortify Static Code Analyzer. This guide includes examples that apply rule-writing concepts to real-world security issues. **Note:** This document is included only with the product download.

Chapter 2: Custom Rules Overview

This section contains the following topics:

Fortify Secure Coding Rulepacks 14
Custom Rules 14
Custom Rules and User Roles 15
Rulepacks and Common Rule XML Elements 16
Custom Descriptions 28

Fortify Secure Coding Rulepacks

Fortify Static Code Analyzer uses a knowledge base of rules to model important attributes of the program under analysis. These rules provide meaning to relevant data values and enforce secure coding standards applicable to the codebase. The Secure Coding Rulepacks describe general secure coding idioms for popular languages and public APIs, out of the box. You can write custom rules for ABAP, ASP.NET, C, C++, Java, .NET, PL/SQL, T-SQL, and VB.NET.

Although Fortify provides a wide range of rules, it is possible that your projects leverage unsupported third-party APIs, include organization-specific libraries, or fall under the purview of proprietary secure-coding guidelines. In this case, Fortify provides the ability to create custom rules that suit your needs.

Custom rules can improve the completeness and accuracy of the Fortify Static Code Analyzer analysis. This is accomplished by modeling the behavior of the security-relevant libraries, describing proprietary business and input validation, and enforcing organization and industry-specific coding standards.

Custom Rules

You write custom rules to extend the functionality of Fortify Static Code Analyzer and the Secure Coding Rulepacks. For example, you might need to enforce proprietary security guidelines or analyze a project that uses third-party libraries or other pre-compiled binaries that are not already covered by the Secure Coding Rulepacks.

If a project uses resources for which source code is not available at analysis time, analysis of the project succeeds, but might be incomplete until you write the custom rules that provide Fortify Static Code Analyzer with security knowledge about these resources.

To write effective custom rules, it is important to become familiar with known security vulnerability categories and the code constructs with which they are often related. Developing an understanding of the types of functions that often appear in particular types of vulnerabilities facilitates the process of

targeting security-relevant functions for custom rule writing. Because the task of determining the security relevance of a function can be challenging, time spent learning about the relationships between types of functions and vulnerability categories can prove useful.

You must examine the individual behavior of each security-relevant function, either by reviewing source code or with the help of API documentation, to determine the correct type of rule to represent the specific behavior and vulnerability category associated with each of the functions.

You can then develop small test cases that exemplify the undesirable behavior you want your rules to identify. Conversely, test cases designed to reflect correct behavior that should not be flagged also help you to eliminate false positives from the rules you create. After you are satisfied that your rules perform correctly in this controlled environment, the next step is to use them to perform an analysis on a broad range of projects to verify that they behave with the expected level of fidelity.

To simplify the creation of custom rules, you can use the Fortify Custom Rules Editor, which you can install as a component with the Fortify Static Code Analyzer Applications and Tools installer. For installation instructions, see the *Fortify Static Code Analyzer Applications and Tools Guide*.

Custom Rules and User Roles

User roles also play an important part in creating and using custom rules. For example, an individual auditor might require different custom rules than a security team. The rest of this section describes common user roles and identifies custom rules specific to those roles.

Individual Auditor

An individual auditor performs a single security review of a project for a specific organization. A security researcher who looks for bugs in a piece of public software also fits into this role. The goal of this user is to identify specific vulnerabilities based on a narrow set of security criteria.

A person in this role develops and uses custom rules along a narrow set of parameters and does not strive for breadth of coverage. An example of this is to address the strategic shortcoming of the built-in knowledge base of rules.

This includes identifying specific classes of bugs or modeling the behavior of APIs that are likely to lead to vulnerabilities targeted in the current audit.

Development of a large body of custom rules is not a requirement for this user. Any effort that this individual puts into customization should be weighed against the benefit that the customization provides.

Central Security Team

A central security team is typically responsible for developing custom rules that identify a broad set of vulnerabilities across multiple code bases within an organization. The central security team provides value by developing large databases of rules that improve the static analysis results during ongoing audits.

If the central security team is responsible for auditing the results produced by the custom rules, then you might include rules that provide an auditor with a checklist of properties to verify in the audit.

However, if the development team responsible for each project reviews the static analysis tool results directly, then the tolerance for issues that do not correspond directly to security vulnerabilities or other programming bugs would likely be much lower.

In either case, the goal is to produce a large knowledge base of custom rules relevant to projects under analysis, because the rule writers have incentive to improve analysis results during ongoing audits.

Development Team

If a development team is responsible for both implementing custom rules and auditing the results of the static analysis tool, the extent to which you want to customize varies based on the security experience of the development team. If the development team is only tangentially involved in security, their use of custom rules might focus on a narrow field of relevant bugs.

Rulepacks and Common Rule XML Elements

Fortify Static Code Analyzer comprises multiple analyzers that perform different types of analysis and find different types of problems in code. Each analyzer supports one or more distinct rule types. Secure Coding Rulepacks are represented in XML. A Rulepack contains one or more rules of an arbitrary type.

This document covers these rule types (listed in alphabetical order):

- AliasRule
- CharacterizationRule (for Dataflow Analyzer)
- ConfigurationRule
- ContentRule
- ControlflowRule
- CustomDescriptionRule
- DataflowCleanseRule
- DataflowEntrypointRule
- DataflowPassthroughRule
- DataflowSinkRule
- DataflowSourceRule
- RegexRule
- ResultFilterRule
- StructuralRule
- SuppressionRule

RulePack Element

The root element of a Rulepack is `<RulePack>`. The `<RulePack>` element includes header information that describes the Rulepack.

The following example shows a `<RulePack>` element:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RulePack>
  <RulePackID>06A6CC97-8C3F-4E73-9093-3E74C64A2AAF</RulePackID>
  <Name><![CDATA[
   Sample Custom Fortify Rulepack
  ]]></Name>
  <Version>0000.0.0.0000</Version>
  <Language>java</Language>
  <Description><![CDATA[
   Custom Rules for Java
  ]]></Description>
  <Rules version="23.1">...</Rules>
</RulePack>
```

The following table describes the `<RulePack>` child elements.

Element	Description
RulePackID	A unique identifier for the Rulepack. By convention, Fortify uses a globally unique identifier (GUID) generator to define Rulepack and rule identifiers to ensure that the identifiers are unique.
Name	A name for the Rulepack.
SKU	A global unique identifier.
Language	(Optional) The programming language applicable to all rules in the Rulepack. Fortify Static Code Analyzer only loads the Rulepack when processing source files of the specified language. If the `<Language>` element is not included, Fortify Static Code Analyzer always loads the Rulepack.
Version	(Optional) An arbitrary numeric version used to relate multiple versions of the same Rulepack (Rulepacks with the same Rulepack identifier).
Description	A description of the Rulepack.

Element	Description
Locale	(Optional) The locale for the Rulepack. The valid values are `en`, `es`, `ja`, `ko`, `pt_BR`, `zh_CN`, and `zh_TW`.
Rules	Contains one `<RuleDefinitions>` element. See "Rules Element" below. This element contains the following attribute: • `version`—The version of the rules schema for which the custom rules are written.

Rules Element

The `<Rules>` element contains all the rule definitions.

```
<Rules version="23.1">
  <RuleDefinitions>
    <!--... rules go here ...-->
    <xyzRule>...</xyzRule>
      ...
    <xyzRule>...</xyzRule>
  </RuleDefinitions>
</Rules>
```

The following table describes the `<Rules>` child elements.

Element	Description
RuleDefinitions	Contains one or more top-level rules.

<table>
<tr>
<td>xyzRule</td>
<td>
Each rule has a unique rule element: <*xyz*Rule> where *xyz* is a valid rule type. Examples of valid rule elements are <StructuralRule>, <DataflowRule>, <ControlflowRule>, and so on. This element has the following required attributes:

• formatVersion—The version of Fortify Static Code Analyzer with which the rule is compatible. Specify the installed Fortify Static Code Analyzer version number to take advantage of all the current functionality. To determine the Fortify Static Code Analyzer version, type sourceanalyzer -v on the command line. The version number format is *<major>*.*<minor>*.*<patch>*.*<buildnumber>* (for example, 23.1.0.0140). Only the major and minor portions of the version are required.

• language—The programming language to which the rule applies. The valid values for language are abap, cpp, dotnet, java, and sql.

The language attribute can apply to more than one programming language. The following table describes how the language attribute value is applied for each programming language.
<table>
<tr><th>Language Attribute</th><th>Rule applies to...</th></tr>
<tr><td>abap</td><td>ABAP</td></tr>
<tr><td>cpp</td><td>C, C++, Objective-C, and Objective-C++</td></tr>
<tr><td>dotnet</td><td>.NET (C# and VB.NET)</td></tr>
<tr><td>java</td><td>Java, JSP, kotlin, Scala, Android, and Xamarin

Note: For Xamarin projects, the Java APIs used in C# are converted to the associated Java API.</td></tr>
<tr><td>sql</td><td>SQL, T-SQL, and PL/SQL</td></tr>
</table>
</td>
</tr>
</table>

Common Rule Elements

The top-level rule element contains different elements depending on the rule type. Fortify Static Code Analyzer rules share a few common elements. All rules have a `<RuleID>` element.

```
<xyzRule formatVersion="23.1">
  <RuleID>...</RuleID>
  <MetaInfo>
   <Group name="Accuracy">4.0</Group>
   <Group name="Impact">5.0</Group>
   <Group name="Probability">4.0</Group>
  </MetaInfo>
  <Notes>...</Notes>
  ...
</xyzRule>
```

The following table describes the common child elements of the top-level rule element.

Element	Description
RuleID	A required unique identifier for the rule, which can be an arbitrary string of characters. As with Rulepack IDs, by convention Fortify uses a globally unique identifier (GUID) generator for unique rule identifiers.
MetaInfo	(Optional) Provides additional information about a rule for prioritizing analysis results. The child element is `<Group>`. Use the `<Group>` element's `name` attribute to specify accuracy, impact, and probability for a vulnerability. The valid values are `0.1` to `5.0`. You can specify the following information for the `name` attribute: • `Accuracy`—Possibility that the rule correctly identifies a vulnerability • `Impact`—Negative outcome resulting from a vulnerability • `Probability`—Likelihood that the vulnerability is discovered and acted upon
Notes	(Optional) Your own internal comments about the rule.

The following top-level rule elements are common only to rules that directly cause the respective analyzer to report an issue:

```
<xyzRule formatVersion="23.1">
  <RuleID>C9ECD6EC-DAA1-41BE-9715-033F74CE664F</RuleID>
  <VulnCategory>Poor Error Handling</VulnCategory>
```

```
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description>...</Description>
</xyzRule>
```

The following table describes the rule elements common to vulnerability-producing rules.

Element	Description
VulnKingdom	(Optional) Vulnerability kingdom assigned to issues the rule uncovers.
VulnCategory	Vulnerability category assigned to issues the rule uncovers.
VulnSubcategory	(Optional) Vulnerability subcategory assigned to issues the rule uncovers.
Description	Description of the vulnerability the rule identifies. The `<Description>` element can contain any of the following elements: `<Abstract>`, `<Explanation>`, `<Recommendations>`, `<References>`, and `<Tips>`. This element has the following attribute: • `ref`—(Optional) Specifies a description identifier to use a Fortify description for your custom rule. You can find description identifiers at https://vulncat.fortify.com. You can find the description identifier at the bottom of each vulnerability description. For more information, see "Adding Custom Descriptions to Fortify Rules" on page 28.
DefaultSeverity	This element is no longer used but is required for backward compatibility. Specify a value of `2.0` for this element.
FunctionIdentifier	(Optional) Specifies a rule that refers to a function or method call. See "FunctionIdentifier Element" below. This element has the following attribute: • `id`—(Optional) Specifies a name for the function identifier so you can refer to it elsewhere.

FunctionIdentifier Element

Rules that refer to function or method calls (as opposed to configuration files, property files, HTML, and other content) can use the `<FunctionIdentifier>` element.

> **Note:** Any rule that contains the `<FunctionIdentifier>` element must have the `language` attribute specified for the rule.

```
<xyzRule formatVersion="23.1" language="java">
  <RuleID>...</RuleID>
```

```
  <VulnCategory>...</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description>...</Description>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.lang</Value>
    </NamespaceName>
    <ClassName>
      <Value>String</Value>
    </ClassName>
    <FunctionName>
      <Value>trim</Value>
    </FunctionName>
    <ApplyTo implements="true" overrides="true" extends="true"/>
    <Parameters>...</Parameters>
  </FunctionIdentifier>
</xyzRule>
```

The following table describes the `<FunctionIdentifier>` child elements. For object-oriented languages, always specify `<ClassName>` and `<NamespaceName>`.

Element	Description
FunctionName	The name of the method or function that the rule matches.
ClassName	(Optional) The name of the class that the rule matches. If you do not specify a `<ClassName>`, the rule only matches functions that are not inside a class. To match a function in any class, specify the class name using the `<Pattern>.*</Pattern>` child element. To match a nested class, use the dot notation (for example, `<Value>OuterClass.NestedClass</Value>`). **Note:** For .NET languages, the convention for the `<ClassName>` of a generic class is to append the class name with an at sign (@) and the number of type parameters. Example: for `System.Func<T, TResult>`, the `<ClassName>` is `<Value>Func@2</Value>`.
NamespaceName	(Optional) The name of the package or namespace that the rule matches. If you do not specify a `<NamespaceName>`, the rule only matches functions that are not inside a namespace. **Note:** In Java, the namespace name is equivalent to the Java package name.

Element	**Description**
ApplyTo	(Optional) Controls how the rule matches against classes that extend the specified class or implement the specified interface. This element contains the following attributes: • `implements`—(Optional) True indicates that the rule should match methods that implement the interface methods that the rule specifies. • `overrides`—(Optional) True indicates that the rule should match methods defined in sub-classes that override the method that the rule specifies. • `extends`—(Optional) True indicates that the rule should match methods in classes that extend the class that the rule specifies. The default value of all three `<ApplyTo>` element attributes is `false`.
ReturnType	(Optional) Limits the functions matched to the functions with the specified return type. A return type is a language-specific primitive type or a defined type (for example, `java.lang.String` or `std::string`). You can optionally modify types with `*`, `[]`, or & representing pointer, array, or C++ references, respectively. To match a nested type, use the dot notation (for example, `OuterType.NestedType`).
MatchExpression	(Optional) Expression to match a function. You cannot combine this element with `<FunctionName>`, `<ClassName>`, `<NamespaceName>`, `<ApplyTo>`, `<Modifiers>`, `<Parameters>`, or `<ReturnType>`.
Parameters	(Optional) Limits the methods that the rule matches based on the type signature of the function. See the "Parameters Element" on the next page.
Modifiers	(Optional) Limits the methods that the rule matches to those declared with the specified modifiers. See the "Modifiers Element" on page 25.
Except	(Optional) Specifies nested `<FunctionIdentifier>` elements that should not match.

The `<FunctionName>`, `<ClassName>`, and `<NamespaceName>` elements are expressed using one of the child elements described in the following table.

Element	Description
Value	Fortify Static Code Analyzer interprets the name as a standard string. For example: `<Value>java.util</Value>`
Pattern	Fortify Static Code Analyzer interprets the name as a valid Java regular expression. Make sure that you escape regular expression symbols. For example: `<Pattern>java\.util</Pattern>`

Parameters Element

The `<Parameters>` element limits the methods that the rule matches. The following example shows a `<Parameters>` element with both optional child elements:

```
<Parameters>
  <ParamType>java.lang.String</ParamType>
  <WildCard min="0" max="2"/>
</Parameters>
```

The following table describes the `<Parameters>` child elements.

Elements	Description
ParamType	Specifies a single parameter of a language-specific primitive type or a defined type (for example, `java.lang.String` or `std::string`). You can optionally modify types with `*`, `[]`, or `&` representing pointer, array, or C++ references, respectively. To match a nested type, use the dot notation (for example, `OuterType.NestedType`). **Note:** For .NET builtin types, you can use the following types as the `<ParamType>` element parameter. Specify all other custom types by their fully qualified namespace. **.NET Type** — **ParamType Parameter** System.Boolean — boolean

<table>
<tr><th>Elements</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>.NET Type</th><th>ParamType Parameter</th></tr>
<tr><td>System.Byte</td><td>byte</td></tr>
<tr><td>System.SByte</td><td>short</td></tr>
<tr><td>System.Char</td><td>char</td></tr>
<tr><td>System.Decimal
System.Double</td><td>double</td></tr>
<tr><td>System.Single</td><td>float</td></tr>
<tr><td>System.Int32
System.UInt32</td><td>int</td></tr>
<tr><td>System.Int64
System.UInt64</td><td>long</td></tr>
<tr><td>System.Object</td><td>System.Object</td></tr>
<tr><td>System.Int16
System.UInt16</td><td>short</td></tr>
<tr><td>System.String</td><td>System.String</td></tr>
</table>
</td></tr>
<tr><td>WildCard</td><td>(Optional) Represents a variable number of arbitrarily-typed parameters at the end of the parameter list for the method. This element can contain the following attributes:
• `min`—Specifies the fewest number of wildcard parameters the rule allows
• `max`—Specifies the maximum number of wildcard parameters the rule allows</td></tr>
</table>

Modifiers Element

The `<Modifiers>` element restricts the methods that the rule matches to those declared with the specified modifiers.

```
<Modifiers>
  <Modifier>static</Modifier>
</Modifiers>
```

The following table describes the `<Modifiers>` child element.

Element	Description
Modifier	Specifies the modifier type. Fortify supports the following modifiers: • `native` • `final` • `private` • `protected` • `public` • `static` • `suspend` (Kotlin only)

Conditional Elements

Many rule types allow matching to be further restricted using a conditional expression with the `<Conditional>` element. Function identifiers specify the functions or methods to which the rule pertains. Conditional expressions restrict the calls to those functions that the rule matches. You can write conditional expressions to examine constant values (boolean (true/false), integer, string (case-insensitive), and null) used in method calls and the types of method arguments (as distinct from the declared formal parameter types of the method). For dataflow sinks, conditional expressions can also examine taint flags.

The following example shows various conditional elements:

```
<Conditional>
  <And>
    <Not>
      <TaintFlagSet taintFlag="XSS"/>
    </Not>
    <And>
      <ConstantEq argument="0" value="strong"/>
      <ConstantGt argument="1" value="1023"/>
      <ConstantLt argument="2" value="2048"/>
    </And>
    <IsType argument="0">
      <NamespaceName>
        <Value>javax.servlet</Value>
      </NamespaceName>
      <ClassName>
        <Pattern>(Http)?ServletRequest</Pattern>
      </ClassName>
```

```
    </IsType>
    <Or>
      <NameEq argument="3" name="xyz"/>
    </Or>
  </And>
</Conditional>
```

The following table describes the <Conditional> child elements.

Element	Description
And Or Not	Boolean logic operators that apply the corresponding logical operation to the nodes they contain.
IsConstant	True if the zero-indexed `argument` attribute is a compile-time numeric or string constant (a literal or a variable/field that is assigned exactly once).
IsType	True if the zero-indexed `argument` attribute matches the `<NamespaceName>` and `<ClassName>` elements specified inside the `<IsType>` element. To match a nested type, use the dot notation (for example, `OuterType.NestedType`).
ConstantEq	True if the zero-indexed `argument` attribute is a compile-time numeric or string constant that matches the value specified by the `value` attribute. To match a nested type, use the dot notation (for example, `OuterType.NestedType`).
ConstantGt	True if the zero-indexed `argument` attribute is a compile-time numeric constant that is greater than the value specified by the `value` attribute.
ConstantLt	True if the zero-indexed `argument` is a compile-time numeric constant that is less than the value specified by the `value` attribute.
ConstantMatches	True if the zero-indexed `argument` attribute contains a substring that matches the regular expression specified in the `<Pattern>` child element.
NameEq	True if the zero-indexed `argument` attribute name argument equals the value specified for the `name` attribute.
NameMatches	True if the zero-indexed `argument` attribute name argument contains a substring that matches the string specified in the `<Pattern>` child element.

Element	Description
TaintFlagSet	True for taint paths that include the taint flag specified by the `taintFlag` attribute. **Note:** This element is only valid for dataflow sink rules.

Custom Descriptions

Some organizations want to either add custom descriptions to Fortify rules or add Fortify descriptions to custom rules. Custom descriptions enable you to add organization-specific content to issues the Fortify Secure Coding Rulepacks produce. Custom description content can include organization-specific secure coding guidelines, best practices, references to internal documentation, and so on. Adding Fortify descriptions to custom rules enables you to leverage descriptions Fortify creates in custom rules that identify categories of vulnerabilities that Secure Coding Rulepacks already reported.

- "Adding Custom Descriptions to Fortify Rules" below
- "Adding Fortify Descriptions to Custom Rules" on page 30

Adding Custom Descriptions to Fortify Rules

You can add custom descriptions with the `<CustomDescriptionRule>` element. Each custom description rule defines new description content and specifies a set of Fortify rules that determine how it is applied. By default, the Fortify Static Code Analyzer applications display the custom descriptions before the Fortify descriptions.

The following custom description rule example adds a custom `<Abstract>` and `<Explanation>` for `SQL Injection and Access Control: Database` issues:

```
<CustomDescriptionRule formatVersion="23.1">
  <RuleID>D40B319C-F9D6-424F-9D62-BB1FA3B3C644</RuleID>
  <RuleMatch>
    <Category>
      <Value>SQL Injection</Value>
    </Category>
  </RuleMatch>
  <RuleMatch>
    <Category>
      <Value>Access Control</Value>
    </Category>
    <Subcategory>
      <Value>Database</Value>
    </Subcategory>
  </RuleMatch>
  <Description>
    <Abstract>[custom abstract text]</Abstract>
    <Explanation>[custom explanation text]</Explanation>
  </Description>
  <Header>[string to replace Custom]</Header>
</CustomDescriptionRule>
```

To add custom descriptions to Fortify rules, do the following:

1. Define the custom description content—Use the `<Description>` and `<Header>` elements of the custom description rule to define the custom description content.
2. "Identify Rules to Modify" on the next page—Use the `<RuleMatch>` element to identify the rules to which Fortify Static Code Analyzer adds the custom description content.

The following table describes the `<CustomDescriptionRule>` child elements.

Element	**Description**
RuleID	A required unique identifier for the rule. See "Common Rule Elements" on page 20.
RuleMatch	Specifies the criteria to identify which rules Fortify Static Code Analyzer adds the custom description content.

Description	The common `<Description>` element described in "Common Rule Elements" on page 20. The custom description can specify all or a subset of the `<Description>` child elements.
Header	(Optional) Specifies text to replace the word "Custom" when Fortify Static Code Analyzer applications display the rule descriptions.

Identify Rules to Modify

A custom description can contain several rule matches. Each rule match specifies rules based on any combination of category, subcategory, rule identifier, and description identifier. Fortify Static Code Analyzer applies a custom description to issues a rule produces only if the rule matches all criteria specified in the rule match.

The following table describes the `<RuleMatch>` child elements.

Element	Description
Category	Vulnerability category
Subcategory	Vulnerability subcategory
RuleID	Identifier of the rule to match
DescriptionID	Identifier for the description you want to use (for example, `desc.dataflow.java.sql_injection`)

For example, a `<RuleMatch>` element that specifies `<Category><Value>Buffer Overflow</Value></Category>` and `<Subcategory><Value>Obsolete</Value></Subcategory>` matches only Buffer Overflow:Obsolete issues. The custom description content is not applied to issues in other Buffer Overflow sub-categories, such as Buffer Overflow: Off-by-One.

A rule need only satisfy one or more rule matches for a custom description rule. For example, a custom description rule with a rule match for `<Category><Value>Buffer Overflow</Value></Category>` and another distinct rule match for `<Subcategory><Value>Obsolete</Value></Subcategory>`, matches any issues in the Buffer Overflow category or the Obsolete subcategory.

Adding Fortify Descriptions to Custom Rules

You can use Fortify descriptions to describe the issues that custom rules find. To use a Fortify description in a custom rule, you must first determine the identifier for the description you want to use. Description identifiers are available on https://vulncat.fortify.com. After you locate the identifier for the description you want to use, set the `ref` attribute of the custom rule to the identifier of the Fortify description.

Note: To use this feature, make sure that the description IDs are unique across all Rulepacks.

For example, the following rule produces SQL injection results with the same description as SQL injection results from Fortify rules for Java:

```
<DataflowSinkRule formatVersion="23.1" language="java">
  ...
  <Description ref="desc.dataflow.java.sql_injection"/>
  ...
</DataflowSinkRule>
```

Chapter 3: Structural Analyzer Rules

This section contains the following topics:

Structural Analyzer and Custom Rules 32

XML Representation of Structural Analyzer Rules 38

Custom Structural Rule Scenarios 38

Structural Analyzer and Custom Rules

The Structural Analyzer matches arbitrary program constructs in source code. Unlike other Fortify Static Code Analyzer analyzers, it is not designed to find problems that arise from flow of execution or data. Instead, it detects issues by identifying certain patterns of code.

Structural Tree

The Structural Analyzer operates on a model of the program source code called the *structural tree*. The structural tree is made up of a set of nodes that represent program constructs such as classes, functions, fields, code blocks, statements, and expressions.

Nodes in the structural tree can have a single parent and many children. For example, a node that represents a field is the child of a node that represents the class in which that field is declared. Likewise, a node that represents an expression is the child of a node that represents the statement in which that expression appears.

Each node in the structural tree also has a set of properties. Some properties encode simple values, such as the name of a function or the type of a variable. Properties can also express relationships between nodes that are not directly connected by a parent-child relationship. For instance, a property might be used to connect the use of a variable in one part of a function to its declaration in another, a class declaration to an interface it implements, or a function call expression to the declaration of the function it calls.

Sometimes a node might be connected to another node both via a parent or child connection and by a property. An assignment statement, for example, has two child expressions (one on the left-hand side of the = and one on the right-hand side). These expressions can also be reached individually by the `lhs` and `rhs` properties. This enables rules to perform more precise queries against the tree. For instance, a query that looks for an assignment with `x` as a child would match both `x = y` and `y = x`, but a query that looks for an assignment with `x` as `lhs` would match `x = y` but not `y = x`.

Each node in the structural tree has a type, referred to as the *structural type*. The structural type of a node that represents a function declaration is different than the structural type of a node that represents a class declaration, and is likewise different from the structural type of a node that represents an expression.

Structural types enable you to write queries that look for certain types of nodes. The structural type of a node also determines the set of properties that it has. The *Structural Type and Property Reference* provides a full list of all structural types and their properties.

Structural Tree Query Language

The structural tree query language enables the Structural Analyzer to perform complex matches against the structural tree. Each structural rule contains a single query. The Structural Analyzer reports an issue for each construct in the program that matches that query.

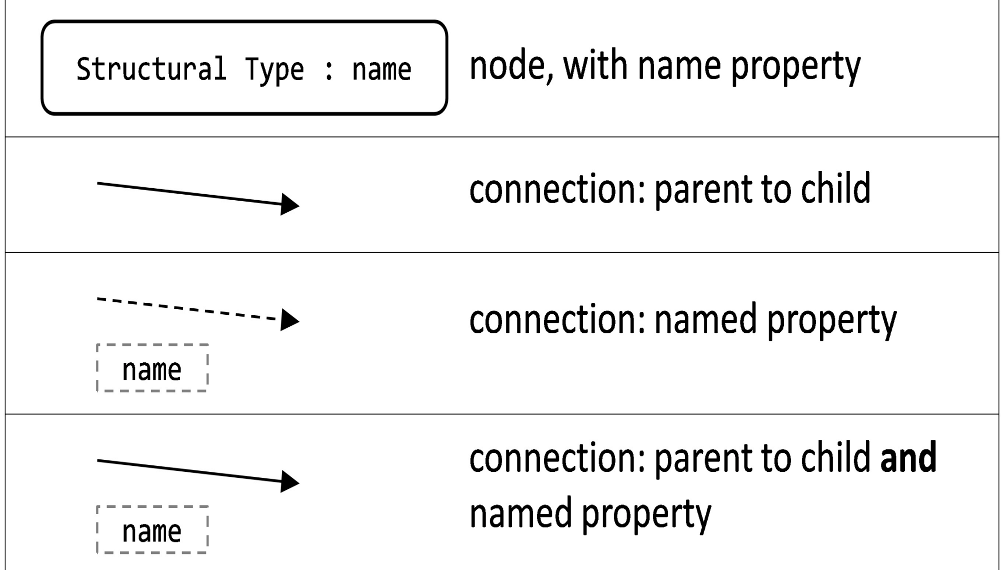
To write a query that matches a specific code construct, you must understand how the code looks when represented in a structural tree. The query expresses constraints in terms of the structural type of nodes to match and the relationships between those nodes (parent-child and property relationships).

Structural Tree Examples

The following examples shows the construction of a simplified structural tree for a small Java program. Each example includes program source code, a diagram of the structural tree, and an explanation.

These examples include structural tree diagrams for illustrative purposes. These diagrams exclude some database attributes for the sake of simplicity. As the example program becomes more complex, some edges shown in the tree are omitted. This is to make the illustration easier to read.

Use the following legend to interpret diagrams in the examples.

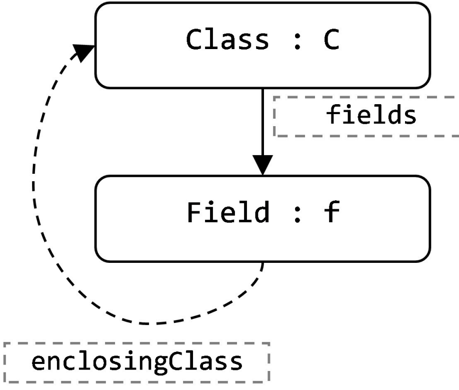


Example 1

The following program consists only of a class with a single member field:

```
class C {
   private int f;
}
```

In the structural tree, the field is related to the class with the fields property, which lists all the fields of a class.

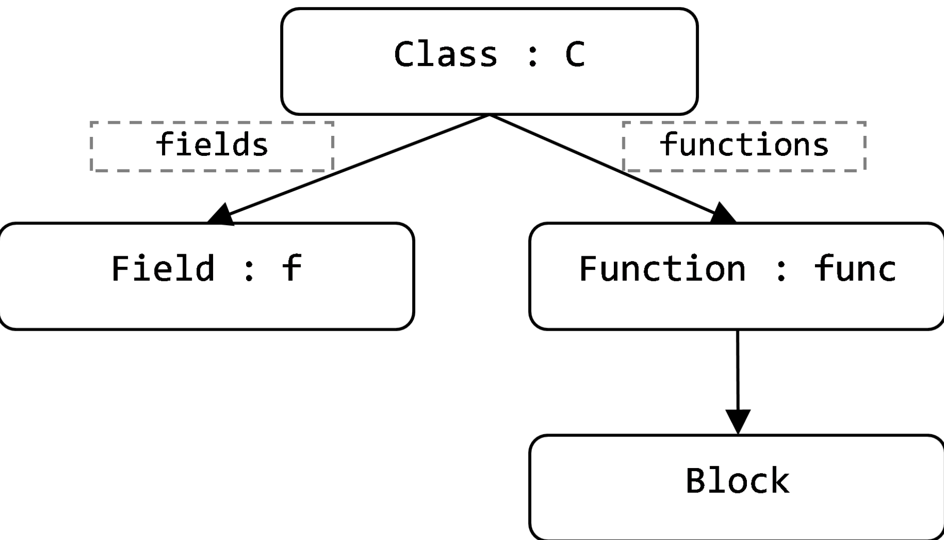


Example 2

This example adds an empty function to the class:

```
class C {
   private int f;
   void func() {
   }
}
```

The structural tree now includes nodes for the function and its body block.



A query to specifically match the field in this code could look like the following:

```
Field field: field.name == "f" and field.enclosingClass is
[Class class: class.name == "C"]
```

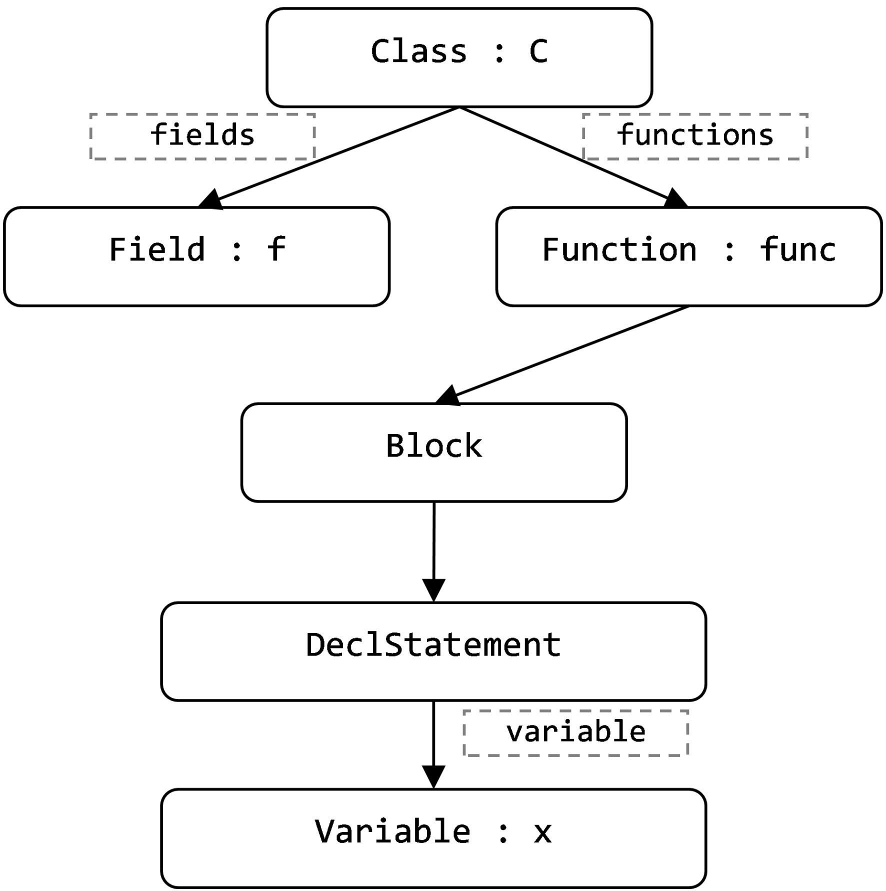
The query includes constraints on the name properties of the class and field nodes, so it would no longer match the code if the class or field were renamed. Normally, structural queries are designed to be less specific than this example.

Example 3

This example adds a local variable declaration to the function:

```
class C {
   private int f;
   void func() {
      int x;
   }
```

The body block now has a child node for the statement that declares the variable.

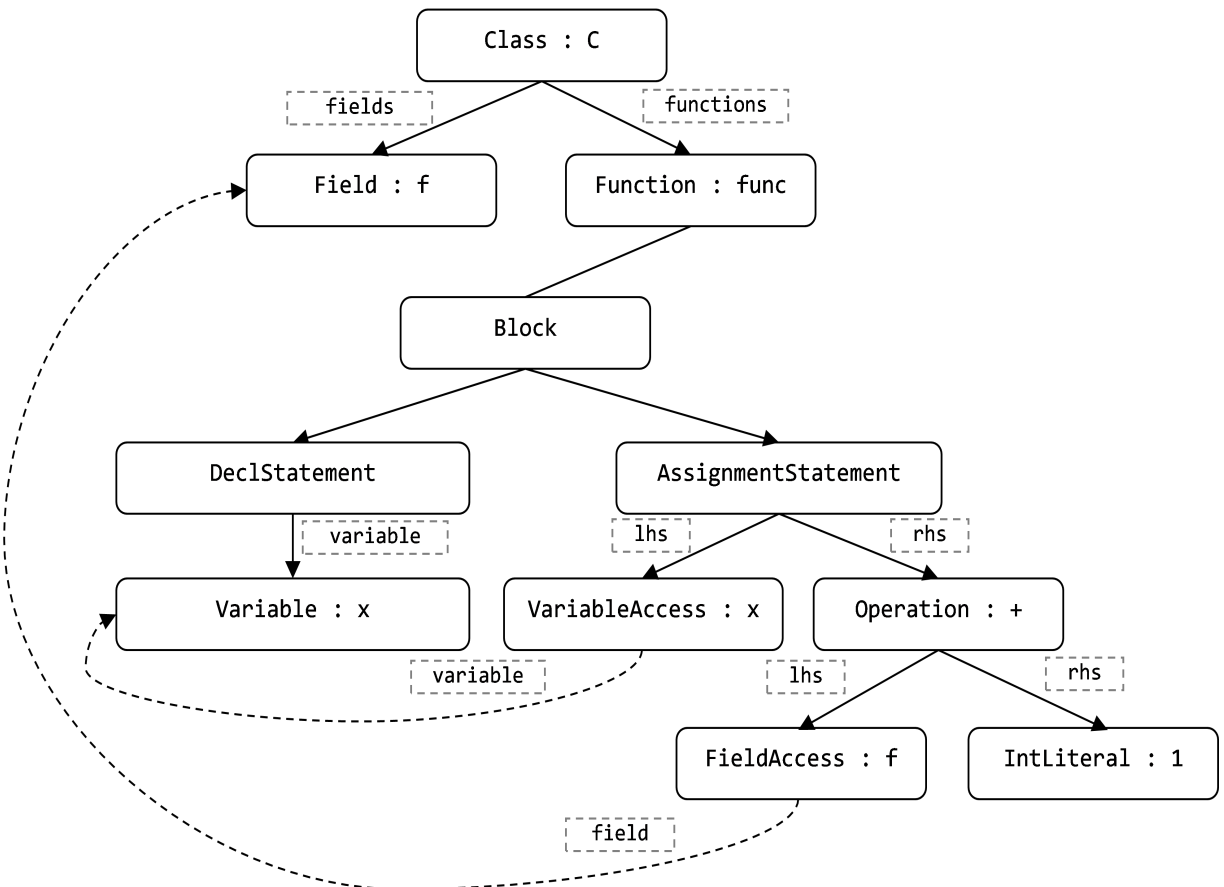


Example 4

The following final version of the program has an additional statement that performs arithmetic on field value and assigns the result to a local variable:

```
class C {
   private int f;
   void func() {
      int x;
      x = f + 1;
   }
}
```

The structural tree now includes an assignment statement that relates two expressions. The left-hand side expression (`lhs`) denotes the location being assigned to, while the right-hand side (`rhs`) is the value being assigned. The expression on the right-hand side of the assignment breaks down further into an operation (addition) on two components: the field and an integer. The expressions that access the field and variable include properties that connect to the corresponding declarations.



As an example, the following query matches any assignment in the program in which the location being written to is a local variable and the expression for the value includes a read of a field, which belongs to the same class as the class in which the function appears. This matches the previous example code. Unlike the query in "Example 2" on page 34, it does not include constraints on names. It is general enough to match similar code patterns in other parts of the program.

```
AssignmentStatement a: a.lhs is [VariableAccess:] and a.rhs contains
[FieldAccess fa: fa.field.enclosingClass ==
a.enclosingFunction.enclosingClass]
```

XML Representation of Structural Analyzer Rules

The following example shows a structural rule that matches on all instances of functions named `hashcode`:

```
<StructuralRule formatVersion="23.1" language="java">
  <RuleID>5707596F-F163-7D69-35F6-B18C9FEFDB1B</RuleID>
  <VulnCategory>Confusing Method Name</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Predicate><![CDATA[
      Function: name is "hashcode"
  ]]></Predicate>
</StructuralRule>
```

The following table describes the XML elements introduced in the structural rule shown in the previous example.

Element	Description
Predicate	This element specifies one or more structural queries. If a program construct matches the query contained in any `<Predicate>` element, the Structural Analyzer reports a vulnerability for that program construct. Enclose the contents of the `<Predicate>` element in a CDATA section to avoid the need to escape any XML special characters in the query.

Custom Structural Rule Scenarios

This section provides examples of structural rules. You can use these examples as a base from which to write your custom rules. Match your requirement with one of the examples, and tailor the rules to suit your software.

This section contains the following topics:

Leftover Debug 39
Dangerous Function Calls 40
Overly Broad Catch Blocks 42
Password in Comments 45
Poor Logging Practice 46
Empty Catch Block 47

Leftover Debug

This scenario highlights the rules necessary for the Structural Analyzer to detect leftover debug code. This scenario demonstrates how leftover debug code can introduce unexpected vulnerabilities in a production environment. It then shows custom rules that identify this type of vulnerability.

This scenario highlights the following type of vulnerability:

- Leftover debug code—Debug code can expose unintended functionality in a deployed application.

This scenario highlights the following analysis and rule concepts:

- Function construct objects
- Slot construct objects
- Startswith operator
- Structural rule

Source Code

The application contains methods that developers call to debug the retrieval of sensitive data. The following code shows how a developer temporarily debugs this method. The `debugTransactions()` method is called to examine the contents of the transactions.

```
public static List getTransactions(String acctno) throws Exception {
  ...
  // TODO: remove this before deploying to production
  debugTransactions(transactions);
  return transactions;
}
```

The following code shows how the application debugs the transaction:

```
public static void debugTransactions(List transactions) throws Exception {
  Logger debugLogger = Logger.getLogger(TransactionService.class.getName
());
  debugLogger.setLevel(Level.FINEST);
  FileHandler fh = new FileHandler("debug.log");
  fh.setLevel(Level.FINEST);
  debugLogger.addHandler(fh);

  for (int index=0; index < transactions.size(); index++) {
  Transaction proposedTransaction = (Transaction)transactions.get(index);

  debugLogger.finest("Request transaction statement: "
```

```
    + proposedTransaction.getId()+": "
    + proposedTransaction.getAcctno() + "; "
    + proposedTransaction.getAmount() + "; "
    + proposedTransaction.getDate() + "; "
    + proposedTransaction.getDescription());
  }
}
```

This method records sensitive data in an unencrypted log file. If the application executes this method within a production environment, sensitive data is written to an unencrypted file. This raises the risk of accidental disclosure of sensitive data to a third party.

Rules

There is a common method signature that identifies every debug method in the application. The code in the "Source Code" on the previous page illustrates that each debug method's name starts with the word `debug`. Also, the method accepts one parameter of type `java.util.List`.

The structural rule in the following example identifies all methods that match this debug signature:

```
<StructuralRule formatVersion="23.1" language="java">
  <RuleID>8206ED21-9FB0-44AC-9058-6FCDA601E699</RuleID>
  <VulnCategory>J2EE Bad Practices</VulnCategory>
  <VulnSubcategory>Leftover Debug Code</VulnSubcategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Predicate><![CDATA[
   Function: name startsWith "debug" and
    parameterTypes.length == 1 and
    parameterTypes[0].name == "java.util.List"
  ]]></Predicate>
</StructuralRule>
```

The analyzer uses this rule to identify and report all debug methods. First, the rule inspects each function object's name property to verify that the method's name begins with the word `debug`. Then the rule verifies that there is only one parameter to this method. The rule then verifies that the parameter is of type `java.util.List`.

Dangerous Function Calls

This scenario highlights the rules that are necessary for the Structural Analyzer to detect dangerous function call vulnerabilities. The scenario illustrates why an application should never call particular methods. It then shows how the Structural Analyzer uses structural rules to identify the dangerous function call vulnerability.

This scenario highlights the following vulnerabilities:

- Dangerous method—Never use functions that are unsafe

This scenario highlights the following analysis and rules concepts:

- FunctionCall construct object
- Structural rule

Source Code

A cross-site scripting vulnerability exists in the application. A validation function attempts to mitigate this vulnerability. However, it is inadequate and does not fully remove the cross-site scripting vulnerability. Do not use this function for any current or future projects within the organization.

The application receives messages from the user and writes the contents to a database. Persistent cross-site scripting vulnerabilities might result.

The following code sample shows a method that is called to filter any malicious characters from the messages before the application writes them to disc:

```
private static Message validateMessage(Message incomingMessage) throws
Exception {
  // Validate sender
  String incomingSender = incomingMessage.getSender();
  if ((incomingSender == null) || (incomingSender.length() == 0))
    throw new Exception("invalid sender in message");

  // Validate subject
  String incomingSubject = incomingMessage.getSubject();
  if (incomingSubject == null)
    throw new Exception("invalid subject in message");

  // Validate severity
  String incomingSeverity = incomingMessage.getSeverity();
  if ((incomingSeverity == null) || (incomingSeverity.length() == 0))
    throw new Exception("invalid sender in message");

  // Validate body
  String incomingBody = incomingMessage.getBody();
  if (incomingBody == null)
    throw new Exception("invalid sender in message");
  return incomingMessage;
}
```

The function does not perform an allow list-type validation of the `incomingMessage` message, and no application code should ever call this function.

Rules

The following structural rule identifies all instances where the application calls the `MessageService.validateMessage()` method:

```
<StructuralRule formatVersion="23.1" language="java">
  <RuleID>95C67A96-5AF7-402E-B451-6CEFF4EB8973</RuleID>
  <VulnKingdom>API Abuse</VulnKingdom>
  <VulnCategory>Dangerous Method</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Predicate><![CDATA[
    FunctionCall call: call.function.name == "validateMessage" and
    call.function.enclosingClass.name ==
      "com.fortify.samples.riches.model.MessageService"
  ]]></Predicate>
</StructuralRule>
```

The rule uses the `FunctionCall` construct object to inspect every method that the application calls. The Structural Analyzer reports a vulnerability when the conditions of the rule are met.

Overly Broad Catch Blocks

This scenario demonstrates how overly broad catch blocks can cause security issues. The scenario then provides examples of rules that work with the Structural Analyzer to find vulnerabilities caused by overly broad catch blocks.

This scenario highlights the following vulnerability:

- Poor error handling-broad catch—The catch block handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.

This scenario highlights the following analysis and rules concepts:

- CatchBlock construct object
- Contains operator
- Exception construct object
- Not operator
- ThrowStatement construct object
- Structural rule

Source Code

The following code shows an example with overly broad exception handling code:

```
public static void addMessage(Message message) {
  Session session = null;
  try {
    session = ConnectionFactory.getInstance().getSession();
    Transaction tx = session.beginTransaction();
    session.save(message);
    tx.commit();
    session.flush();
    session.close();
    }
  catch(Exception e) {
    // Treat all exceptions the same here
  }
}
```

The catch block catches the generic `Exception` class. Ideally, separate catch blocks handle specific or relevant security exceptions individually. Programs should process these security exceptions separately to create audits that are necessary to track bugs and detect security breaches.

Not every overly broad catch block represents a problem. For example, the following code catches all exceptions and throws them up the call stack:

```
public static boolean isAdmin(int roleid) throws Exception {
  boolean auth = false;
  Connection conn = ConnFactory.getInstance().getConnection();
  ResultSet rs = null;
  try {
    Statement statement =  conn.createStatement();
    rs = statement.executeQuery("SELECT rolename FROM auth WHERE
      roleid = " + roleid);
    rs.next();

    if (rs !=null && rs.getString("rolename").equals("admin"))
      auth = true;
    conn.close();
  }
  catch(Exception e) {
    throw e;
  }
  return auth;
```

```
}
```

A higher catch block can handle the exception correctly. It is also acceptable to perform a broad catch at the highest-level method of the application. The following code shows an example of an appropriately broad catch block that catches all exceptions immediately before they exit the program:

```
public static void main(String args[]) {
try {
  BannerAdServer obj = new BannerAdServer();
  BannerAdSource stub =
    (BannerAdSource)UnicastRemoteObject.exportObject(obj, 0);

  // Bind the remote object's stub in the registry
  Registry registry = LocateRegistry.getRegistry();
  registry.bind("BannerAdSource" stub);
}
catch (Exception e) {
  // Process any exceptions that are not handled anywhere else
}
```

Rules

A rule needs to report all overly broad catch blocks that are not defined within the `main()` method and do not throw the exception up the call stack. The following rule reports catch blocks that meet these requirements:

```
<StructuralRule formatVersion="23.1" language="java">
  <RuleID>C9ECD6EC-DAA1-41BE-9715-033F74CE664F</RuleID>
  <VulnCategory>Poor Error Handling</VulnCategory>
  <VulnSubcategory>Overly Broad Catch</VulnSubcategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Predicate><![CDATA[
    CatchBlock: exception.type.name == "java.lang.Exception" and
    not contains [ThrowStatement: ] and
    not (enclosingFunction.name == "main")
  ]]></Predicate>
</StructuralRule>
```

This rule identifies all catch blocks in the program that use the catch blocker and inspects the class type of the exception caught in each catch block. The `exception.type.name` property describes the name of the class specified by the catch block. This property must equal the generic exception class `java.lang.Exception` for the rule to report this catch block.

The rule then excludes catch blocks that contain a `ThrowStatement`, which represents a throw statement inside the catch block.

The catch block construct object's `enclosingFunction.name` property defines the name of the method that contains the catch block, which must not equal the value `main`.

When a catch block satisfies all three of these conditions, the Structural Analyzer reports an overly broad catch vulnerability.

Password in Comments

This scenario demonstrates a rule that enables the Structural Analyzer to detect passwords in comments. This includes how passwords might appear in comments and how an attacker can exploit this vulnerability. The scenario then shows how the Structural Analyzer uses rules to identify this type of vulnerability.

This scenario highlights the following vulnerability:

- Password management: passwords in comments—Hardcoded passwords can compromise system security in a way that you cannot easily remedy.

This scenario highlights the following analysis and rules concepts:

- Comment construct object
- Java regular expressions
- Structural rule

Source Code

If the source code of an application contains authentication credentials for the production database, anyone with access to the development environment and its source code can access data in a production environment.

The following code shows hardcoded credentials in the `ProfileService` class:

```
public class ProfileService {
  // NOTE: sample profiles can be reproduced through internal server
  // host: db1.riches.com; username: service, password: passw0rd1!
{
```

Rules

The following structural rule identifies text that contains the word `password` in a comment block, inline comment, or JavaDoc:

```
<StructuralRule formatVersion="23.1" language="java">
  <RuleID>C938AE93-EA38-403b-ABDA-3F01BEFA7933</RuleID>
  <VulnCategory>Password Management</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
```

```
  <Description/>
  <Predicate><![CDATA[
    Comment c: (c.doc or c.inline or c.block)
       and c.text matches "(?i).*password.*"
  ]]></Predicate>
</StructuralRule>
```

First, this rule inspects the `doc`, `inline`, and `block` properties of every comment construct object in the application. If one of these properties is true, the comment satisfies the criterion that it must be a block, inline, or JavaDoc comment.

Then the rule inspects the text property of the object text to see if the value of the property value matches the Java regular expression `'(?i).*password.*'`. This expression matches any text that contains `password` anywhere in its value, regardless of capitalization.

The Structural Analyzer reports an issue when it finds a comment that satisfies both of these conditions.

Poor Logging Practice

This scenario demonstrates a rule that enables the Structural Analyzer to identify logging objects that are not declared static and final. The scenario demonstrates a poor logging practice. Then it illustrates the way the Structural Analyzer uses rules to identify this type of issue.

This scenario highlights the following vulnerability:

- Poor logging practice: logger not declared static final—Declare loggers to be static and final.

This scenario highlights the following analysis and rules concepts:

- Class construct objects
- Contains operator
- Field construct objects
- Not operator
- Structural rule

Source Code

A good programming practice is to share a single logger object between all the instances of a specific class and to use the same logger throughout the duration of the application. The way the application implements `ConnectionClass` class in the following example illustrates a violation of this practice:

```
public class ConnectionFactory {
  private static Logger log =
    Logger.getLogger(ConnectionFactory.class.getName());
  private static ConnectionFactory instance = null;
```

Rules

The following rule reports any instance of the `java.util.logging.Logger` object that is declared as a field but is not using both the `static` and `final` keywords:

```
<StructuralRule formatVersion="23.1" language="java">
  <RuleID>B95EB686-8EBC-498F-B332-55E31F9DFB8A</RuleID>
  <VulnCategory>Poor Logging Practice</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Predicate>
    Field f: not (static and final) and type.definition.supers contains
    [Class: name == "java.util.logging.Logger
  </Predicate>
</StructuralRule>
```

To identify an improperly declared `Logger` field object, the Structural Analyzer inspects the static and final properties of every `Field` construct object. If either value is false, the field satisfies the rule's first set of conditions.

After a `Field` construct object satisfies the first condition, the rule inspects the `Field` object's declared type. The field must be an instance of a `java.util.logging.Logger` object or an extension that inherits from that class.

If a `Field` construct object satisfies both conditions, the Structural Analyzer reports the field declaration as an issue.

Empty Catch Block

This scenario shows a rule for the Structural Analyzer to detect empty catch block vulnerabilities. The scenario demonstrates how an attacker can exploit an empty catch block vulnerability. It then shows how the Structural Analyzer uses structural rules to identify this type of vulnerability.

The scenario highlights the following vulnerability:

- Poor error handling: empty catch block—Ignoring an exception can cause the program to overlook unexpected states and conditions.

The scenario highlights the following analysis and rules concepts:

- Catch block construct object
- Structural rule

Source Code

The following code builds `Hibernate` sessions that the application uses in subsequent database operations. The `ConnectionFactory` class constructor contains code that might throw software exceptions.

```
private ConnectionFactory() {
try {
  String pFile = System.getProperty("ConnectionFactory.pfile");
  if (pFile != null) {
    java.util.Properties props = new java.util.Properties();
    props.load( new java.io.FileInputStream(pFile) );
    }
}
catch (Exception e) {
  //TODO: fill in this code
}
...
```

In this code, the catch block is empty. The application cannot maintain an accurate log of any security events that might occur.

Rules

To identify the empty catch block shown in the previous code example, the Structural Analyzer should examine the empty property of each `CatchBlock` construct object. This boolean property indicates that the corresponding catch block does not contain any code.

The following rule identifies empty catch blocks:

```
<StructuralRule formatVersion="23.1" language="java">
  <RuleID>D693090B-3F8C-48BD-BCDE-C6DCA2266710</RuleID>
  <VulnCategory>Poor Error Handling</VulnCategory>
  <VulnSubcategory>Empty Catch Block</VulnSubcategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Predicate><![CDATA[
    CatchBlock: empty
  ]]></Predicate>
</StructuralRule>
```

With this structural rule, the Structural Analyzer reports any empty catch blocks in the application.

Chapter 4: Dataflow Analyzer Rules

This section contains the following topics:

Dataflow Analyzer and Custom Rules 49

Dataflow Analyzer and Custom Rules Concepts 50

XML Representation of Dataflow Analyzer Rules 55

Custom Dataflow Analyzer Rule Scenarios 67

Dataflow Analyzer and Custom Rules

The Fortify Static Code Analyzer Dataflow Analyzer finds security issues that involve tainted data (user-controlled input) that is put to potentially dangerous use. The Dataflow Analyzer uses interprocedural taint propagation analysis to detect the flow of data between a source (site of user input) and a sink (dangerous function call or operation).

This analysis enables Fortify Static Code Analyzer to precisely identify many different types of security problems. A common example is a SQL injection. In a SQL injection, the program eventually uses tainted data acquired from a taint source (such as an HTTP request parameter) to construct and invoke a SQL query (a taint sink). When the analyzer detects this, the Dataflow Analyzer reports a SQL injection issue.

Because the Dataflow Analyzer performs interprocedural analysis, it can track tainted data across method calls and through global variables in the program.

The Dataflow Analyzer operates on a model of the program. Fortify Static Code Analyzer constructs this model from program source code and rules. The program source code provides the base layer for the model. This layer describes the behavior of methods, the relationships between different methods, and the relationship between methods and global variables. Fortify Static Code Analyzer then augments the model with rules. These rules describe the points in the program that act as taint sources and sinks. They also describe program points that can manipulate or transfer tainted data.

> **Note:** If the model is incomplete due to necessary source code not translated into the model, or if the rules required to accurately identify the weakness are not provided, then Fortify Static Code Analyzer can potentially have false negatives in its analysis (unreported weaknesses that truly exist).

The following example of a simple program illustrates a command-injection vulnerability:

```
function run() {
  readFromNetwork(buffer);
  command = concatenate("/usr/bin/", buffer);
  execute(command);
}
```

In this example, `readFromNetwork()`, `concatenate()`, and `execute()` are API calls to a standard library linked to the program. The call `readFromNetwork()` reads the tainted input into the buffer. The `run()` function then concatenates the buffer with a string literal to form `command`, passes `command` to the `execute()` function, which executes a new process specified by the command string.

By building a model from the source code, the Dataflow Analyzer can understand that `run()` calls three external functions and that there is a dataflow relationship among those calls through local variables.

Because the source code for those functions is not part of the program, the model is incomplete without a set of rules that describe the relevant characteristics of those functions. Without any knowledge of the external functions, the Dataflow Analyzer cannot determine how tainted data enters and moves through the program.

For this example, the Dataflow Analyzer can detect the vulnerability with the following dataflow rules and taint characterization rules:

- A DataflowSourceRule or Characterization TaintSource rule for `readFromNetwork()`
- A DataflowPassthroughRule or Characterization TaintTransfer rule for `concatenate()`
- A DataflowSinkRule or Characterization TaintSink rule for `execute()`

Dataflow Analyzer and Custom Rules Concepts

This section provides information on dataflow core concepts. These concepts coincide with rules that you can write to instruct Fortify Static Code Analyzer on how Dataflow Analyzer models the code. This section also provides more advanced concepts that illustrate how the Dataflow Analyzer performs in a given situation.

This section contains the following topics:

Taint Source 51
Taint Write 51
Taint Entrypoint 51
Taint Sink 52
Taint Passthrough / Transfer 52
Taint Cleanse 52
Taint Flags 53
Taint Path 54
Validation Constructs 54
Types of Dataflow Analyzer Rules 55

Taint Source

Tainted data enters a program through a program point called a taint source. Taint sources are function and method calls, variable and field accesses, and other expressions explicitly invoked from your source code that introduce tainted input. Common examples include:

- A function that reads data from network sources such as an HTTP request
- A function that reads data from untrusted data sources (for example, a database to which other programs write)
- Access to a field that stores input collected from a user

Taint Write

Taint write is a source of taint introduced with a write operation as opposed to taint source that introduces tainted input with a read operation. Common examples of taint write include writing to a variable that is used as:

- An argument to a function call
- An instance object of a function call
- A field access
- A variable access
- An array access

Taint Entrypoint

A taint entrypoint is a special type of taint source that describes a function invoked with tainted input from the environment, framework, or the arguments passed. Some programming frameworks invoke a function in an application without any explicit call in the user code. For example, the `main()` function, common in most modern programming languages, acts as the entrypoint during execution. The operating system invokes the `main()` function executables written in C or C++ at runtime. The Java Runtime Environment invokes the `main()` function in Java applications. The JVM running in the web browser invokes the `init()` and `destroy()` functions in Java applets when a page that contains the applet is first visited and when the browser is closed, respectively.

Common examples include:

- The main function of a program that is called with arguments specified on the command line
- A function in a web application framework that is called directly by the framework with an input parameter
- Parameters to a function with an annotation that indicates that the underlying framework invokes the function with an input parameter

Taint Sink

Taint sinks are program points to which tainted data must not flow. When the Dataflow Analyzer detects a path through which tainted data can flow from source to sink, it reports an issue. A taint sink rule can contain a conditional expression that, by examining taint flags, limits which paths that end at a sink are reported.

Common examples include:

- A function that takes a SQL string and executes a query against a database connection
- A function that takes a string and executes the command described by the string
- An assignment to a variable that is automatically written to a web page by an underlying framework

Taint Passthrough / Transfer

The Dataflow Analyzer automatically tracks data and propagates taint (passthrough behavior) for functions defined in the source code. You must model externally defined functions (source code not available for translation, such as in the JDK library) with passthrough behavior using a rule.

For example, the default Fortify Secure Coding Rulepacks contain a rule that describes the passthrough behavior of `StringBuilder.append()`.

A passthrough or taint transfer rule might add or remove taint flags from the tainted data.

Taint Cleanse

A taint cleanse is a point at which taint is removed or modified. Typically, this is a validation function.

There are two types of taint cleanse points:

- Complete cleanse—Rule that describes a taint cleanse that does not specify taint flags to add or remove. The Dataflow Analyzer stops taint propagation completely at this point.
- Partial cleanse —Rule that specifies to add or remove taint flags. In this instance, the data is still tainted, but the taint flag set changes.

Cleanse rules are always the last set of rules applied during the scan. If a cleanse rule matches a function call, field, variable, or other code construct, the cleanse rule applies on top of the taint path. It is applied after any passthrough rules that match the same code construct.

It is often possible to describe a code construct in terms of either a passthrough or a cleanse rule. See "Validation Constructs" on page 54 for a description of the differences between passthrough rules and cleanse rules.

Taint Flags

A taint flag is an attribute of tainted data that enables the Dataflow Analyzer to discriminate among different types of taint. This is important because it enables the Dataflow Analyzer to accurately identify issues.

For example, the input from both HTTP parameters and local configuration files of a web application might be tainted. The attack vectors in each instance are substantially different. An attacker can easily manipulate HTTP parameters. Manipulating configuration files on the system is much more difficult.

Consider a function that checks input for SQL metacharacters. After tainted data passes through this function, it is safe to use in a taint sink for SQL injection. However, you cannot consider the data untainted. It is still dangerous to use in other contexts, such as a taint sink for command injection. The use of taint flags in rules enables the Dataflow Analyzer to determine whether the tainted data is safe in a specific context.

Each taint path through the program carries a set of taint flags. The Dataflow Analyzer can add or remove taint flags that originated at the taint source point as taint passes through passthrough and cleanse points in the program. A taint sink can check for the presence or absence of taint flags that determine whether the Dataflow Analyzer reports a path from source to sink.

Taint Flag Types

Fortify Static Code Analyzer provides three types of taint flags. These taint flag types help to simplify writing conditional expressions for taint sinks.

- General—This is the default taint flag type. These usually indicate a source of untrusted data and are used for tracking data that comes into the application (such as SQL injection, cross-site scripting, and so on).
- Neutral—These taint flags represent “informational” content. Neutral taint flags are most often used to note that a specific vulnerability category has been validated. Neutral taint flags are useful in filtering out false positives. These are usually used to describe properties of data.
- Specific—You create specific taint flags by including a declaration that describes the category of taint flag in the Rulepack. These are usually used for tracking data that leaves the application (such as system information leak, privacy violation, and so on).

For descriptions of the taint flags included with the Fortify Secure Coding Rulepacks, see "Taint Flag Reference" on page 134.

Taint flag typing provides a method to introduce new types of taint into the system without producing unexpected results. Specific taint flags enable a rule writer to create a pairing of source and sink rules. In such a pairing, taint from the paired source rule does not interact with other sinks. Likewise, any taint from other sources in the program cannot interact with the paired sink.

For example, consider a program that uses the APIs `getSecret()` and `shareData()`. In this example, `getSecret()` returns secret data, the output of which should never get passed to `shareData()`. You can write a rule that prevents this by describing `getSecret()` as a taint source and `shareData()` as a taint sink.

This works fine if these are the only rules used to analyze the program. However, if you use the default Secure Coding Rulepacks to scan the program, Fortify Static Code Analyzer might report unintended issues. For example, Fortify Static Code Analyzer might report input from HTTP parameters reaching `shareData()`, or input from `getSecret()` being used in a SQL query, even though these usages are safe.

For these rules to work more precisely, you can introduce a new taint flag (SECRET) to the source and sink rules. The source rule would add the SECRET taint flag, and the sink rule would check for the presence of the SECRET taint flag.

This solves half of the problem; the sink at `shareData()` only reports input from `getSecret()` and not from other sources. However, input from `getSecret()` might unintentionally trigger the reporting of issues at other sinks, because those sinks do not explicitly check against the absence of the new SECRET taint flag. This example demonstrates the benefit of specific taint flags. By declaring the SECRET taint flag as specific, taint from the `getSecret()` source is prevented from interacting with existing sinks in unintended ways. Sinks that do not explicitly check for the specific taint flag SECRET ignore the taint from `getSecret()`.

Taint Flag Behavior

It can be challenging to understand the exact behavior of sinks in the presence of different types of taint. For any sink that does not explicitly check for the presence or absence of any specific taint flag in the taint flag set, Fortify Static Code Analyzer automatically adds a check to ensure that the taint flag set is not specific. A taint flag set is specific if it contains one or more specific taint flags and does not contain any general taint flags.

Taint Path

The Dataflow Analyzer reports a vulnerability when it finds one or more taint paths between a source and a sink in the application.

A taint path contains a sequence of method calls, stores (assignment variables or fields), and loads (reads from variables or fields). It denotes a path along which tainted data propagates from a taint source point to a taint sink point. In fact, because a program can contain loops or recursion, there can be an infinite number of paths. The Dataflow Analyzer cannot consider all taint paths from a source to a sink due to performance implications. However, it does consider every path that has a unique set of taint flags applied to it. Some of the taint flags indicate that something is validated, which means that the Dataflow Analyzer does not report an issue on that path. If another path exists on which the data is not validated, Dataflow Analyzer reports an issue.

Validation Constructs

One of the most basic rule-writing tasks is to write rules for validation constructs such as validation functions. You can do this by either writing a passthrough or cleanse rule. The rule that is appropriate depends on the circumstances.

In cases where the function completely validates the input for all cases, a complete cleanse rule (which removes all taint) is appropriate. In most cases, it is preferable to add a taint flag to the taint path to indicate that a certain type of validation was performed.

For examples of validation construct rules, see "Validation Construct Examples" on page 85.

Types of Dataflow Analyzer Rules

Dataflow analysis requires the definition of source and sink rules to mark the endpoints of a dataflow path tracking tainted data. Understanding the context of the various operations that occur along the dataflow path can lead to more precise rules and can reduce the number of false positives reported during analysis.

You can only use dataflow rules to define taint sources, sinks, and passthroughs around function calls. Taint characterization rules provide a way to specify the characteristics of the execution context of the various functions in the dataflow. You can also use taint characterization rules to track tainted data through arbitrary expressions. This creates more concise constraints for the various nodes in a dataflow trace. Taint characterization rules are considered a superset of dataflow rules. The following table summarizes the types of rules you can use to represent various Dataflow Analyzer rule concepts.

Concept	Dataflow Rule	Taint Characterization Rule
Taint Source	DataflowSourceRule	TaintSource TaintWrite
Taint Entrypoint	DataflowEntryPointRule	TaintEntrypoint
Taint Passthrough	DataflowPassthroughRule	TaintTransfer
Taint Sink	DataflowSinkRule	TaintSink
Taint Cleanse	DataflowCleanseRule	TaintCleanse

XML Representation of Dataflow Analyzer Rules

The following example shows the general XML structure of a dataflow rule:

```
<DataflowXYZRule formatVersion="23.1" language="...">
  <RuleID>...</RuleID>
  <MetaInfo>...</MetaInfo> <!-- DataflowSinkRule only -->
  <VulnKingdom>...</VulnKingdom> <!-- DataflowSinkRule only -->
  <VulnCategory>...</VulnCategory> <!-- DataflowSinkRule only -->
  <VulnSubCategory>...</VulnSubcategory> <!-- DataflowSinkRule only -->
  <Description>...</Description> <!-- DataflowSinkRule only -->
  <FunctionIdentifier>
      ...
```

```
  </FunctionIdentifier>
  <InArguments>...</InArguments> <!-- Optional -->
  <OutArguments>...</OutArguments> <!-- Optional -->
  <TaintFlags>...</TaintFlags> <!-- Optional -->
</DataflowXYZRule>
```

The <InArguments>, <OutArguments>, and <TaintFlags> elements are described in the following relevant dataflow rule sections.

Note: In the DataflowSinkRule, a <Sink> element encloses the <InArguments> element (<Sink><InArguments>...</InArguments</></Sink>).

The following example shows the general XML structure of a taint characterization rule:

```
<CharacterizationRule formatVersion="23.1" language="...">
  <RuleID>...</RuleID>
  <MetaInfo>...</MetaInfo> <!-- TaintSink only -->
  <VulnKingdom>...</VulnKingdom> <!-- TaintSink only -->
  <VulnCategory>...</VulnCategory> <!-- TaintSink only -->
  <VulnSubCategory>...</VulnSubcategory> <!-- TaintSink only -->
  <Description>...</Description> <!-- TaintSink only -->
  <StructuralMatch><[CDATA[
   ...
  ]]></StructuralMatch>
  <Definition><![CDATA[
   ...
  ]]><Definition>
</CharacterizationRule>
```

The following table describes the XML elements introduced in the characterization rule shown in the previous example.

Element	Description
StructuralMatch	Specifies what the rule should match in the code using one or more structural queries. This is the same as the <Predicate> element in structural rules. For more information about the predicate language, see "Structural Rules Language Reference" on page 143. Enclose the contents of this element in CDATA section to avoid the need to escape XML special characters.
Definition	Specifies the rule type (TaintSource, TaintWrite, TaintEntrypoint, TaintSink, TaintTransfer, or TaintCleanse), input/output parameters

Element	Description
	for the taint, addition of taint, and taint flag constraints. **Note:** You can have multiple taint characterization rule properties in this element. To identify the characterization rule property that generated the issue, the index (zero-based) of the taint characterization rule property is appended to the end of the ruleID shown in the results. Enclose the contents of this element in a CDATA section to avoid the need to escape XML special characters.

This section describes the XML for the following Dataflow Analyzer rule concepts:

Source Rules 57
Sink Rules 60
Passthrough Rules 62
Entrypoint Rules 64
Cleanse Rules 65

Source Rules

Use Dataflow Analyzer source rules to identify points at which tainted data enters a program.

The following example shows a dataflow source rule that identifies the Java method `ServletRequest.getParameter()` as a source of tainted data:

```
<DataflowSourceRule formatVersion="23.1" language="java">
  <RuleID>82EBC382-1341-4A81-9FA0-3A9AF3D3EFDA</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Pattern>javax\.servlet</Pattern>
    </NamespaceName>
    <ClassName>
      <Value>ServletRequest</Value>
    </ClassName>
    <FunctionName>
      <Value>getParameter</Value>
    </FunctionName>
    <ApplyTo implements="true" overrides="true" extends="true"/>
  </FunctionIdentifier>
  <OutArguments>return,this</OutArguments>
  <TaintFlags>+WEB,+XSS</TaintFlags>
</DataflowSourceRule>
```

The following table describes the XML elements introduced in the dataflow source rule shown in the previous example.

Element	Description
OutArguments	Determines the method parameters that introduce taint into the application. Specify parameters as a comma-delimited list of either the keywords: `return`, `this`, or `globals`, or the zero-based index of the target parameter.
TaintFlags	(Optional) Specifies the taint flags to associate with taint introduced by the method that the rule matches. Specify taint flags as a comma-delimited list. Each taint flag must include a plus (+) or minus (-) prefix to indicate whether to add or remove it from the taint path. Only the plus prefix is valid in source and entrypoint rules.

You can also write the previous dataflow source rule as two taint characterization rules:

- TaintSource to taint the return value of a method call
- TaintWrite to taint the request object on which the method is called

The following example shows a TaintSource characterization rule that is equivalent to tainting the return value in the `<OutArgument>` of the previous `<DataflowSourceRule>`:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>75A7A75E-BBB7-44CB-B12C-CE13D0DE3DC8</RuleID>
  <StructuralMatch><![CDATA[
    FunctionCall call: call.function is
      [Function f: f.name == "getParameter" and
       f.enclosingClass.supers contains
        [Class c: c.name == "javax.servlet.ServletRequest"]]
  ]]></StructuralMatch>
  <Definition><![CDATA[
    TaintSource(call, {+XSS +WEB})
  ]]></Definition>
</CharacterizationRule>
```

The previous taint characterization rule looks for a call to a method `getParameter()` defined in a class, one of the super-classes of which is the `javax.servlet.ServletRequest` class.

> **Note:** The `f.enclosingClassses.supers contains [...]` clause is equivalent to setting the `<ApplyTo>` element attributes in the `<DataflowSourceRule>` to `true`.

The `<Definition>` element specifies that the rule represents a taint source that occurs on a read operation. The following table describes the parameters of the TaintSource characterization rule type: `TaintSource(Expression, {TaintFlags})`.

Parameter	Description
Expression	Specifies an expression and optional access path (for example, `.foo.bar`) to treat as tainted. If the expression is a function call, the effect is that the return value is tainted.
TaintFlags	(Optional) Specifies the taint flags to associate with taint introduced by the construct the rule matches. Specify taint flags as a space-delimited list. Each taint flag must include a plus (+) prefix to indicate that it is added to the taint path.

The following example shows the accompanying TaintWrite characterization rule that is equivalent to tainting `this` in the `<OutArgument>` of the previous `<DataflowSourceRule>`:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>B27373A5-1403-4986-8221-112CE21D7A5F</RuleID>
  <StructuralMatch><![CDATA[
    FunctionCall call: call.function is
      [Function f: f.name == "getParameter" and
       f.enclosingClass.supers contains
        [Class c: c.name == "javax.servlet.ServletRequest"]]
      and call.instance is [Expression request: ]
  ]]></StructuralMatch>
  <Definition><![CDATA[
    TaintWrite(request, {+XSS +WEB})
  ]]></Definition>
</CharacterizationRule>
```

The previous taint characterization rule looks for a call to a method `getParameter()` defined in a class, one of the super-classes of which is `javax.servlet.ServletRequest` class. The `<Definition>` element specifies that the rule represents a taint source that occurs on a write operation. The following table describes parameters of the TaintWrite characterization rule type: `TaintWrite(Expression, {TaintFlags})`.

Parameter	Description
Expression	Specifies an expression and optional access path (for example, `.foo.bar`) that is treated as tainted.
TaintFlags	(Optional) Specifies the taint flags to associate with taint introduced by the construct the rule matches. Specify taint flags as a space-delimited list. Each flag must include a plus (+) prefix to indicate that it is added to the taint path.

Sink Rules

Use Dataflow Analyzer sink rules to identify points in a program that tainted data must not reach.

The following example shows a dataflow sink rule that indicates taint must not reach the `Statement.executeQuery()` method:

```
<DataflowSinkRule formatVersion="23.1" language="java">
  <RuleID>CCC3523A-3E34-4890-8F7D-23F16CB3C4339</RuleID>
  <VulnCategory>SQL Injection</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
```

```
  <Description/>
  <Sink>
    <InArguments>0</InArguments>
  </Sink>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.sql</Value>
    </NamespaceName>
    <ClassName>
      <Value>Statement</Value>
    </ClassName>
    <FunctionName>
      <Value>executeQuery</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
</DataflowSinkRule>
```

The following table describes the XML element introduced in the previous dataflow sink rule example.

Element	Description
InArguments	Specifies the parameters of the method that must not receive taint. If taint reaches any of these parameters, Fortify Static Code Analyzer reports an issue. Specify parameters as a comma-delimited list of either the keywords: `return`, `this`, or `globals`, or the zero-based index of the target parameter.

You can also write the previous dataflow sink rule as a taint characterization rule. The following example shows a TaintSink characterization rule:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>0434D772-00B8-44AA-A23F-04C9BD7115D3</RuleID>
  <VulnCategory>SQL Injection</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <StructuralMatch><![CDATA[
    FunctionCall call: call.function is
     [Function f: f.name == "executeQuery" and f.enclosingClass.supers
contains
       [Class c: c.name == "java.sql.Statement"]]
     and call.arguments[0] is [Expression inArgument: ]
  ]]></StructuralMatch>
```

```
  <Definition><![CDATA[
    TaintSink(inArgument, [])
  ]]></Definition>
</CharacterizationRule>
```

The previous taint characterization rule looks for a call to a method `executeQuery()` defined in a class, one of the super-classes of which is the `java.sql.Statement` class, and specifies argument zero as a sink. The `<Definition>` element specifies that the rule represents a taint sink. The following table describes the parameters of the TaintSink characterization rule type: `TaintSink (Sink, TaintFlags)`.

Parameter	Description
Sink	Specifies an expression that is considered a sink for tainted data. This creates an issue if taint that satisfies the condition expression reaches the expression.
TaintFlags	(Optional) Specifies a conditional taint flag expression. Specify taint flags with the logical &&, `||`, and `!` operators, for example `(WEB || NETWORK) && !FILE`. To specify that the rule matches regardless of the taint involved, use empty braces (`{}`).

Passthrough Rules

Use Dataflow Analyzer passthrough rules to describe how functions and methods propagate taint from input to output.

The following example shows a dataflow passthrough rule that indicates that taint on the string on which the `trim()` method is called is also returned from the method:

```
<DataflowPassthroughRule formatVersion="23.1" language="java">
  <RuleID>8A1E8BA1-6C03-4F77-B648-75C3CF3C28CB</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.lang</Value>
    </NamespaceName>
    <ClassName>
      <Value>String</Value>
    </ClassName>
    <FunctionName>
      <Value>trim</Value>
    </FunctionName>
  </FunctionIdentifier>
  <InArguments>this</InArguments>
  <OutArguments>return</OutArguments>
```

```
<DataflowPassthroughRule>
```

The dataflow passthrough rule in the previous example combines the concepts of `<InArguments>` and `<OutArguments>` to map taint that enters the method on one parameter to taint exiting the method on another parameter. If a passthrough rule includes taint flags, which the previous example does not, those taint flags are added (flags prepended with a plus +) or removed (flags prepended with a minus -) from the parameter specified by the `<OutArguments>` element.

You can also write the previous dataflow passthrough rule as a taint characterization rule. The following example shows a TaintTransfer characterization rule:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>C30E2CFB-5133-48FF-86A5-854E4D282631</RuleID>
  <StructuralMatch><![CDATA[
    FunctionCall call: call.function is
      [Function f: f.name == "trim" and f.enclosingClass.supers contains
        [Class c: c.name == "java.lang.String"]]
      and call.instance is [Expression inArgument: ]
   ]]></StructuralMatch>
  <Definition><![CDATA[
    TaintTransfer(inArgument, call, {})
   ]]></Definition>
</CharacterizationRule>
```

The previous taint characterization rule looks for a call to a method `trim()` defined in a class, one of the super-classes of which is the `java.lang.String` class. This rule also specifies that taint needs to propagate from the object the method is called on to the return value. The `<Definition>` element specifies that the rule represents a passthrough. The following table describes parameters of the TaintTransfer characterization rule type: `TaintTransfer(In, Out, {TaintFlags})`.

Parameter	Description
In	Specifies an expression and optional access path (for example, `inArgument.foo`) that contains incoming taint.
Out	Specifies an expression and optional access path (for example, `.foo.bar`) that receives outgoing taint.
TaintFlags	(Optional) Specifies the taint flags to associate with taint introduced by the construct the rule matches. Specify the taint flags as a space-delimited list. Each flag must include a plus (+) or minus (-) prefix to indicate if it is added to or removed from the taint path. If you are not adding or removing taint flags, include empty braces ({}) or leave it out.

Entrypoint Rules

Use Dataflow Analyzer entrypoint rules to describe program points that introduce tainted data to a program. Entrypoint rules do this by describing the functions and methods that the program can invoke (either externally or through an internal framework or other mechanism for which the source code is not included in the analysis).

The following example shows a dataflow entrypoint rule that indicates that the array of strings passed as the first parameter to the Java `main()` method is tainted:

```
<DataflowEntryPointRule formatVersion="23.1" language="java">
<RuleID>64FDA988-4770-498E-9709-4497CCDA4E48</RuleID>
  <TaintFlags>+ARGS</TaintFlags>
  <FunctionIdentifier>
    <NamespaceName>
      <Pattern>.*</Pattern>
    </NamespaceName>
    <ClassName>
      <Pattern>.*</Pattern>
    </ClassName>
    <FunctionName>
      <Value>main</Value>
    </FunctionName>
    <Parameters>
      <ParamType>java.lang.String[]</ParamType>
    </Parameters>
    <ApplyTo implements="true" overrides="true" extends="true"/>
    <Modifiers>
      <Modifier>static</Modifier>
    </Modifiers>
  </FunctionIdentifier>
  <InArguments>0</InArguments>
</DataflowEntryPointRule>
```

The previous dataflow entrypoint rule example uses the `<InArguments>` element to define the parameters to consider tainted when the body of the specified method is analyzed.

You can also write the previous dataflow entrypoint rule as a taint characterization rule. The following example shows a TaintEntrypoint characterization rule:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>278379D6-7CA3-4195-8857-7E5435B13053</RuleID>
  <StructuralMatch><![CDATA[
```

```
    Function f: f.static and f.name == "main" and f.parameters.length == 1
      and f.parameterTypes[0] == T"java.lang.String[]"
      and f.parameters[0] is [Expression inArgument: ]
  ]]></StructuralMatch>
  <Definition><![CDATA[
    TaintEntrypoint(inArgument, {+ARGS})
  ]]></Definition>
</CharacterizationRule>
```

The previous taint characterization rule looks for a declaration of a static function `main()` that takes a `java.lang.String` array as a parameter. The `<Definition>` element specifies that the rule represents a taint entrypoint. The following table describes parameters of the TaintEntrypoint characterization rule type: `TaintEntrypoint(In, {TaintFlags})`.

Parameter	**Description**
In	Specifies an expression and optional access path (for example, `inArgument.foo`) that is treated as tainted.
TaintFlags	(Optional) Specifies the taint flags to associate with taint introduced by the construct the rule matches. Specify the taint flags as a space-delimited list. Each flag must include a plus (+) prefix to indicate it is added to the taint path.

Cleanse Rules

Use Dataflow Analyzer cleanse rules to describe validation logic and other actions that render tainted data either partially or completely cleansed. You can add or remove specified taint in a cleanse rule.

The following example shows a dataflow cleanse rule that demonstrates how the `Map.clear()` method cleanses the map:

```
<DataflowCleanseRule formatVersion="23.1" language="java">
  <RuleID>878BFCE2-6333-4AA2-8E3F-211B477FF409</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.util</Value>
    </NamespaceName>
    <ClassName>
      <Value>Map</Value>
    </ClassName>
    <FunctionName>
      <Value>clear</Value>
    </FunctionName>
    <ApplyTo implements="true" overrides="true" extends="true"/>
```

```
  </FunctionIdentifier>
  <OutArguments>this</OutArguments>
</DataflowCleanseRule>
```

The dataflow cleanse rule in the previous example uses the `<OutArguments>` element to specify that the parameters should be considered cleansed after a call to the specified method. If a cleanse rule includes taint flags, which the previous example does not, then the specified taint flags are either added (flags prepended with a plus +) or removed (flags prepended with a minus -) from the parameter specified by the `<OutArguments>` element.

You can also write the previous dataflow cleanse rule as a taint characterization rule. The following example shows a TaintCleanse characterization rule:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>C56AE490-963B-45E8-86D1-FF1DEE474639</RuleID>
  <StructuralMatch><![CDATA[
    FunctionCall call: call.function is
      [Function f: f.name == "clear" and f.enclosingClass.supers contains
        [Class c: c.name == "java.util.Map"]]
      and call.instance is [Expression outArgument: ]
  ]]></StructuralMatch>
  <Definition><![CDATA[
    TaintCleanse(outArgument, {})
  ]]></Definition>
</CharacterizationRule>
```

The previous taint characterization rule looks for a call to a method `clear()` defined in a class, one of the super-classes of which is `java.util.Map` class, and specifies that taint is completely removed from the map on which the method is called. The `<Definition>` element specifies that the rule represents a cleanse. The following table describes parameters of the TaintCleanse characterization rule type: `TaintCleanse(Expr, {TaintFlags})`.

Parameter	**Description**
Expr	Specifies an expression and optional access path (for example, `outArgument.foo`) that is cleansed of all taints or a set of taints when taint flags are removed, or receives taints when taint flags are added.
TaintFlags	(Optional) Specifies the taint flags to associate with taint introduced by the construct the rule matches. Specify the taint flags as a space-delimited list. Each flag must include a plus (+) or minus (-) prefix to indicate if it is added to or removed from the taint path. To remove all taint, omit the list of taint flags (specify empty braces `{}`).

Custom Dataflow Analyzer Rule Scenarios

This section provides examples of custom dataflow and taint characterization rules. Use these examples as a basis to write custom rules. Match your requirement with one of the examples, and tailor the rules to suit your software.

This section contains the following topics:

SQL Injection and Access Control 67
Persistent Cross-Site Scripting 72
Path Manipulation 78
Command Injection 81
Validation Construct Examples 85

SQL Injection and Access Control

This scenario highlights the rules that are necessary for the Dataflow Analyzer to detect access control vulnerabilities in the application. Because the analyzer detects SQL injection vulnerabilities with similar rules, this scenario also covers SQL injection vulnerabilities and corresponding detection rules.

The scenario describes the source code that includes an example of a SQL injection vulnerability. Then, the scenario demonstrates how the Dataflow Analyzer uses source, sink, and passthrough rules to identify this type of vulnerability.

This scenario highlights the following vulnerabilities:

- Access control—Without proper access control, execution of a SQL statement that contains a user-controlled primary key can enable an attacker to view unauthorized records.
- SQL Injection—Construction of a dynamic SQL statement with user input can enable an attacker to modify the statement intent or to execute arbitrary SQL commands.

This scenario highlights the following analysis and rule concepts:

- Conditionals
- Full cleanse function
- Neutral taint
- Paired sinks
- Partial cleanse functions
- Passthrough

Source Code

The application contains an access control vulnerability in its transaction service. The application enables users to provide their account identifier and retrieve their account details. An attacker can type any account identifier in the transaction service request, which causes the server to respond with the account details of the user.

The following RWO application example includes the JSP page that shows transaction details and has an access control vulnerability:

```
<% String accountNumber = request.getParameter("acctno");%>
...
<%
if ((accountNumber != null) && (accountNumber.length() > 0)) {
  Long account = Long.valueOf(accountNumber);
  List transactions = TransactionService.getTransactions(account);
  PrintWriter outputWriter = response.getWriter();
  outputWriter.println("<h1>Transactions reported from database
    for account <i>"+accountNumber+"</i></h1>");
  try {
    ...
  }
%>
```

The JSP calls `TransactionService.getTransactions()` with the account number as an argument to retrieve the account details. The transaction service queries the database for the associated transactions.

The following example shows how this method retrieves the accounts:

```
public static List getTransactions(Long acctno) throws Exception {
  Session session = ConnectionFactory.getInstance().getSession();
  String queryStr = "from Transaction transaction where
    transaction.acctno ='" + acctno + "'ORDER BY date DESC";
  if (ServletActionContext.getServletContext() != null) {
    ServletActionContext.getServletContext().log(queryStr);
  }
  Query query = session.createQuery(queryStr);
  List transactions = query.list();
  session.close();

  return transactions;
}
```

The method generates a dynamic SQL statement using the account number read from a request parameter. The code assumes that the account number only belongs to the current user. The code does not verify that the user has authorization to view the returned data.

This vulnerability type is closely related to the SQL injection vulnerability type. A SQL injection vulnerability exists when code appends an untrusted string, which can contain arbitrary characters. An attacker can input additional SQL code and change the entire meaning of the query.

The previous example does not contain a SQL injection vulnerability because the attack vector is of type `Long` and can only contain digits.

The following example shows an equivalent SQL injection vulnerability:

```
public static List getTransactions(String acctno) throws Exception {
  Session session = ConnectionFactory.getInstance().getSession();
  String queryStr = "from Transaction transaction where
    transaction.acctno ='" + acctno + "' ORDER BY date DESC";
  if (ServletActionContext.getServletContext() != null)
    ServletActionContext.getServletContext().log(queryStr);
  Query query = session.createQuery(queryStr);
  List transactions = query.list();
  session.close();
  return transactions;
}
```

Rules

In the first JSP page example, untrusted data enters the application through a method call to `getParameter()`.

The following example shows a dataflow source rule that models that call as a source of tainted data:

```
<DataflowSourceRule formatVersion="23.1" language="java">
  <RuleID>120E80B3-7EA2-4A18-82F2-0F7E53E97480</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Pattern>javax\.servlet</Pattern>
    </NamespaceName>
    <ClassName>
      <Value>ServletRequest</Value>
    </ClassName>
    <FunctionName>
      <Value>getParameter</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <OutArguments>return</OutArguments>
```

```
</DataflowSourceRule>
```

The dataflow source rule in the previous example matches the method `ServletRequest.getParameter()`. The `<OutArguments>` element indicates that the return value of the method is tainted. The lack of a `<TaintFlags>` element indicates that this is a general source of taint, which does not assign any taint flags.

The first JSP page example processes the incoming account number by converting it from a string type to a numeric type.

The following example shows the dataflow passthrough rule that enables the Dataflow Analyzer to follow taint from the `accountNumber` variable to the `account` variable:

```
<DataflowPassthroughRule formatVersion="23.1" language="java">
  <RuleID>73371DA9-10AD-4D13-823D-4BD0C9F2104F</RuleID>
  <TaintFlags>-XSS,+NUMBER</TaintFlags>
  <FunctionIdentifier>
    <NamespaceName>
      <Pattern>java\.lang</Pattern>
    </NamespaceName>
    <ClassName>
      <Value>Long</Value>
    </ClassName>
    <FunctionName>
      <Value>valueOf</Value>
    </FunctionName>
  </FunctionIdentifier>
  <InArguments>0</InArguments>
  <OutArguments>return</OutArguments>
</DataflowPassthroughRule>
```

The passthrough rule targets the `Long.valueOf()` method. The `<InArguments>` and `<OutArguments>` elements specify how tainted data flows through the method. When code calls the method with a tainted parameter, Fortify Static Code Analyzer considers that the return value from the call is tainted. The rule adds a specific taint flag NUMBER to the returned value to indicate the object is strictly numeric in nature. The rule removes any XSS taint flag from the returned value because it can no longer be used to conduct an XSS attack.

Eventually, the JSP page example executes the `TransactionService.getTransactions()` method, which then executes the `Session.createQuery()` method. The following example shows the sink rule that detects the access control vulnerability.

It checks that the `VALIDATED_ACCESS_CONTROL_DATABASE` taint flag is not present. If a validation function is later introduced to the flow of data in the source code, you can write a rule for the validation function that adds the `VALIDATED_ACCESS_CONTROL_DATABASE` taint flag. This ensures that Fortify Static Code Analyzer does not report a vulnerability for paths that flow through that function.

```
<DataflowSinkRule formatVersion="23.1" language="java">
  <RuleID>2B8502DE-E54E-4C59-AFC6-B6E3BCA67B3B</RuleID>
  <VulnCategory>Access Control</VulnCategory>
  <VulnSubcategory>Database</VulnSubcategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Sink>
    <InArguments>0</InArguments>
    <Conditional>
      <And>
        <TaintFlagSet taintFlag="NUMBER"/>
        <Not>
          <TaintFlagSet taintFlag="VALIDATED_ACCESS_CONTROL_DATABASE"/>
        </Not>
      </And>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
    <NamespaceName>
      <Pattern>net\.sf\.hibernate</Pattern>
    </NamespaceName>
    <ClassName>
      <Value>Session</Value>
    </ClassName>
    <FunctionName>
      <Value>createQuery</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
 </FunctionIdentifier>
</DataflowSinkRule>
```

Often, an access control sink rule is paired with a SQL injection rule. The method `Session.createQuery()` contains an access control vulnerability. You can convert an access control sink rule to a SQL injection sink rule.

The following example shows the equivalent SQL injection sink rule to the previous access control sink rule:

```
<DataflowSinkRule formatVersion="23.1" language="java">
  <RuleID>AE637178-A9D2-4BE6-A7B2-EEEA293B506F</RuleID>
  <VulnCategory>SQL Injection</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Sink>
    <InArguments>0</InArguments>
    <Conditional>
      <And>
        <Not>
          <TaintFlagSet taintFlag="NUMBER"/>
        </Not>
        <Not>
          <TaintFlagSet taintFlag="VALIDATED_SQL_INJECTION"/>
        </Not>
      </And>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>net.sf.hibernate</Value>
    </NamespaceName>
    <ClassName>
      <Value>Session</Value>
    </ClassName>
    <FunctionName>
      <Value>createQuery</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
</DataflowSinkRule>
```

Both rules target the first parameter of the same method. As opposed to the access control sink rule, the SQL injection sink rule must have an incoming parameter that is not a number. The Dataflow Analyzer checks for the presence of the neutral taint flag `VALIDATED_SQL_INJECTION`. If that taint is present, no vulnerability can occur and Fortify Static Code Analyzer does not report a vulnerability.

Persistent Cross-Site Scripting

This scenario highlights the rules that are necessary for Fortify to detect cross-site scripting (XSS) vulnerabilities in the application. The Dataflow Analyzer uses the source, sink, and passthrough rules

to identify this type of vulnerability.

The scenario demonstrates how an attacker can exploit a cross-site scripting vulnerability. It then shows how the Dataflow Analyzer uses source, sink, and passthrough rules to identify this type of vulnerability.

This scenario highlights the following vulnerability:

- Cross-site scripting—Sending unvalidated data to a web browser can result in the browser executing malicious code.

This scenario highlights the following analysis and rule concepts:

- General taint
- Neutral taint
- Passthrough
- Sink
- Source
- Specific taint

Source Code

The application contains a cross-site scripting vulnerability in the transaction page. An attacker can type malicious content into a transaction description. The victim receives a transaction notice. When the victim views the transaction details, the application delivers malicious content to the browser. The attacker can use this vector to execute JavaScript or other malicious content in the victim's browser. Any code that renders the details of a transaction is potentially vulnerable to this attack.

The following example shows the JSP page that renders these details for a given account number:

```
<%
  String accountNumber = request.getParameter("acctno");
  if ((accountNumber != null) && (accountNumber.length() > 0)) {
    Long account = Long.valueOf(accountNumber);
    List transactions = TransactionService.getTransactions(account);
    pageContext.getOut().println(
        "<h1>Transactions reported from database for account <i>"
        + accountNumber + "</i></h1>");

    try {
      for (Iterator it = transactions.iterator(); it.hasNext();) {
        Transaction transaction = (Transaction)it.next();
        String transactionDescription =
                "Transaction reported["+transaction.getId()+"]: "
                + "Account "+ transaction.getAcctno() + "; "
                + "Amount " + transaction.getAmount() + "; "
                + "Date " + transaction.getDate() + "; "
                + "Description " + transaction.getDescription();
        pageContext.getOut().flush();
        pageContext.getOut().println("<pre>"+transactionDescription+"</pre>");
```

```
    }
  ...
```

The code enumerates an account's transactions and prints the details of each transaction to the response stream. To do this, the JSP page calls `TransactionService.getTransactions()` to retrieve the transactions associated with the account specified by `acctno`.

The following example shows the source code that retrieves the data from the database:

```
public static List getTransactions(Long acctno) throws Exception {
  Session session = ConnectionFactory.getInstance().getSession();
  String queryStr = "from Transaction transaction where transaction.acctno ='"
                    + acctno
                    + "' ORDER BY date DESC";
  if (ServletActionContext.getServletContext() != null)
    ServletActionContext.getServletContext().log(queryStr);
  Query query = session.createQuery(queryStr);
  List transactions = query.list();
  session.close();

  return transactions;
}
```

This method calls `Query.list()` to retrieve the associated transactions from the database. The code in the previous example calls this method and does not validate the transactions list. This code contains a cross-site scripting vulnerability.

Rules

First, the JSP code calls a method to retrieve data from the database. A dataflow source rule models this method as a source of taint for Fortify Static Code Analyzer. Then, the JSP code calls methods to traverse the data. Fortify Static Code Analyzer uses dataflow passthrough rules to track the tainted data through these methods. Finally, the JSP code writes the data to the response stream. Fortify Static Code Analyzer uses dataflow sink rules to detect the final output.

The following example dataflow source rule models the call to `Query.list()` as a source of tainted data:

```
<DataflowSourceRule formatVersion="23.1" language="java">
  <RuleID>9ECA2C61-7625-41DB-967B-92768358C811</RuleID>
  <TaintFlags>+XSS,+DATABASE</TaintFlags>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>net.sf.hibernate</Value>
    </NamespaceName>
    <ClassName>
      <Value>Query</Value>
    </ClassName>
```

```
    <FunctionName>
      <Value>list</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <OutArguments>return</OutArguments>
</DataflowSourceRule>
```

The `<OutArguments>` element in the previous rule indicates that the return value of the method should be considered tainted. The rule also adds the taint flag XSS. This is a general taint flag that enables the Dataflow Analyzer to associate sources of data that might be used for a cross-site scripting attack with sinks that are potentially vulnerable to cross-site scripting.

The code in "Source Code" on page 73 iterates through the transaction list object returned from the call to `TransactionService.getTransactions()`. The Dataflow Analyzer applies the previous source rule with the result that the list object is considered tainted.

The following example shows a passthrough rule that enables the Dataflow Analyzer to propagate and track taint from the transactions list in "Source Code" on page 73 to the `it` iterator variable:

```
<DataflowPassthroughRule formatVersion="23.1" language="java">
  <RuleID>217417FB-7E50-41BA-ACB7-8159BD5211AC</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.util</Value>
    </NamespaceName>
    <ClassName>
      <Value>Collection</Value>
    </ClassName>
    <FunctionName>
      <Value>iterator</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <InArguments>this</InArguments>
  <OutArguments>return</OutArguments>
</DataflowPassthroughRule>
```

The in and out arguments specify how tainted data flows through the method. When the application code calls the method on a tainted target object (`this`), the Dataflow Analyzer propagates taint to the return value.

The following example shows the passthrough rule that enables the analyzer to understand how taint is returned from the iterator object on the call to `Iterator.next()`:

```
<DataflowPassthroughRule formatVersion="23.1" language="java">
  <RuleID>D56C1363-C303-4AAB-99A9-98075D0FEB80</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Pattern>java\.util</Pattern>
    </NamespaceName>
    <ClassName>
      <Value>Iterator</Value>
    </ClassName>
    <FunctionName>
      <Value>next</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <InArguments>this</InArguments>
  <OutArguments>return</OutArguments>
</DataflowPassthroughRule>
```

Finally, the JSP code in the following example constructs a transaction description and displays it to the user using the following code (repeated for convenience):

```
...
  String transactionDescription = "Transaction reported["+transaction.getId()+"]: "
                                + "Account "+ transaction.getAcctno() + "; "
                                + "Amount " + transaction.getAmount() + "; "
                                + "Date " + transaction.getDate() + "; "
                                + "Description " + transaction.getDescription();
      outputWriter.flush();
      outputWriter.println("<pre>"+transactionDescription+"</pre>");
  ...
```

Fortify Static Code Analyzer has access to all the source code for the transaction object, which means the Dataflow Analyzer can automatically track taint through the object's getter methods. This means the Dataflow Analyzer can successfully track taint from the transaction object to the `transactionDescription` string without the need for additional rules.

The following example shows the dataflow sink rule the Dataflow Analyzer uses to identify the XSS vulnerability. This rule marks the `JspWriter.println()` function as a sink. The rule checks that the XSS flag is present, and that the `VALIDATED_CROSS_SITE_SCRIPTING` flag is not. A developer might later introduce a validation function that verifies the contents of the data. Fortify Static Code Analyzer requires a new cleanse rule for that validation function that adds the `VALIDATED_CROSS_SITE_SCRIPTING` taint flag to the data. This ensures that Fortify Static Code Analyzer does not report a vulnerability for paths that flow through that function.

```
<DataflowSinkRule formatVersion="23.1" language="java">
  <RuleID>5F0C1BA2-3F30-483F-9232-9DB09442801E</RuleID>
  <VulnCategory>Cross-Site Scripting</VulnCategory>
  <VulnSubcategory>Persistent</VulnSubcategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Sink>
    <InArguments>0</InArguments>
    <Conditional>
      <And>
        <TaintFlagSet taintFlag="XSS"/>
        <Not>
          <TaintFlagSet
            taintFlag="VALIDATED_CROSS_SITE_SCRIPTING_PERSISTENT"/>
        </Not>
      </And>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>javax.servlet.jsp</Value>
    </NamespaceName>
    <ClassName>
      <Value>JspWriter</Value>
    </ClassName>
    <FunctionName>
      <Value>println</Value>
    </FunctionName>
    <Parameters>
      <ParamType>java.lang.String</ParamType>
      <WildCard min="0" max="2"/>
    </Parameters>
    <ApplyTo implements="true" overrides="true" extends="true"/>
  </FunctionIdentifier>
</DataflowSinkRule>
```

The `<Parameters>` element in the function identifier ensures that this rule only matches versions of the `JspWriter.println()` function, which takes a string as the first parameter. The `<Sink>` element specifies that the first parameter is the parameter that is sensitive to taint, and specifies the set of taint flag constraints in the `<Conditional>` element.

Path Manipulation

This scenario highlights the rules necessary for the Fortify Static Code Analyzer Dataflow Analyzer to detect path manipulation vulnerabilities. The scenario demonstrates how an attacker can exploit a path manipulation vulnerability. It then shows how the Dataflow Analyzer uses TaintEntrypoint and sink rules to identify a path manipulation vulnerability.

This scenario highlights the following vulnerability:

- Path manipulation—This type of vulnerability enables an attacker input to control the paths used in file system operations. An attacker can exploit this type of vulnerability to access or modify otherwise-protected system resources.

This scenario highlights the following analysis and rule concepts:

- Conditional
- Constructor token
- TaintEntrypoint
- General taint
- Input argument
- Annotation
- Neutral taint
- Parameter signature
- Sink

Source Code

The application in this scenario is written using Spring MVC framework and contains a path manipulation vulnerability in its banner advertisement web service. The web service enables affiliates to provide an identifier and retrieve a JPEG image that contains an advertisement. An attacker can enter a malicious identifier in the web request, which causes the server to respond to the request with the contents of sensitive files.

The following code retrieves banner ads for affiliates:

```
@Controller
@RequestMapping("/ad")
public class BannerAdController {
  static private String baseDirectory = "/images/bannerAds";

  @RequestMapping("/retrieveBannerAd")
  public @ResponseBody File retrieveBannerAd(@RequestParam("clientAd")
    String clientAd) {
    // Retrieve banner with given guid
    File targetFile = new File(baseDirectory + clientAd);
    return targetFile;
  }
...
}
```

When an affiliate executes a web service call to the method `BannerAdController.retreiveBannerAd()`, the application returns the image file associated with the affiliate identifier `clientAd`.

The code assumes that the incoming affiliate identifier specified only a single file name, but if an attacker provides the identifier `'../../../../../windows/system.ini'`, the server retrieves the file `/images/bannerAds/../../../../../windows/system.ini`. On most systems, this is equivalent to `/windows/system.ini`.

Rules

In the example, untrusted data enters through the web service entry point and is passed to a file constructor. The analyzer models that entry point as a source of taint using a TaintEntrypoint characterization rule. The following example shows the rule that models this method as a source of taint:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>AE31740B-140B-4338-BE4F-33D7F05CC840</RuleID>
  <StructuralMatch><![CDATA[
    Variable p: p.enclosingFunction is
      [Function f: f.parameters contains p]
      and p.annotations contains
      [Annotation: type.name ==
      "org.springframework.web.bind.annotation.RequestParam"]
  ]]></StructuralMatch>
  <Definition><![CDATA[
    foreach p {TaintEntrypoint(p, {+WEB +XSS}) }
  ]]></Definition>
```

```
</CharacterizationRule>
```

The entrypoint rule in the previous example matches the method `BannerAdController.retrieveBannerAd()` whose parameter is annotated with the `org.springframework.web.bind.annotation.RequestParam` annotation. The `foreach` block indicates interest in all assignments that satisfy the predicate rather than an arbitrary one. In this rule, taint flags are added to all variables that satisfy the predicate instead of a single variable from the set that satisfies the predicate.

The following example describes the dataflow sink rule that matches the corresponding constructor:

```
<DataflowSinkRule formatVersion="23.1" language="java">
  <RuleID>98558CD1-708D-48E8-8C68-F93481CB15A9</RuleID>
  <VulnCategory>Path Manipulation</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description ref="desc.dataflow.java.path_manipulation"/>
  <Sink>
    <InArguments>0</InArguments>
    <Conditional>
      <Not>
        <TaintFlagSet taintFlag="VALIDATED_PATH_MANIPULATION"/>
      </Not>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.io</Value>
    </NamespaceName>
    <ClassName>
      <Value>File</Value>
    </ClassName>
    <FunctionName>
      <Pattern>init\^</Pattern>
    </FunctionName>
    <Parameters>
      <ParamType>java.lang.String</ParamType>
    </Parameters>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
</DataflowSinkRule>
```

The dataflow sink rule uses the special keyword `init^` to match the `File.File()` constructor. This keyword is reserved for class constructors and enables rules to match across inheritance relationships.

When taint reaches the sink, the `<Conditional>` element ensures that no vulnerability is reported if the neutral taint flag `VALIDATED_PATH_MANIPULATION` is also present. This taint flag indicates that the data has been correctly validated beforehand. You can write a separate cleanse or passthrough rule to add the neutral taint flag `VALIDATED_PATH_MANIPULATION` to data that passes through the appropriate validation method.

Command Injection

This scenario highlights rules that are necessary for the Dataflow Analyzer to detect command injection vulnerabilities. The scenario demonstrates how an attacker can exploit a command injection vulnerability. It then illustrates how Dataflow Analyzer uses characterization TaintSource, sink, and passthrough rules to identify this type of vulnerability.

This section highlights the following vulnerability:

- Command injection—Executing commands from an untrusted source or in an untrusted environment can cause an application to execute malicious commands on behalf of an attacker.

This scenario highlights the following analysis and rule concepts:

- Input arguments
- Output arguments
- Passthrough
- Sink
- TaintSource

Source Code

The application contains a command injection vulnerability in its messaging service. An attacker can formulate an email using the messaging service. The attacker enters malicious commands into a to-address. Then the attacker submits the message to the server for processing. When the victim receives the message, the server executes the embedded commands. Code that formulates emails using an internal messaging class is vulnerable to this attack.

The following example shows a JSP page that uses this class to broadcast alert messages:

```
<% GlobalURLObject globalURLObject = applicationContext.getGlobalURLObject
();
  String alertMessage = globalURLObject.message;
   int messageCount = 0;

   if ((alertMessage != null) && (alertMessage.length() > 0)) {
     SendMessage msgClass = new SendMessage();
     String specifiedUsers = globalURLObject.users;
     if ((specifiedUsers != null) && (specifiedUsers.length() > 0)) {

       String[] users = specifiedUsers.split(";");
```

```
        for (int index=0; index < users.length; index++) {
          String emailAddress = users[index];

          msgClass.setTo(emailAddress);
          msgClass.setSubject("Technical Difficulties");

          String processedMessage = alertMessage.replaceAll("<code1>"
            "The system is currently experiencing technical
            difficulties.");

           msgClass.setBody(processedMessage);
           msgClass.setSeverity("Highest");
           msgClass.execute();
           messageCount++;
          }
        ...
```

The JSP does some superficial processing of the message and then calls `SendMessage.execute()`.

The following example shows how this method handles the processed message:

```
public void execute() {
  if (isInvalidEmail(this.to)) return INPUT;

  String[] cmd = getMailCommand();
  String message = sendMail(cmd);

  addActionMessage(message);
}
```

The `SendMessage.execute()` method calls `SendMessage.getMailCommand()` to generate a command string that is executed to send the email.

The following example shows how the command string is generated:

```
public String[] getMailCommand() {
  ...
  cmd[2] = java + " -cp "+ cp +
    " com.fortify.samples.riches.legacy.mail.SendMail \"
    " + subject + "\" \"" + severity + "\" \"" + body + "\"
    " + to;

  return cmd;
}
```

This code assumes that the email message fields do not contain `|`, `;`, or `&` symbols. These symbols represent command string delimiters on different platforms. You can include these delimiters in a command string to execute multiple commands in the same string. For example, an attacker can provide the message body `'" & dir C:\ > c:\files.txt &'`. The JSP code eventually calls the `SendMessage.execute()` method to generate and execute a shell command string based on the mail command. This method calls the `SendMessage.sendMail()` method to execute the command string:

```
public String sendMail(String[] cmd) {
  Runtime rt = Runtime.getRuntime();
  //call "legacy" mail program
  Process proc = null;
  StringBuilder message = new StringBuilder();
  try {
    proc = rt.exec(cmd);
    ...
```

If an attacker submits the sample message body, the shell executes the original command and the additional commands included in the sample message body.

Rules

Tainted data enters the JSP code through the access of the fields of the `GlobalURLObject` object. The "Source Code" on page 81 illustrates this access on lines 2 and 7.

The following example taint characterization rule causes Fortify Static Code Analyzer to model that field access as a source of tainted data:

```
<CharacterizationRule formatVersion="23.1" language="java">
  <RuleID>471ABD87-96E0-4327-ACBB-D74C9B767155</RuleID>
  <StructuralMatch><![CDATA[
    FieldAccess fa0: fa0.instance is
    [FieldAccess fa: fa.field.enclosingClass.supers contains
    [Class c: c.name == "com.mypackage.GlobalURLObject"]
    and not fa in [AssignmentStatement: lhs.location is
    [Location l: l.transitiveBase === fa.transitiveBase]]]
  ]]></StructuralMatch>
  <Definition><![CDATA[
    TaintSource(fa0, {+XSS +WEB})
  ]]></Definition>
</CharacterizationRule>
```

The rule taints any field accessed from the object of type `GlobalURLObject` that is not on the left-hand side of the assignment statement with WEB taint to indicate that the object contains data that originated from the web. Traditionally, Fortify associates WEB taint with XSS taint because objects coming from a web source might also contain JavaScript. Other rules use this extra taint to identify

cross-site scripting vulnerabilities and are not directly applicable to command injection vulnerability detection. Note that in this example characterization rule, a `foreach` block is not required in the `<Definition>` element because the TaintSource is written about `fa0` associated with the main predicate, which instructs the rule to match only one code structure.

The JSP code processes the incoming email message by calling the `String.replaceAll()` method to replace identifier keys with message text.

The following example shows a dataflow passthrough rule that enables Fortify Static Code Analyzer to follow taint from the `alertMessage` variable to the `processedMessage` variable:

```
<DataflowPassthroughRule formatVersion="23.1" language="java">
  <RuleID>B1D159AE-EE88-4760-A112-8BFC5F774DE3</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.lang</Value>
    </NamespaceName>
    <ClassName>
      <Value>String</Value>
    </ClassName>
    <FunctionName>
      <Value>replaceAll</Value>
    </FunctionName>
    <ApplyTo implements=true" overrides="true" extends="true"/>
  </FunctionIdentifier>
  <InArguments>this</InArguments>
  <OutArguments>return</OutArguments>
</DataflowPassthroughRule>
```

The following example dataflow sink rule detects the command injection vulnerability. This rule marks the Java `Runtime.exec()` method as a sink. It verifies that the `VALIDATED_COMMAND_INJECTION` taint flag is not present. To add a validation function to validate the contents of the data, the developer can write a rule for the validation function that adds the `VALIDATED_COMMAND_INJECTION` taint flag to the data objects. This ensures that Fortify Static Code Analyzer does not report a vulnerability for paths that flow through that function.

```
<DataflowSinkRule formatVersion="23.1" language="java">
  <RuleID>E6E0AC3D-1C7B-48B1-B80D-2AC4619B0D81</RuleID>
  <VulnCategory>Command Injection</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Sink>
    <InArguments>0...</InArguments>
    <Conditional>
      <Not>
        <TaintFlagSet taintFlag="VALIDATED_COMMAND_INJECTION"/>
```

```
      </Not>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.lang</Value>
    </NamespaceName>
    <ClassName>
      <Value>Runtime</Value>
    </ClassName>
    <FunctionName>
      <Value>exec</Value>
    </FunctionName>
    <ApplyTo implements="true" overrides="true" extends="true"/>
  </FunctionIdentifier>
</DataflowSinkRule>
```

Validation Construct Examples

The following example shows source code that validates against SQL injections in Java:

```
package com.company.package;
public final class MyValidationClass {
  ...
  public static String cleanseSQLString(String arg) {
    return arg.replaceAll("[^a-zA-Z\\s]", "");
  }
  ...
}
```

Rules

If the function is part of an external library and its source is not included in the scan, write a passthrough rule with the appropriate taint flag modifications. The following example passthrough rule describes to the Dataflow Analyzer that tainted data does flow through the function, but that validation is performed in the process:

```
<DataflowPassthroughRule formatVersion="23.1" language="java">
  <RuleID>98DE1262-6A55-4C74-8A24-497FD8198421</RuleID>
  <TaintFlags>+VALIDATED_SQL_INJECTION</TaintFlags>
  <FunctionIdentifier>
    <NamespaceName>
```

```
      <Value>com.company.package</Value>
    </NamespaceName>
    <ClassName>
      <Value>MyValidationClass</Value>
    </ClassName>
    <FunctionName>
      <Value>cleanseSQLString</Value>
    </FunctionName>
    <ApplyTo implements="true" overrides="true" extends="true"/>
  </FunctionIdentifier>
  <InArguments>0</InArguments>
  <OutArguments>return</OutArguments>
</DataflowPassthroughRule>
```

The following example sink rule is also needed to check for this taint flag:

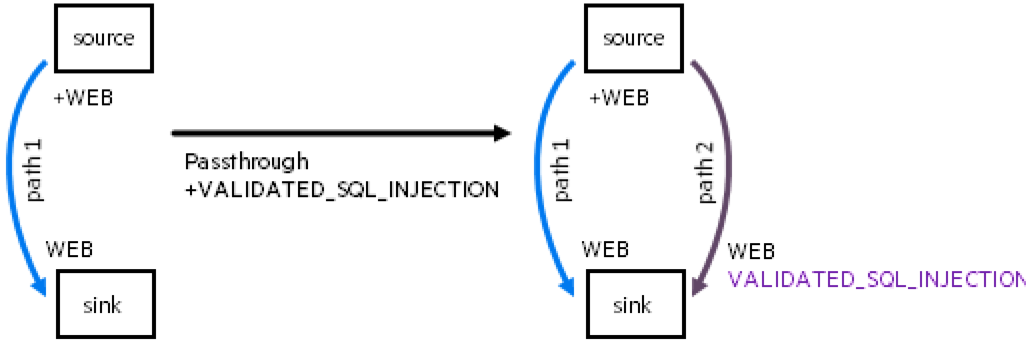
```
<DataflowSinkRule formatVersion="23.1" language="java">
  <RuleID>B5808D41-BA35-4D2F-89BF-4273BCA763E4</RuleID>
  <VulnCategory>SQL Injection</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Sink>
    <InArguments>0</InArguments>
    <Conditional>
      <And>
        <Not>
          <TaintFlagSet taintFlag="NUMBER"/>
        </Not>
        <Not>
          <TaintFlagSet taintFlag="VALIDATED_SQL_INJECTION"/>
        </Not>
      </And>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>com.company.package</Value>
    </NamespaceName>
    <ClassName>
      <Value>Connection</Value>
    </ClassName>
    <FunctionName>
```

```
      <Value>executeQuery</Value>
    </FunctionName>
    <ApplyTo implements="true" overrides="true" extends="true"/>
  </FunctionIdentifier>
</DataflowSinkRule>
```

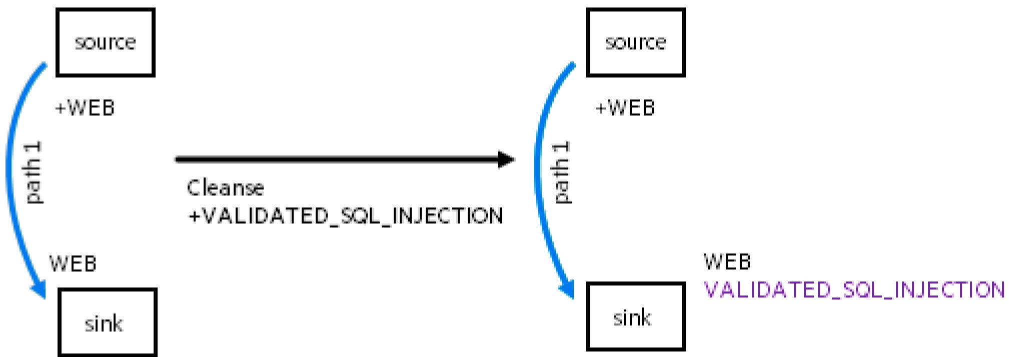
If the function is part of the source code being scanned, a cleanse rule is more appropriate. Because the Dataflow Analyzer already derived the passthrough behavior of the function by looking at its code, you only need to describe the taint flags that the analyzer adds or removes with a cleanse rule as shown in the following example:

```
<DataflowCleanseRule formatVersion="23.1" language="java">
  <RuleID>E01B88BE-F6B2-4EF9-BE43-035A008FE1C0</RuleID>
  <TaintFlags>+VALIDATED_SQL_INJECTION</TaintFlags>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>com.company.package</Value>
    </NamespaceName>
    <ClassName>
      <Value>MyValidationClass</Value>
    </ClassName>
    <FunctionName>
      <Value>cleanseSQLString</Value>
    </FunctionName>
    <ApplyTo implements="true" overrides="true" extends="true"/>
  </FunctionIdentifier>
  <OutArguments>return</OutArguments>
</DataflowCleanseRule>
```

Do this with a cleanse rule so that the analyzer applies the cleanse rule to the taint path *after* the derived passthrough. A passthrough rule is applied in parallel, creating a separate taint path and would not have the desired effect. The following diagram illustrates this concept:



Passthrough Rule Produces Additional Taint Paths



Cleanse Rule Augments Existing Taint Path

The previous rules work for the following usage of the validation function:

```
package com.company.package;
public class MyClass {
  ...
  public void myMethod(HttpRequest request, Connection connection) {
    String query = request.getParameter("query");
    connection.executeQuery(MyValidationClass.cleanseSQLString(query));
  }
  ...
}
```

Many developers define their validation functions to return true or false depending on the validity of the input. This can cause challenges for code maintenance because whether the data is validated or not is unclear. This could lead to accidentally introduced vulnerabilities. If the validation function is

defined this way, you must modify the previous usage and the rules. The following example shows a redefined validation function:

```
package com.company.package;
...
public final class MyValidationClass {
  ...
  public static boolean isBadQuery(String arg) {
    return arg.matches("[^a-zA-Z\\s]");
  }
  ...
}
```

For the previous case, you can use the validation function in the following way:

```
package com.company.package;
public class MyClass {
  ...
  public void myMethod(HttpRequest request, Connection connection)
   throw BadInputException {
    String query = request.getParameter("query");
    if (!MyValidationClass.isBadQuery(query)) {
      connection.executeQuery(query);
    }
    else {
      throw BadInputException("Invalid query.");
    }
  }
  ...
}
```

Fortify Static Code Analyzer does not perform path-sensitive dataflow analysis because of performance implications, and therefore can generate false positives. However, it is possible to write a custom TaintSink characterization rule to handle this type of scenario. Even though you can write the if statement conditional in different ways, the following taint characterization rule is only valid for the previous example.

```
<CharacterizationRule formatVersion="16.20" language="java">
  <RuleID>D06F30A5-DB9C-46A2-965C-A74FBC23501C</RuleID>
  <VulnCategory>SQL Injection</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <StructuralMatch><![CDATA[
    FunctionCall call: call.function is
```

```
    [Function f: f.name == "executeQuery" and f.enclosingClass.supers
    contains [Class c: c.name == "com.company.package.Connection"]]
    and call.arguments[0] is [Expression inArgument: ]
    and not call.enclosingFunction contains
    [IfStatement ifStmt: ifStmt.ifBlock contains
    [FunctionCall fc2: fc2 == call] and ifStmt.expression is
    [Operation onot: onot.unary and onot.lhs is
    [FunctionCall fc: fc.function is [Function valf: valf.name == "isBad"
    and valf.enclosingClass.supers contains
    [Class valc: valc.name == "com.company.package.MyValidationClass"]]]]]
  ]]></StructuralMatch>
  <Definition><![CDATA[
    TaintSink(inArgument, [!NUMBER && !VALIDATED_SQL_INJECTION])
  ]]></Definition>
</CharacterizationRule>
```

The previous taint characterization rule matches calls to `com.company.package.Connection.executeQuery()` where non-numeric and not validated taint reaches its first argument. Fortify Static Code Analyzer does not report the vulnerability if the call happens inside an `if` block of an `if` statement that checks that the call to `com.company.package.MyValidationClass.isBad()` function returns `false`.

Chapter 5: Control Flow Analyzer Rules

This section contains the following topics:

Control Flow Analyzer and Custom Rules 91

Control Flow Analyzer and Custom Rule Concepts 93

XML Representation of Control Flow Analyzer Rules 94

Custom Control Flow Rule Scenarios 96

Control Flow Analyzer and Custom Rules

The Control Flow Analyzer finds security issues in programs that have insecure sequences of operations. This enables Fortify Static Code Analyzer to identify many types of security problems.

The Control Flow Analyzer models each security property as a state machine. Each state machine has the following states:

- Initial state
- Any number of internal states
- One or more error states

The state machine is in the initial state at the beginning of a function. The Control Flow Analyzer reports a vulnerability when a state machine enters an error state.

The states in the state machine are connected by transitions. A transition leads from one state (the source state) to another state (the destination state) and has one or more associate rule patterns. Rule patterns specify program constructs. The state of a state machine changes from source to destination when one of the transition's rule patterns matches a statement that the Control Flow Analyzer is analyzing.

A state can have any number of transitions leading out of or into it. The Control Flow Analyzer checks the transitions leading out of a state one at a time in the order in which they appear in the state machine definition. The Control flow Analyzer executes the first statement that matches a statement. The Control Flow Analyzer ignores any other transition out of the same state.

You can use this to limit the number of functions that the program can call in a given context: the state representing that context would have a transition to a safe state (possibly itself) if the program calls an allowed function, and a transition to an error state if the program calls any function.

The Control Flow Analyzer operates interprocedurally, so if one function calls a second function, and a state transition occurs inside that second function, the analyzer updates the state in the first (calling) function as well.

The Control Flow Analyzer applies the transitions to the code in the order they are specified in the rule, so the order is important. For example, you might want to have a transition into a safe state prior

to a transition into an error state, because otherwise the analyzer might always transition into an error state without ever providing an opportunity to transition into a safe state.

The following example program uses a locking API. The API contract states that a function that acquires the lock must release it before returning. In some cases, the sample program does not release the lock before returning.

The following sample program does not always release the lock before returning:

```
function readFile(File file) {
  Lock fileLock = getLock(file);
  if (!isReadable(file)) {
    return;
  }
  doRead(file);
  releaseLock(fileLock);
  return;
}
```

The contract for the locking API is described as a state machine.

The following table shows the states and transitions of the state machine provided in the following state machine control flow rule.

Source State	**Destination State**	**Program Construct That Causes Transition**
Unlocked (start state)	Locked	Call to `getLock()`
Locked	Released	Call to `releaseLock()`
Locked	Leaked (error state)	Function ends

The following control flow rule encodes this state machine:

```
1  state Unlocked (start);
2  state Locked;
3  state Released;
4  state Leaked (error);
5  var lock;
6  Unlocked -> Locked { lock = getLock(...) }
7  Locked -> Released { releaseLock(lock) }
8  Locked -> Leaked { #end_function() }
```

When the Control Flow Analyzer uses this rule to check the previous example function, the state machine is initially in the `Unlocked` state. When the program acquires the lock on line 2, the state machine transitions to the `Locked` state, and the rule variable maps the rule variable `lock` to the program variable `fileLock` (see below for more discussion of rule variables). At the branch on line 3,

the Control Flow Analyzer copies the state machine. One copy runs in the "true" branch of the conditional, and the other copy runs in the "false" branch.

Both copies are initially in the `Locked` state. When the copy running on the "true" branch encounters the return statement on line 4, it transitions to the `Leaked` state. Because `Leaked` is an error state, the Control Flow Analyzer reports a vulnerability. Meanwhile, the copy of the machine running on the "false" branch encounters the program releasing the lock on line 7 and transitions to the `Released` state. When this copy encounters the return statement on line 8, it does not transition to the error state because there is no transition from `Released` to `Leaked`.

Control Flow Analyzer and Custom Rule Concepts

Rule Pattern

A rule pattern specifies the program constructs that cause a state transition to occur. The rule patterns are the parts enclosed in braces.

Rule Variable

A rule variable is a part of a rule pattern that is a placeholder for an actual program value. Rule variables tie together values used in different rule patterns. In the control flow rule on page 92, the rule variable "lock" ties together the return value from `getLock()` and the parameter to `releaseLock()`. Without this rule variable, the state machine transitions to the Released state whenever any lock is released, even if some locks in the function are still unreleased.

Rule Binding

A rule binding is a mapping between a rule variable and a program value (or a set of program values). In the control flow rule shown in "Control Flow Analyzer and Custom Rules" on page 91, the analyzer creates a rule binding that ties the rule variable `lock` to the `fileLock`, which is a local variable. When the analyzer evaluates other rule patterns that use the rule variable `lock`, the pattern only matches if the rule binding for `lock` matches the program value used in its place.

Rule variables and rule bindings enable the Control Flow Analyzer to model the behavior of specific objects in the program, rather than just the global state of the program.

The following is an example:

```
1  function useTwoLocks() {
2      Lock lock1 = getLock();
3      Lock lock2 = getLock();
4      releaseLock(lock1);
5      return;
6  }
```

This function acquires two locks, but only releases one of them. Without rule variables, the Control Flow Analyzer is not able to detect this error, because it sees only that `releaseLock()` is called, without correlating the calls to `getLock()` and `releaseLock()`. With the rule variables in the previous code sample, however, the analyzer correlates these two calls.

When the analyzer encounters the first `getLock()` call on line 2, it creates a rule binding between the rule variable `lock` and the program variable `lock1`, and moves to the `Locked` state. It also creates a copy of the state machine that remains in the `Unlocked` state. The analyzer then encounters the second call to `getLock()`.

The copy of the state machine that is in the `Locked` state ignores this call, because it does not match any transitions out of the `Locked` state. The copy that is in the `Unlocked` state, however, does match this call. The analyzer creates a second rule binding that maps the rule variable `lock` to the program variable `lock2`, and this second copy of the state machine changes to the `Locked` state.

In the previous code sample, the first state machine transitions to the `Released` state, while the second machine remains in the `Locked` state. At the return statement, the second machine changes to the `Leaked` state, and the analyzer reports an issue.

XML Representation of Control Flow Analyzer Rules

The XML representation of a control flow rule is based on the representation of a vulnerability-causing rule. In addition to the elements common to all such rules, there are some elements that are specific to control flow rules or that are used differently in control flow rules.

The following rule shows a control flow rule example with a primary state rule:

```
<ControlflowRule formatVersion="23.1" language="java">
  <RuleID>6FC83768-C5A0-0E26-044B-59E8A1EBA0BA</RuleID>
  <VulnCategory>Resource Leak</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Limit>
    <FunctionIdentifier>
      <FunctionName>
        <Value>ProcessRequest</Value>
      </FunctionName>
    </FunctionIdentifier>
  </Limit>
  <FunctionCallIdentifier id="allocate">
    <FunctionIdentifier>
      <FunctionName>
        <Value>AllocateResource</Value>
      </FunctionName>
    </FunctionIdentifier>
    <Conditional>
```

```
      <Not><ConstantEq argument="0" value="0"/></Not>
    </Conditional>
  </FunctionCallIdentifier>
  <FunctionIdentifier id="deallocate">
    <FunctionName>
      <Value>ReleaseResource</Value>
    </FunctionName>
  </FunctionIdentifier>
  <PrimaryState>Allocated</PrimaryState>
  <Definition><![CDATA[
      state Unallocated (start);
      state Allocated;
      state Deallocated;
      state Leaked;var resource;
      Unallocated -> Allocated { resource = allocate(…) }
      Allocated -> Deallocated { deallocate(resource) }
      Allocated -> Leaked { #end_scope(resource) }
  ]]></Definition>
</ControlflowRule>
```

The following table describes the XML elements introduced in the previous control flow rule example.

Element	Description
FunctionIdentifier	Like other rule types, control flow rules use `<FunctionIdentifier>` elements to identify functions. Unlike most other rule types, control flow rules can contain multiple function identifiers. This is because a state machine defined by a control flow rule can refer to multiple functions. This element has the following optional attribute: • `id`—Specifies the name by which you can use the function identifier within the rule definitions
FunctionCallIdentifier	Function call identifiers combine `<FunctionIdentifier>` and `<Conditional>` elements to match specific calls to a function. This element has the following optional attribute: • `id`—Specifies the name by which you can use the function identifier within the rule definitions **Note:** The `id` attribute of the `<FunctionIdentifier>` inside the `<FunctionCallIdentifier>` is not used.

Element	Description
Limit	Control flow rules should only check specific properties in certain functions. For example, a control flow rule could check that every function called `ProcessRequest` must call the `CheckCredentials` function before calling the function `AccessPrivateData`. You can prevent this rule from running on methods other than `ProcessRequest` by adding a `<Limit>` element to the rule definition. In this case, the `<Limit>` element contains one or more `<FunctionIdentifier>` elements. The rule only evaluates functions that match one of these function identifiers. A rule with no `<Limit>` element runs on all functions.
PrimaryState	You specify the primary state by putting the state name inside the `<PrimaryState>` element. If the rule does not explicitly specify a primary state, the error state is primary. Control flow state machines contain multiple states. You can designate one of these states as the primary. When you view an issue, the trace element that displays first is the first one that transitioned into its primary state. If several control flow traces transition into their primary state at the same program location, the Control Flow Analyzer groups these traces into one control flow issue. This issue contains multiple traces.
Definition	The control flow state machine definition is enclosed in the `<Definition>` element. You can enclose the contents of this element in a CDATA section to avoid the need to escape XML special characters in the state machine definition.

Custom Control Flow Rule Scenarios

This section provides examples of custom control flow rules. You can use these examples as a basis to write custom rules. Match your requirement with one of the examples, and tailor the rules to suit your software.

This section contains the following topics:

Resource Leak 97
Null Pointer Check 103

Resource Leak

This scenario highlights the rules that are necessary for the Control Flow Analyzer to detect resource leaks. This scenario demonstrates how an attacker can exploit a resource leak vulnerability. Then, it shows how the Control Flow Analyzer uses control flow rules to identify this type of vulnerability.

This scenario highlights the following vulnerability:

- Poor code quality: resource leaks—Program can fail to release a system resource

This scenario highlights the following analysis and rule concepts:

- Control flow rules
- Finite state machines
- Non-returning rules
- `#end_scope` operator
- `#ifblock` operator

Source Code

An attacker exploits a resource leak vulnerability as a logical denial-of-service attack. Imagine code that uses a scarce system resource and contains a resource leak. The attacker depletes the associated resource by repeatedly executing the code. This leads to resource depletion that prevents legitimate users from using the service.

The following code contains many resource leaks. It illustrates how the application typically sets up a connection to its database and performs some query for necessary data. This method retrieves detailed data about a list of roles and reports the ones that have administrative privileges:

```
public static void debugAdminRoles(List roles) throws Exception {
  boolean auth = false;
  Connection conn = null;
  Statement statement = null;
  ResultSet rs = null;

  try {
    conn = ConnFactory.getInstance().getConnection();
    statement = conn.createStatement();

    for (int index=0; index < roles.size(); index++) {
      int roleid = ((Integer)roles.get(index)).intValue();

      rs = statement.executeQuery
           ("SELECT rolename FROM auth WHERE roleid = " + roleid);
      rs.next();
```

```
      if (rs !=null && rs.getString("rolename").equals("admin")) {
        System.err.println("Roleid: "+roleid+" is an admin");
        rs.close();
        rs = null;
      }
    }
  }catch(Exception e) {
    if (rs != null) {
      rs.close();
      rs = null;
    }
    throw e;
  }
  finally {
    System.err.println("Terminating here temporarily");
    System.exit(-1);

    if (statement != null) {
      statement.close();
      statement = null;
    }
  }
}
```

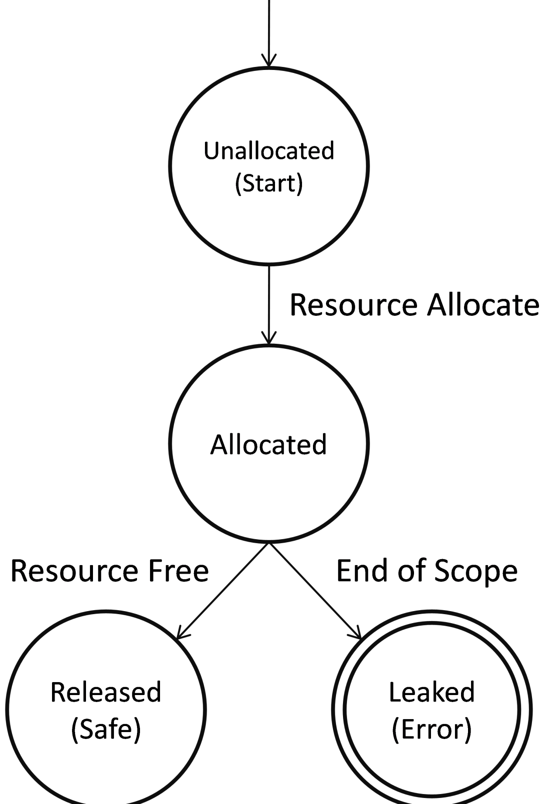
First, the code creates a connection object based on an existing Hibernate database connection. Then, the code creates a statement object using the new connection object. Finally, the code executes the statement object's query method that returns a result-set object. The code must then free all the associated resources by closing the connection, statement, and result-set objects.

The code fails to close these objects under all conditions. The code never closes the connection object under any conditions. Also, the code attempts to close the statement object within the finally block. However, the code executes the `System.exit()` method first and the `Statement.close()` method is never reached. Finally, the code does not close the result-set object when the role is not an administrator and an exception does not occur.

Rules

The Control Flow Analyzer uses an object's finite state machine (FSM) to identify unsafe sequences of operations that should not be performed on that object.

The following illustration describes the dynamically allocated/deallocated object states of an object:



First, the analyzer allocates a separate FSM for each object. Then, the analyzer sets the object's initial state as unallocated before code allocates the object. After code allocates an object, the analyzer updates the object's FSM state to the allocated state. Then, the analyzer examines all codepaths that are within the object's scope.

The analyzer encounters a codepath where the code calls the object's `close()` method. In such a case, the analyzer updates the object's FSM state to the safe released state. Eventually, the object falls out of scope. This codepath correctly releases the resource and no vulnerability exists. The analyzer does not report a vulnerability for this path because the object falls out of scope in a safe state.

The analyzer encounters codepaths where the object falls out-of-scope and the code has not previously called the object's `close()` method. In such a case, the analyzer updates the object's FSM state to the unsafe leaked state. The analyzer reports the vulnerability because the analyzer has explicitly set the object's FSM state to an unsafe state.

The following rule describes the FSM model that applies for the safe and unsafe allocation of the `Connection`, `Statement`, or `ResultSet` objects:

```
<ControlflowRule formatVersion="23.1" language="java">
  <RuleID>84C341ED-9917-4901-A792-C93E6D72C5A6</RuleID>
  <VulnCategory>Unreleased Resource</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
```

```
<Description/>
<FunctionIdentifier id="resource1">
  <NamespaceName>
    <Value>javax.sql</Value>
  </NamespaceName>
  <ClassName>
    <Value>DataSource</Value>
  </ClassName>
  <FunctionName>
    <Value>getConnection</Value>
  </FunctionName>
  <ApplyTo overrides="true" implements="true" extends="true"/>
</FunctionIdentifier>
<FunctionIdentifier id="resource2">
  <NamespaceName>
    <Value>java.sql</Value>
  </NamespaceName>
  <ClassName>
    <Value>Connection</Value>
  </ClassName>
  <FunctionName>
    <Value>createStatement</Value>
  </FunctionName>
  <ApplyTo overrides="true" implements="true" extends="true"/>
</FunctionIdentifier>
<FunctionIdentifier id="resource3">
  <NamespaceName>
    <Value>java.sql</Value>
  </NamespaceName>
  <ClassName>
    <Value>Statement</Value>
  </ClassName>
  <FunctionName>
    <Value>executeQuery</Value>
  </FunctionName>
  <ApplyTo overrides="true" implements="true" extends="true"/>
</FunctionIdentifier>
<FunctionIdentifier id="release1">
  <NamespaceName>
    <Value>java.sql</Value>
  </NamespaceName>
  <ClassName>
    <Value>Connection</Value>
```

```
    </ClassName>
    <FunctionName>
      <Value>close</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <FunctionIdentifier id="release2">
    <NamespaceName>
      <Value>java.sql</Value>
    </NamespaceName>
    <ClassName>
      <Value>Statement</Value>
    </ClassName>
    <FunctionName>
      <Value>close</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <FunctionIdentifier id="release3">
    <NamespaceName>
      <Value>java.sql</Value>
    </NamespaceName>
    <ClassName>
      <Value>ResultSet</Value>
    </ClassName>
    <FunctionName>
      <Value>close</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <Definition><![CDATA[
      state unallocated (start);
      state allocated;
      state released;
      state leaked (error);

      var c;

      unallocated -> allocated{ c = resource1(...) | c = resource2(...) |
                                  c = resource3(...) }
      allocated-> released { c.release1(...) | c.release2(...) | c.release3
(...) |
```

```
                              #ifblock(c == null, true) }
      allocated-> leaked { #end_scope(c) }
  ]]></Definition>
</ControlflowRule>
```

The rule declares the initial state unallocated using the additional `(start)` keyword. Also, the rule declares the unsafe leaked state using the additional `(error)` keyword. Each method that allocates a `Connection`, `Statement`, or `ResultSet` objects has a separate function identifier element `resource1`, `resource2`, or `resource3`. The corresponding methods for releasing these objects are identified as `release1`, `release2`, and `release3`. The analyzer transitions between the declared states for a given object based on declared conditions in the rule such as the execution of the declared functions.

The condition `#end_scope(x)` describes the special circumstance where the `object x` has exited scope and is no longer accessible. In this rule, the object has been allocated in the allocated state. It reaches the error state leaked if the object falls out of scope and is in the allocated state at the time.

The condition `#ifblock(x == y,z)` describes the presence of an if-block statement within the code. It states that if `x` equals `y` with a result of `z`, the condition is satisfied and the analyzer should transition to the declared state. In this rule, the conditional `#ifblock(c == null,true)` describes an equality comparison between the tracked `object c` and the value null. If `c` is equal to `null`, code did not successfully allocate `object c`. The analyzer should safely transition the `object c` to its safe state because it is impossible for the object to leak resources.

There is a leak that the analyzer does not correctly identify using just this rule. The code deallocates the Statement object within the finally block after it calls the `System.exit()` method. The code never deallocates the object correctly because the `System.exit()` method prematurely exits the code. The allocated object reaches the end-of-scope condition prematurely.

The analyzer needs special knowledge of methods that prematurely force an out-of-scope condition. Otherwise, the analyzer cannot always identify when code forces an end-of-scope condition. The following non-returning rule describes this special quality of the `System.exit()` method:

```
<NonReturningRule formatVersion="23.1" language="java">
  <RuleID>775F5047-856C-4874-92A0-ADCE882AE4BB</RuleID>
  <FunctionIdentifier>
    <NamespaceName>
      <Value>java.lang</Value>
    </NamespaceName>
    <ClassName>
      <Value>System</Value>
    </ClassName>
    <FunctionName>
      <Value>exit</Value>
    </FunctionName>
  </FunctionIdentifier>
</NonReturningRule>
```

When Fortify Static Code Analyzer includes the non-returning rule and control flow rules in a scan, the Control Flow Analyzer identifies that the `Statement` object is not disposed of before it reaches its premature end-of-scope condition.

Null Pointer Check

This scenario highlights rules that enable the Control Flow Analyzer to detect missing null pointer check vulnerabilities. The scenario demonstrates how to exploit a missing null pointer check vulnerability. Then it illustrates how the Control Flow Analyzer uses rules to identify this type of vulnerability.

This scenario highlights the following vulnerability:

- Missing check against null—Program can dereference a null pointer because it does not check the return value of a function that might return null

This scenario highlights the following analysis and rules concepts:

- Error state
- Finite state machine
- Starting state

Source Code

The application contains a missing null pointer check within its messaging service. An attacker can submit a request to display a message and omit necessary pieces of information from the request. The application throws an exception, and discloses architecture and configuration information to the attacker.

The following example JSP code is from the application that retrieves and displays a message. It contains a missing null check vulnerability:

```
<% String incomingParameter = request.getParameter("id");
   Long decodedParameter = Long.decode(incomingParameter.trim());

   Message msg = (Message)(MessageService.getMessage(decodedParameter).get
(0));
   pageContext.setAttribute("severity" msg.getSeverity());
   pageContext.setAttribute("sender" msg.getSender());
   pageContext.setAttribute("subject" msg.getSubject());
   pageContext.setAttribute("body, msg.getBody());
%>
   ...
```

To view a message, the browser submits a HTTP request on behalf of the user:

```
http://localhost:8080/riches/pages/content/ViewMessage.jsp?id=1
```

To exploit the missing null check vulnerability, the attacker submits a modified HTTP request:

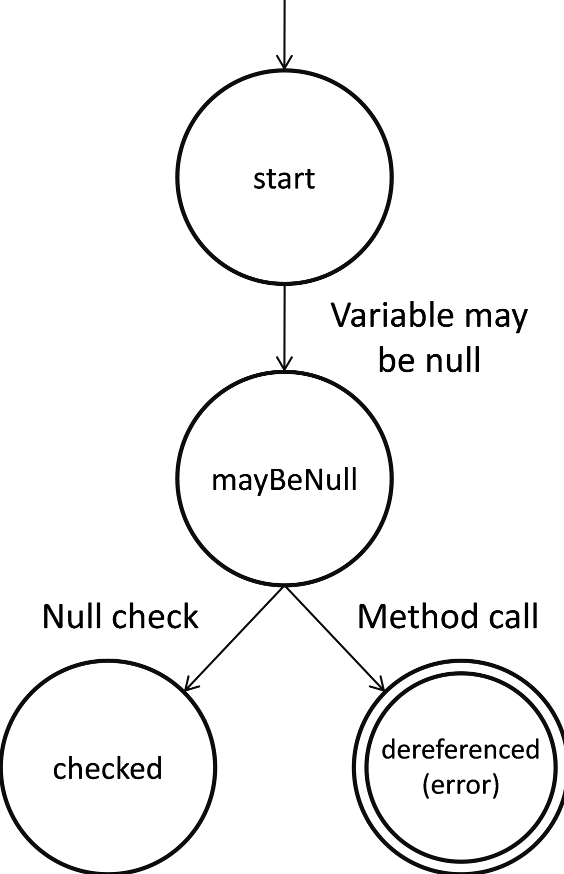
```
http://localhost:8080/riches/pages/content/ViewMessage.jsp
```

The `id` parameter is no longer present and the `incomingParameter` variable is set to null. Then, the JSP code calls `incomingParameter.trim()` and a null pointer exception occurs. Finally, the framework sends the unhandled exception and other sensitive information to the attacker's browser.

Rules

The application contains a missing null pointer check within its messaging service. An attacker can submit a request to display a message and omit necessary pieces of information from the request. The application throws an exception and discloses sensitive information to the user about its architecture and configuration.

The following illustration shows JSP code from the application that retrieves and displays a message:



The Control Flow Analyzer sets the FSM state to `mayBeNull` when it observes that the JSP code assigns a value to the `incomingParameter` variable. At this point, the code has not yet verified that the variable's value is not null.

Then, the analyzer observes that the code calls a method on the `incomingParameter` variable without inspecting its value. The analyzer transitions the variable's FSM from the `mayBeNull` state to

the `dereferenced` error state. The analyzer reports the vulnerability when it transitions the FSM into the error state.

Ideally, the code should inspect the object's value before using it. The analyzer then observes that the code performs this check and transitions the object's FSM from the `mayBeNull` state to the `checked` safe state.

The following rule describes the FSM model as a control flow:

```
<ControlflowRule formatVersion="23.1" language="java">
  <RuleID>4A2D77FD-C901-4F22-9994-23330BC56D96</RuleID>
  <VulnCategory>Missing Check against Null</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <FunctionIdentifier id="get">
    <NamespaceName>
      <Value>javax.servlet</Value>
    </NamespaceName>
    <ClassName>
      <Value>ServletRequest</Value>
    </ClassName>
    <FunctionName>
      <Value>getParameter</Value>
    </FunctionName>
    <ApplyTo overrides="true" implements="true" extends="true"/>
  </FunctionIdentifier>
  <FunctionIdentifier id="any">
    <NamespaceName>
      <Pattern>.*</Pattern>
    </NamespaceName>
    <ClassName>
      <Pattern>.*</Pattern>
    </ClassName>
    <FunctionName>
      <Pattern>.*</Pattern>
    </FunctionName>
  </FunctionIdentifier>
  <Definition><![CDATA[
      state start (start);
      state mayBeNull;
      state checked;
      state dereferenced (error);

      var f;
      start -> mayBeNull { f = $get(...) }
```

```
      mayBeNull -> checked { #compare(f, null) }
      mayBeNull -> dereferenced { f.$any(...) | *f }
  ]]></Definition>
</ControlflowRule>
```

The analyzer initializes the FSM in the `start` state start. The transition from the `start` state to the `mayBeNull` state occurs when the analyzer observes a call to a function matched by `$get`, and the FSM is bound to the value returned by that function.

The analyzer transitions the FSM from the `mayBeNull` to the `checked` state when it encounters code that compares the value to null. The `#compare(f,null)` statement describes this transition.

Alternatively, the analyzer transitions the FSM from the `mayBeNull` state to the `dereferenced` error state if code dereferences the value while in this state. The statement following statement describes this transition:

```
allocated -> used { f.$any(...) | *f }
```

Chapter 6: Content and Configuration Analyzer Rules

This section contains the following topics:

Content Analyzer and Custom Rules 107
XML Representation of Content Analyzer Rules 107
Configuration Analyzer and Custom Rules 108
XML Representation of Configuration Analyzer Rules 108
Custom Configuration Rule Scenarios 112

Content Analyzer and Custom Rules

The Content Analyzer finds security issues and policy violations in HTML content. In addition to static HTML pages, the Content Analyzer performs these checks on files that contain dynamic HTML, such as PHP, JSP, and classic ASP files.

Content Analyzer rules use XML XPath notation to describe problematic constructs in HTML files. The Content Analyzer converts the HTML content into an XML form and applies the XPath rules to this XML form.

XML Representation of Content Analyzer Rules

The following example shows a content rule:

```
<ContentRule formatVersion="23.1">
  <RuleID>941E1563-D3A2-B73D-10D1-8C035CCCDE66</RuleID>
  <VulnCategory>Form Definition</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <XPathMatch expression="//*[local-name()='form']"/>
</ContentRule>
```

The following table describes the XML elements introduced in the previous content rule example.

Element	Description
XPathMatch	This element has the following attribute: • `expression`—Specifies the XPath expression that the Content Analyzer evaluates against the XML representation of HTML documents

Configuration Analyzer and Custom Rules

The Configuration Analyzer finds security issues in application configuration files. This analysis can find instances where an application is configured insecurely, and can also enforce security policies by identifying configuration files that are not in compliance with those policies. Configuration Analyzer rules specify constraints on configuration properties.

The Configuration Analyzer understands XML files, Java properties files, and Dockerfiles. Each rule operates on one type of file. Rules that analyze XML files and Dockerfiles use XPath notation to describe the XML constructs that the analyzer should report. Rules that analyze properties files specify either property names or property values that the analyzer should report. You can restrict rules of each type to run only on files with specific names.

The Configuration Analyzer includes regular expression analysis to detect vulnerable secrets such as passwords, keys, and credentials in source code.

XML Representation of Configuration Analyzer Rules

This section describes the XML for the following Configuration Analyzer rules:

Configuration Rules 108

Regular Expression Rules 111

Configuration Rules

Use configuration rules to check XML, Dockerfiles, or properties files. Configuration rules have a sequence of `<Check>` elements. Each `<Check>` element specifies the properties and files that the Configuration Analyzer checks. The contents of the `<Check>` element varies depending on the type of file that the Configuration Analyzer is checking.

The following example shows a configuration rule for an XML file:

```
<ConfigurationRule formatVersion="23.1">
  <RuleID>8104EB17-C54C-7F22-C308-42C207C74BBD</RuleID>
  <VulnCategory>Servlet Mapping</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Check>
    <ConfigFile type="xml">
      <Value>web.xml</Value>
    </ConfigFile>
    <XPathMatch expression="//servlet-mapping"/>
  </Check>
</ConfigurationRule>
```

The following table describes the `<Check>` child elements.

Element	Description
ConfigFile	Specifies the file to check. This element has the following attribute: • `type`—Defines the type of configuration file. The valid values are `docker`, `properties`, and `xml`. The `<ConfigFile>` element also contains a `<Value>` or `<Pattern>` element that is checked against the file name of every file of the specified type. The check only applies to files for which the file type matches the `type` attribute and the file name matches the `<Value>` or `<Pattern>` inside the `<ConfigFile>` element.
XPathMatch	This element is required for XML files and Dockerfiles. This element has the following attribute: • `expression`—Specifies the XPath expression that the Configuration Analyzer evaluates against the XML representation of HTML documents.

For properties files, set the `type` attribute of the `<ConfigFile>` element to `properties`. The following example shows a name and value match for a properties file:

```
<ConfigurationRule formatVersion="23.1">
  <RuleID>FEC3D9F0-F29A-231B-3BD5-765CCEAF1CE5</RuleID>
  <VulnCategory>Security Not Enabled</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <Check>
    <ConfigFile type="properties">
```

```
      <Value>security.properties</Value>
    </ConfigFile>
    <NameMatch><Value>security</Value></NameMatch>
    <ValueMatch><Value>off</Value></ValueMatch>
  </Check>
  <Check>
    <ConfigFile type="properties">
      <Value>security.properties</Value>
    </ConfigFile>
    <NameMatch><Value>security</Value></NameMatch>
    <NotPresent/>
  </Check>
</ConfigurationRule>
```

The following table describes the `<Check>` child elements introduced in the previous configuration rule for a properties file example.

Element	Description
NameMatch	Specifies the property name check.
ValueMatch	(Optional) Contains a `<Pattern>` or `<Value>` element to check against the value of properties whose name matches the `<NameMatch>` element.
NotPresent	(Optional) Specifies whether the analyzer should report an issue if no property matching the `<NameMatch>` element is specified in a properties file that the `<ConfigFile>` element matches.

The following example shows a configuration rule for a Dockerfile:

```
<ConfigurationRule formatVersion="23.1">
  <RuleID>43f78d03-c2cb-46ed-bfaa-d2d507a61725</RuleID>
  <VulnKingdom>Environment</VulnKingdom>
  <VulnCategory>Dockerfile Misconfiguration</VulnCategory>
  <VulnSubcategory>Unapproved Image</VulnSubcategory>
  <DefaultSeverity>5.0</DefaultSeverity>
  <Description />
  <Check>
    <ConfigFile type="docker">
      <Pattern>.*\.[D|d]ockerfile</Pattern>
    </ConfigFile>
    <XPathMatch expression="//Instruction[@name='FROM']//Image[not(matches
(text(),'(approvedImage1|approvedImage2)\.company\.com'))]"/>
  </Check>
</ConfigurationRule>
```

Regular Expression Rules

Use regular expression rules to find vulnerabilities in both file content and file names (paths) using regular expressions. The rule can include regular expressions for the file content or a file name or both. If you use both file content and file name in a RegexRule, both regular expressions must match in order for the rule to flag a vulnerability.

The following example uses both file content and file name regular expressions to find occurrences of `jsmith` in files with extensions `.txt`, `.js`, and `.html`.

```
<RegexRule formatVersion="23.1">
  <RuleID>8076CEE1-D63B-4DA8-947C-98DE4BD33873</RuleID>
  <VulnKingdom>Security Features</VulnKingdom>
  <VulnCategory>Password Management</VulnCategory>
  <VulnSubcategory>Hardcoded Username</VulnSubcategory>
  <DefaultSeverity>5.0</DefaultSeverity>
  <Description/>
  <ContentRegex>jsmith</ContentRegex>
  <FileNameRegex>.*\.(txt|js|html)</FileNameRegex>
</RegexRule>
```

The following table describes the child elements.

Element	**Description**
ContentRegex	This element specifies a regular expression to match in the file content. Use

Element	Description
	the regular expression syntax defined in com.google.RE2. **Tip:** • `^` and `$` match begin and end of a file (in RE2, the m flag for multi-line mode is false by default) • A `.` does not match newline (in RE2, the s flag is false by default) • Uses the *find* instead of *match* method (regular expressions in the `ContentRegex` element do not need to account for characters or rows before or after the intended match)
FileNameRegex	This element specifies a regular expression to match in the file name (or path). Use the regular expression syntax defined in java.util.regex package. **Tip:** • Regardless of the operating system, use forward slash (/) for path separators • Uses *match* instead of *find* method (for example, to match `secret.key` in the path `C:/path/to/sourcecode/src/main/resources/secret.key`, use the regular expression `.+/secret\.key`) • Matches are made on absolute paths

Custom Configuration Rule Scenarios

This section provides examples of custom configuration rules.

This section contains the following topics:

Property File 112
Tomcat File 114
Authentication Tokens in Files 115

Property File

This scenario demonstrates the rules that enable the Configuration Analyzer to detect configuration vulnerabilities. The scenario illustrates how an incorrect setting can lead to unexpected downtime in a production environment. Then it shows how the Configuration Analyzer uses rules to identify and report these incorrect settings.

This scenario highlights the following vulnerability:

- Environment misconfiguration—Configuration files for an application contain incorrect values in a production environment. These misconfigurations typically introduce other vulnerabilities, including those related to communication security, authentication, authorization, data security, and exception handling.

This scenario highlights the following analysis and rule concepts:

- Configuration rules
- Java regular expressions
- Property files

Source Code

By convention, users should send and receive messages through the gateway of the production mail system. In test cases, however, the system routes messages through the gateway of the test environment. In this scenario, the incorrect SMTP setting is released into the production environment.

The following example shows the sample SMTP configuration:

```
riches.mail.smtpHostname = mail.test.riches.com
riches.mail.smtpPort = 25
riches.mail.username = test
riches.mail.password = passw0rd1!
```

After loading these incorrect values, the mail handling code sends messages through `mail.test.riches.com` instead of the production gateway.

Rule

The following configuration rule detects the invalid SMTP hostname value in the properties file:

```
<ConfigurationRule formatVersion="23.1">
  <RuleID>B8319D1B-65B3-4BFA-A0BE-8F1891D727E9</RuleID>
  <VulnCategory>J2EE Misconfiguration</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <ConfigFile type="properties">
    <Value>mailserver.legacy.properties</Value>
  </ConfigFile>
  <PropertyMatch>
    <NameMatch>
      <Value>riches.mail.smtpHostname</Value>
    </NameMatch>
    <ValueMatch>
      <Pattern caseInsensitive="true">(.*)\.test.riches.com</Pattern>
```

```
    </ValueMatch>
  </PropertyMatch>
</ConfigurationRule>
```

The configuration rule targets the `mailserver.legacy.properties` properties file. It compares the value of the property `riches.mail.smtpHostname` to the Java regular expression `(.*)\.test.riches.com`. The value should never match a string with the following sequence: zero or more characters; a period; and then the characters `test.riches.com`. If this sequence occurs, the Configuration Analyzer identifies a configuration vulnerability.

Tomcat File

This scenario highlights the rules that enable the Configuration Analyzer to identify specific configuration vulnerabilities. The scenario demonstrates how a misconfiguration in the application can lead to the disclosure of sensitive information. It then shows how the Configuration Analyzer uses rules to identify this type of misconfiguration.

This scenario highlights the following vulnerability:

- J2EE Misconfiguration—Underlying infrastructure that supports the application is improperly configured

  This results in new vulnerabilities related to communication security, data security, and exception handling.

This scenario highlights the following analysis and rules concepts:

- Configuration rules
- Java regular expressions
- XML files
- XPath expressions

Source Code

The application is deployed in a Tomcat Web server shared by multiple applications. Some of the applications rely on the server to authenticate incoming requests. The Tomcat configuration file contains a realm that describes the authentication configuration of another application. The following example shows an incorrect configuration detection rule:

```
<Realm className="org.apache.catalina.realm.JAASRealm"
  appName="RichesDiscover"
  userClassNames="com.fortify.samples.riches.security.UserPrincipal"
  roleClassNames="com.fortify.samples.riches.security.RolePrincipal"
  debug = "3"/>
```

The realm descriptor application in the previous example uses an authentication configuration with a debug level greater than two. With this configuration, the authentication service logs the user names and passwords in a plain text file, which can compromise security.

Rule

The following rule identifies an XML document that contains a node Realm with a debug attribute value set to a number greater than two:

```
<ConfigurationRule formatVersion="23.1">
  <RuleID>E9E3B4F0-CBDA-4695-94FD-3D41D68D19CB</RuleID>
  <VulnCategory>J2EE Misconfiguration</VulnCategory>
  <DefaultSeverity>2.0</DefaultSeverity>
  <Description/>
  <ConfigFile type="xml">
    <Pattern>(.*)\.xml</Pattern>
  </ConfigFile>
  <XPathMatch expression="count(//Realm[@debug > 2]) > 0"
   reporton="//Realm[@debug > 2]/@debug"/>
</ConfigurationRule>
```

The XPath expression `//Realm[@debug > 2])` describes the XML content necessary for the Configuration Analyzer to identify the misconfiguration. The expression identifies any Realm elements that have a debug attribute with value greater than two. The `<XPathMatch reporton>` condition specifies that Fortify Static Code Analyzer reports the problematic debug attribute instead of the parent Realm element.

Authentication Tokens in Files

In this scenario, a company has an internal system to which they authenticate with tokens. The tokens are 30 characters long and prefixed with `CST_`. A regular expression rule can be used to find these tokens in source code and other text-based files.

Rule

The following rule identifies files that contain the authentication token:

```
<RegexRule formatVersion="23.1">
  <RuleID>42DC4DF5-663D-456A-9E40-98313BD43C2A</RuleID>
  <VulnKingdom>Security Features</VulnKingdom>
  <VulnCategory>Key Management</VulnCategory>
  <VulnSubcategory>Hardcoded API Token</VulnSubcategory>
  <DefaultSeverity>3.0</DefaultSeverity>
  <Description ref="desc.regex.universal.key_management_hardcoded_
encryption_key" />
```

```
  <ContentRegex>CST_[a-zA-Z0-9]{30}</ContentRegex>
</RegexRule>
```

Chapter 7: Manipulation Rules

This section contains the following topics:

Suppression Rules 117
XML Representation of Suppression Rules 119
Alias Rules 119
XML Representation of Alias Rules 120
Result Filter Rules 120
XML Representation of Result Filter Rules 121

Suppression Rules

You can use a custom suppression rule to disable an existing rule so that it is no longer processed during analysis. You might want to suppress a rule if you notice that a particular rule produces many false positives. If you heavily invest in custom rules, this might be the preferred approach as the suppression rules reside with your custom rules.

Keep in mind that unlike suppression of specific issues during an audit, the suppression rules eliminate all results generated by a particular rule. Furthermore, if a rule that you have suppressed is updated in a future Rulepack release, you will need to manually disable the suppression rule to utilize the update.

The following table describes alternate methods of removing issues from your results.

Method	Stage Used	Advantages	Disadvantages	Notes
Filter file	During the scan	• Shorter scan times • Smaller FPR file size • Easy to filter out an entire class of vulnerabilities or specific rules that detect them	• Requires a rescan to generate filtered-out issues • Issues are blindly filtered out based on category, rule ID, or instance ID	For more information about filter files, see the *Fortify Static Code Analyzer User Guide*.

Method	Stage Used	Advantages	Disadvantages	Notes
Cleanse rule	During the scan	• Rules reside with other custom rules • Only filters out issues that involve validated data	• Requires additional effort to review APIs and develop rules	For information about dataflow cleanse rules, see "Cleanse Rules" on page 65.
Suppress issue post scan	During the audit	• Easy to filter out issues from the view • Issue remains suppressed over rescans and audits	• Does not decrease the scan time • Does not reduce the FPR file size	You can suppress issues in an auditing tool such as Fortify Audit Workbench or Fortify Software Security Center.
Filter set	During the scan	• Easy to filter out issues based on many different attributes • Smaller FPR file size	• Does not always decrease the scan time	For more information see the *Fortify Static Code Analyzer User Guide* and the *Fortify Audit Workbench User Guide*.
	During the audit	• Easy to filter out issues from the view based on many different attributes • No rescan required to make the filtered issues become visible again	• Does not reduce the FPR file size	

XML Representation of Suppression Rules

The following example shows a suppression rule:

```
<SuppressionRule formatVersion="23.1">
  <RuleID>6200BAFA-AA33-4DEE-A1E7-B9EDDBA9D315</RuleID>
</SuppressionRule>
```

The `<RuleID>` element specifies the unique rule identifier for the Fortify rule you want to disable.

> **Note:** If the suppression rule `formatVersion` attribute value is earlier than the format version of the latest rule with that rule ID, then the suppression rule only affects earlier versions of that rule. For example, if the default Rulepacks contain three versions of the rule 941E1563-D3A2-B73D-10D1-8C035CCCDE99 (for versions: 3.60, 16.20, and 20.1), and the formatVersion of the suppression rule is 16.20, then the suppression rule only suppresses the 3.60 and 16.20 versions of the rule. This means that the rule identified by 941E1563-D3A2-B73D-10D1-8C035CCCDE99 is still triggered by Fortify Static Code Analyzer versions later than 16.20.

Alias Rules

Use alias rules to specify a function that mimics the behavior of another function. Alias rules are ideal to use when your organization invests in the usage of proprietary libraries and APIs that mimic the behavior of standard functions covered by the Fortify Secure Coding Rulepacks.

For example, it is common for organizations to implement their own version of memory management functions that behave similarly to the standard `malloc()` function. For this example, assume that the proprietary function is called `my_malloc()`. The standard Secure Coding Rulepacks include several different types of rules for the `malloc()` function, but no rules exist for `my_malloc()`. To generate complete results during the scan of their code, organizations can either:

- Write every type of rule that the Fortify team has already written for `malloc()` for the proprietary `my_malloc()` function
- Write one alias rule to indicate that the proprietary function `my_malloc()` behaves exactly like `malloc()`

The first option is time consuming and error-prone, while the second option makes sure that all the Fortify rules written for `malloc()` are automatically triggered on `my_malloc()`.

XML Representation of Alias Rules

The functions defined in the `<From>` element will match the defined functions that exist in the standard Fortify Secure Coding Rulepacks defined in the `<To>` element. The following example alias rule shows that the proprietary function `my_malloc()` behaves exactly like `malloc()`:

```
<AliasRule formatVersion="23.1" language="cpp">
  <RuleID>5705796F-A199-7D69-35F6-B18C9AA5631C</RuleID>
  <From>
    <FunctionName>
       <Pattern>my_malloc</Pattern>
    </FunctionName>
  </From>
  <To>
    <FunctionName>malloc</FunctionName>
  </To>
</AliasRule>
```

For descriptions of the alias rule elements, see "FunctionIdentifier Element" on page 21.

Result Filter Rules

Fortify Static Code Analyzer uses several analyzers and mechanisms to detect potential vulnerabilities in the source code. Sometimes the same vulnerability is detected in several different ways or manifests itself through more than one Fortify result, generating a duplicate issue. For example, SQL Injection vulnerabilities are detected by both the Semantic Analyzer and the Dataflow Analyzer, which results in two issues being reported: one by each analyzer. Similarly, SQL Injection and Access Control: Database results, when reported on the same line of code, can represent the same lack of validation issue, generating a duplicate issue. While Fortify Static Code Analyzer performs automatic filtering internally, the filtering capability is also available externally in the form of result filter rules. With result filter rules, you can specify that one issue is treated as dominant and is included in the analysis results instead of other (subordinate) issues detected on the same line of code.

XML Representation of Result Filter Rules

Result filter rules have a `<Check>` element that encompasses the primary logic of the filtering directive. You can filter based on the analyzer and optionally a category name and a rule ID. For each of the three options, you must specify the dominant value and the subordinate value with `<Dominant>` and `<Subordinate>` elements, respectively. You can also use a special value of `<SameValue/>` to specify that the dominant and subordinate category name or rule ID are the same.

In the following example, if several dataflow issues are detected by rules A, B, or C on the same line of code, the issue detected by rule A is reported and the issues detected by rules B or C are filtered out of the results:

```
<ResultFilterRule formatVersion="23.1">
  <RuleID>D811682A-C81D-43C6-BA82-7A38143626B9</RuleID>
  <Check>
    <AnalyzerName>
       <Dominant>dataflow</Dominant>
       <Subordinate>dataflow</Subordinate>
    </AnalyzerName>
    <RuleID>
      <Dominant>
        <Value>[UNIQUE_RULEID_A]</Value>
      </Dominant>
      <Subordinate>
        <Pattern>[UNIQUE_RULEID_B]|[UNIQUE_RULEID_C]</Pattern>
      </Subordinate>
    </RuleID>
   </Check>
</ResultFilterRule>
```

In the following example, if both the Dataflow Analyzer and the Control Flow Analyzer detected several issues for category A, only the issue detected by the dominant analyzer (Dataflow) is included in the results:

```
<ResultFilterRule formatVersion="23.1">
  <RuleID>A258D536-F153-4B7E-9BAF-B8B895BA5719</RuleID>
  <Check>
    <AnalyzerName>
      <Dominant>dataflow</Dominant>
      <Subordinate>controlflow</Subordinate>
    </AnalyzerName>
    <Category>
      <Dominant>
        <Value>[CATEGORY_A]</Value>
      </Dominant>
      <Subordinate>
        <Value>[CATEGORY_A]</Value>
      </Subordinate>
    </Category>
  </Check>
</ResultFilterRule>
```

In the following example, if multiple issues are reported on the same line of code, the Dataflow Analyzer results dominate over the Structural Analyzer results with the same category names:

```
<ResultFilterRule formatVersion="23.1">
  <RuleID>A5035F9A-E92B-4337-8F6D-26F6D98737ED</RuleID>
  <Check>
    <AnalyzerName>
      <Dominant>dataflow</Dominant>
      <Subordinate>structural</Subordinate>
    </AnalyzerName>
    <Category>
        <SameValue/>
    </Category>
  </Check>
</ResultFilterRule>
```

The following table describes the `<Check>` child elements used in the previous result filter rule examples.

Element	**Description**
AnalyzerName	Specifies the analyzer name. The valid values are `buffer`, `configuration`, `content`, `controlflow`, `dataflow`, `semantic`, and `structural`.
Category	Use the `<Dominant>` and `<Subordinate>` child elements to specify the dominant and subordinate vulnerability categories. You can specify this value as either a string (`<Value>` element) or a regular expression (`<Pattern>` element).
RuleID	Use the `<Dominant>` and `<Subordinate>` child elements to specify the dominant and subordinate rule identifiers. You can specify this value as either a string (`<Value>` element) or a regular expression (`<Pattern>` element).

Chapter 8: Custom Vulnerability Category Mapping

Fortify distributes an external metadata document with the Secure Coding Rulepacks. This XML document provides mappings from the vulnerability categories in the Fortify Taxonomy to alternative categories (such as CWE, OWASP Top 10, and PCI). For more information, see The Evolution of a Taxonomy: Ten Years of Software Security. You can customize these mappings found in standards and industry best practices or create your own files to map Fortify issues to different taxonomies, such as internal application security standards or additional compliance obligations.

This section contains the following topics:

Mapping Fortify Categories to Alternative External Categories 124

External Metadata XML Structure 125

Example Mappings 132

Mapping Fortify Categories to Alternative External Categories

To add custom vulnerability category mappings for your organization, you need to create your own external metadata document. You need to map the external categories to the Fortify categories. You can find the list of Fortify categories on the Fortify Taxonomy website at https://vulncat.fortify.com.

> **Note:** The Fortify-provided external metadata XML document is in the `<sca_install_dir>/Core/config/ExternalMetadata` directory. This document is overwritten whenever you update the security content.

Use any XML editor to create a new external metadata document. See "External Metadata XML Structure" on the next page for an overview of the XML hierarchy. Save your new document to the `<sca_install_dir>/Core/config/CustomExternalMetadata` directory so that your changes are not lost during security content updates. Name the file you put into the `CustomExternalMetadata` folder `<anything>.xml`. Placing your custom external metadata file into the `CustomExternalMetadata` folder, makes the custom category mappings available to Fortify Audit Workbench.

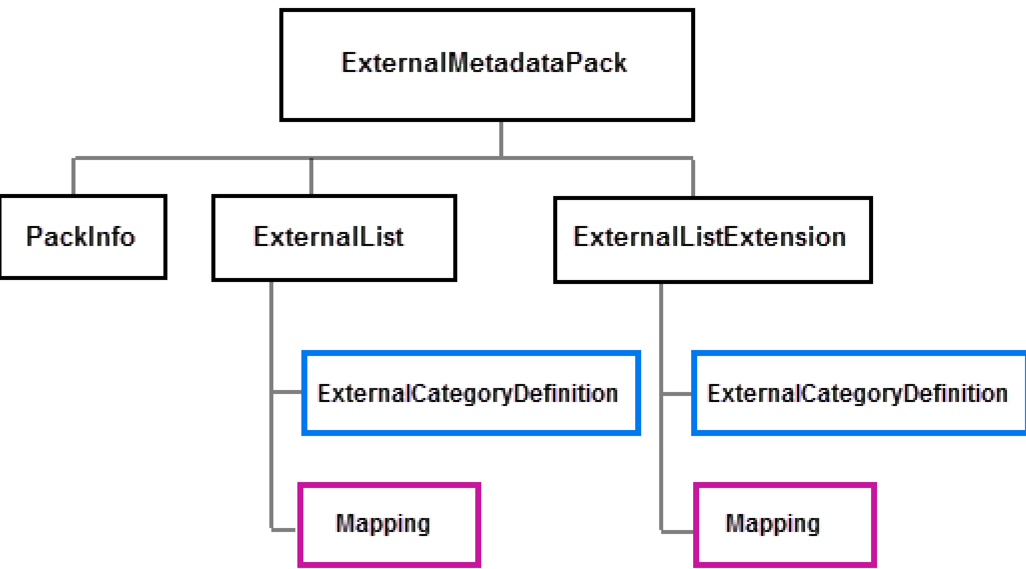
To validate your custom category mappings, use the `externalmetadata.xsd` file, which is available in the `<sca_install_dir>/Core/config/schemas` directory.

To apply the custom external metadata XML document across all applications in Fortify Software Security Center, you must first import it into Fortify Software Security Center. For more information, see the *Fortify Software Security Center User Guide*.

After you import your custom external metadata document and scan your projects, custom category names are displayed in the Group By menu as you review the analysis results in Fortify Audit Workbench and Fortify Software Security Center. You can also search for issues using your custom category names.

External Metadata XML Structure

The following diagram provides an overview of the external metadata document's XML structure. The `<ExternalList>` and the `<ExternalListExtention>` elements contain the same two elements. The following sections describe the elements in detail. Note that all elements and options are required unless indicated otherwise.



ExternalMetadataPack Element

The root element of the external metadata document is `<ExternalMetadataPack>`. The `<ExternalMetadataPack>` element contains the following XML child elements: `<PackInfo>`, `<ExternalList>`, and `<ExternalListExtension>`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ExternalMetadataPack
xmlns="xmlns://www.fortifysoftware.com/schema/externalMetadata" schemaVersion="1.1">
  <PackInfo>...</PackInfo>
  <ExternalList>... </ExternalList>
  <ExternalList>... </ExternalList>
  ...
  <ExternalList>... </ExternalList>
  <ExternalListExtension>...</ExternalListExtension>
  <ExternalListExtension>...</ExternalListExtension>
```

```
  ...
  <ExternalListExtension>...</ExternalListExtension>
</ExternalMetadataPack>
```

Fortify Software checks the `schemaVersion` attribute to confirm that you are using a supported schema version. You can find the current version in the `<sca_install_dir>/Core/config/schemas/externalmetadata.xsd` file.

The following table describes the `<ExternalMetadataPack>` child elements.

Element	**Description**
PackInfo	Contains information about the external metadata package. You must have one `<PackInfo>` element in an external metadata document. See "PackInfo Element" on the next page.
ExternalList	(Optional) A group of external mappings. **Note:** Only use `<ExternalList>` elements for new external mappings. To extend an existing list, use the `<ExternalListExtension>` element. This element contains the following attribute: `obsolete`—(Optional) A value of `true` indicates that the external list group is obsolete. Fortify software products do not display obsolete external lists for grouping and filtering scan results. The default value is `false`. An `<ExternalMetadataPack>` can include multiple `<ExternalList>` elements. Each `<ExternalList>` element contains: • Name • Description • External category descriptions • Mappings of external to Fortify categories See "ExternalList Element" on page 128.

Element	Description
ExternalListExtension	(Optional) A group of supplemental definitions and mappings for an existing `<ExternalList>` element. An `ExternalMetadataPack` can include multiple `<ExternalListExtension>` blocks. The `<ExternalListExtension>` element enables you to supplement the mappings in an existing list, however, you cannot override existing mappings. Each `<ExternalListExtension>` element contains: • Parent identifier (reference to an existing `<ExternalList>` element) • External category descriptions • Mappings of external to Fortify categories See "ExternalListExtension Element" on page 129.

PackInfo Element

The `<PackInfo>` element contains high-level information to provide a unique name, identifier, and version for the mapping file. The following is an example:

```
<PackInfo>
  <Name>Main External List Mappings</Name>
  <PackID>main-external-mappings</PackID>
  <Version>2020.3.0.0009</Version>
</PackInfo>
```

The following table describes the `<PackInfo>` child elements.

Element	Description
Name	A unique name for the `<ExternalMetadataPack>`. Spaces are allowed.
PackID	A unique identifier for the `<ExternalMetadataPack>`.
Version	Version of the `<ExternalMetadataPack>`.
Description	(Optional) A description of this external metadata.
Locale	(Optional) The locale for the Rulepack. The valid values are `en`, `es`, `ja`, `ko`, `pt_BR`, `zh_CN`, and `zh_TW`.

ExternalList Element

The <ExternalList> element contains a list of external categories and mappings. The following example shows a partial external list element for OWASP Top 10 2017:

```
<ExternalList>
  <ExternalListID>3C6ECB67-BBD9-4259-A8DB-B49328927248</ExternalListID>
  <Name>OWASP Top 10 2017</Name>
  <Shortcut>OWASP2017</Shortcut>
  <Shortcut>OWASP 2017</Shortcut>
  <Shortcut>OWASP Top Ten 2017</Shortcut>
  <Shortcut>OWASP Top 10 2017</Shortcut>
  <Description>The OWASP Top Ten 2017 provides a
     powerful awareness document for web application security
     ...
  </Description>
  <Group>OWASP</Group>
  <ExternalCategoryDefinition>...</ExternalCategoryDefinition>
  <ExternalCategoryDefinition>...</ExternalCategoryDefinition>
    ...
  <ExternalCategoryDefinition>...</ExternalCategoryDefinition>

  <Mapping>...</Mapping>
  <Mapping>...</Mapping>
  ...
  <Mapping>...</Mapping>
</ExternalList>
```

The following table describes the <ExternalList> child elements.

Element	Description
ExternalListID	Unique identifier (GUID) for the external list.
Name	Fully qualified name of the external list. This is the primary display string, and is also used for searches along with shortcuts.
Shortcut	(Optional) Shortcut names for the external list. You can use the shortcut strings in searches. You can have multiple shortcuts for an external list.
Description	Description of the external list.
Group	Group name used to tie several external lists together.

Element	Description
ExternalCategoryDefinition	(Optional) The <ExternalCategoryDefinition> element describes an external category. You should have one external category definition for each <ExternalCategory> in a <Mapping> element. You can have multiple <ExternalCategoryDefinitions> for an external list. Each <ExternalCategoryDefinition> contains a name, description, and order information (see "ExternalCategoryDefinition Element" on the next page).
Mapping	(Optional) The <Mapping> element maps the Fortify category to the external category. Each <ExternalCategory> in a <Mapping> element should have a corresponding <ExternalCategoryDefinition>. You can have multiple mappings for an external list. Each <Mapping> element contains a Fortify category and an external category (see "Mapping Element" on page 131).
OrderingInfo	(Optional) An integer that indicates the external list sort order within the <ExternalMetadataPack> element.

ExternalListExtension Element

The <ExternalListExtension> element contains an extension of an existing external list. Use this element to provide an extension to an existing mapping. For example, if you want to have your own custom rules, categories, and descriptions then it would be useful to also extend the mappings for relevant standards rather than write a new one.

```
<ExternalListExtension>
  <ExternalListID>EEE3F9E7-28D6-4456-8761-3DA56C36F4EE</ExternalListID>

  <ExternalCategoryDefinition>...</ExternalCategoryDefinition>
  <ExternalCategoryDefinition>...</ExternalCategoryDefinition>
  ...
  <ExternalCategoryDefinition>...</ExternalCategoryDefinition>

  <Mapping>...</Mapping>
  <Mapping>...</Mapping>
  ...
  <Mapping>...</Mapping>
</ExternalListExtension>
```

The following table describes the <ExternalListExtension> child elements.

Element	Description
ExternalListID	Reference to an existing external list identifier (required).
ExternalCategoryDefinition	(Optional) The `<ExternalCategoryDefinition>` element describes an external category. You should have one external category definition for each `<ExternalCategory>` in a `<Mapping>` element. You can have multiple `<ExternalCategoryDefinitions>` for an external list extension. Each `<ExternalCategoryDefinition>` contains a name, description, and order information. See "ExternalCategoryDefinition Element" below.
Mapping	(Optional) The `<Mapping>` element maps the Fortify category to the external category. Each `<ExternalCategory>` in a `<Mapping>` element should have a corresponding `<ExternalCategoryDefinition>`. You can have multiple mappings for an external list extension. Each `<Mapping>` element contains a Fortify category and an external category. See "Mapping Element" on the next page.

ExternalCategoryDefinition Element

The `<ExternalCategoryDefinition>` element describes an external category. The following example is from the Fortify-provided OWASP Top 10 2017 external list:

```
<ExternalCategoryDefinition>
  <Name>A7 Cross-Site Scripting (XSS)</Name>
  <Description>OWASP Top 10 Application Security Risks,
  A7:2017 states: "XSS flaws...  </Description>
  <OrderingInfo>7</OrderingInfo>
</ExternalCategoryDefinition>
```

The following table describes the `<ExternalCategoryDefinition>` child elements.

Element	Description
Name	Name of the external category.
Description	Description of the external category.
OrderingInfo	(Optional) An integer that indicates the external category definition sort order within the `<ExternalList>` element. Fortify Software components use the sort order when displaying categories, for example in reports.

Mapping Element

The <Mapping> element maps a Fortify category to an external category. The following example is from the Fortify-provided OWASP Top 10 2017 external list:

```
<Mapping>
  <InternalCategory>Cross-Site Scripting: Reflected
  </InternalCategory>
  <ExternalCategory>A7 Cross-Site Scripting (XSS)
  </ExternalCategory>
</Mapping>
```

The following table describes the <Mapping> child elements.

Element	Description
InternalCategory	Name of the Fortify vulnerability category from the seven pernicious kingdoms (for example, Cross-Site Scripting: Reflected). You can use the same Fortify category in more than one mapping, providing for many-to-one or many-to-many relationships. **Note:** The Fortify vulnerability category name is case-sensitive.
ExternalCategory	Name of the external category that is specified as the <Name> element in an <ExternalCategoryDefinition> element. You can use the same external category in more than one mapping, enabling one-to-many or many-to-many relationships.

XML Skeleton

You can copy and paste the following XML skeleton to create a new external metadata document:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<ExternalMetadataPack
xmlns="xmlns://www.fortifysoftware.com/schema/externalMetadata" schemaVersion="1.1">
  <PackInfo>
    <Name>PACKAGE_NAME</Name>
    <PackID>PACKAGE_ID</PackID>
    <Version>VERSION</Version>
  </PackInfo>

  <ExternalList>
    <ExternalListID>GUID</ExternalListID>
```

```
    <Name>EXTERNAL_LIST_NAME</Name>
    <Shortcut>EXTERNAL_LIST_SHORTCUT</Shortcut>
    <Description>DESCRIPTION</Description>
    <Group>GROUP_NAME</Group>

    <ExternalCategoryDefinition>
      <Name>EXTERNAL_CATEGORY_NAME</Name>
      <Description>EXTERNAL_CATEGORY_DESCRIPTION</Description>
      <OrderingInfo>CATEGORY_ORDER_NUMBER</OrderingInfo>
    </ExternalCategoryDefinition>

    <Mapping>
      <InternalCategory>FORTIFY_RULE_CATEGORY</InternalCategory>
      <ExternalCategory>EXTERNAL_CATEGORY</ExternalCategory>
    </Mapping>
    <OrderingInfo>LIST_ORDER_NUMBER</OrderingInfo>

  </ExternalList>
</ExternalMetadataPack>
```

Example Mappings

The following example shows some mappings for CERT SEI Coding Standards for Java:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<ExternalMetadataPack xmlns="xmlns://www.fortifysoftware.com/schema/externalMetadata"
 schemaVersion="1.1">
  <PackInfo>
    <Name>CERT SEI Coding Standard Mappings</Name>
    <PackID>cert-sei-mappings</PackID>
    <Version>2015.1.00</Version>
  </PackInfo>
  <ExternalList>
    <ExternalListID>27b303dc-71bf-4152-b921-105326f4597c</ExternalListID>
    <Name>CERT Java</Name>
    <Shortcut>CERTJAVA</Shortcut>
    <Description>xxx</Description>
    <Group>CERT</Group>

    <ExternalCategoryDefinition>
      <Name>IDS00-J. Prevent SQL injection</Name>
      <Description>xyz</Description>
      <OrderingInfo>1</OrderingInfo>
    </ExternalCategoryDefinition>

    <ExternalCategoryDefinition>
      <Name>FIO16-J. Canonicalize path names before validating them</Name>
      <Description>xyz</Description>
      <OrderingInfo>2</OrderingInfo>
    </ExternalCategoryDefinition>
```

```
    <ExternalCategoryDefinition>
      <Name>IDS03-J. Do not log unsanitized user input</Name>
      <Description>none</Description>
      <OrderingInfo>3</OrderingInfo>
    </ExternalCategoryDefinition>
    <Mapping>
      <InternalCategory>SQL Injection</InternalCategory>
      <ExternalCategory>IDS00-J. Prevent SQL injection</ExternalCategory>
    </Mapping>
    <Mapping>
      <InternalCategory>SQL Injection: Hibernate</InternalCategory>
      <ExternalCategory>IDS00-J. Prevent SQL injection</ExternalCategory>
    </Mapping>

    <Mapping>
      <InternalCategory>Log Forging</InternalCategory>
      <ExternalCategory>IDS03-J. Do not log unsanitized user input</ExternalCategory>
    </Mapping>
    <Mapping>
      <InternalCategory>Path Manipulation</InternalCategory>
      <ExternalCategory>FIO16-J. Canonicalize path names before validating them</ExternalCategory>
    </Mapping>
  </ExternalList>
</ExternalMetadataPack>
```

Appendix A: Taint Flag Reference

The tables in this appendix provide descriptions of the three types of taint flags included with the Fortify Secure Coding Rulepacks.

This section contains the following topics:

General Taint Flags 134

Specific Taint Flags 136

Neutral Taint Flags 139

General Taint Flags

General taint flags, as a group, identify common sources of untrusted data. The following table describes the general taint flags.

General Taint Flag ID	Description
ARGS	Indicates user input from command-line arguments.
CACHE	Indicates cache source for memcached injection sinks.
CHANNEL	Indicates source from `java.nio.channels` classes. Similar to STREAM, but for channels instead of stream classes.
CONSOLE	Indicates user input provided from the console.
CURSES	Indicates input from a curses window.
DATABASE	Indicates output from a database (treated as tainted input). Typically, XSS is set at the same time as this taint flag.
DNS	Indicates a Domain Name System.
ENVIRONMENT	Indicates retrieval of environment variables. Environment security is unknown and therefore this is seen as tainted input.
FILE_SYSTEM	Indicates input from a file.
FORM	Indicates input from a web form (for example, a standard HTML form). Typically, XSS is set at the same time as this taint flag.

General Taint Flag ID	Description
GUI_FORM	Indicates input from a GUI form. Typically, XSS is set at the same time as this taint flag.
ICC	Indicates Android Inter-Component Communication. Typically, XSS is set at the same time as this taint flag.
ICC_CLOUD	Indicates Cloud Inter-Component Communication. Used for sinks on Cross-Site Scripting: Inter-Component Communication (Cloud). Typically, XSS is set at the same time as this taint flag.
JSON	Indicates JSON documents. Typically, XSS is set at the same time as this taint flag.
LDAP	Indicates input from LDAP services. Typically, XSS is set at the same time as this taint flag.
NAMING	Indicates input from Naming services. Typically, XSS is set at the same time as this taint flag.
NETWORK	Indicates general input from a network. This taint flag is set when there is no specific taint flag suitable due to lack of context. For example, reading from a raw socket. Typically, XSS is set at the same time as this taint flag.
PROPERTY	Indicates input from system properties. This can come from a loaded properties file, properties configured on a server, and so on.
REGISTRY	Indicates Windows registry.
RPC	Indicates input from a remote procedure call.
SERIALIZED	Indicates serialized data.
STDIN	Indicates a standard input stream.
STREAM	Indicates input from a stream.
WEB	Indicates general input from the web. This is a more specific categorization than "NETWORK" above. You can also apply "XSS" on the next page with this taint flag.
WEBSERVICE	Indicates input from a web service. This is a more specific categorization than "NETWORK" above. Typically, XSS is set at the

General Taint Flag ID	Description
	same time as this taint flag.
XML	Indicates an XML document. Typically, XSS is set at the same time as this taint flag.
XSS	Used in conjunction with another general taint flag for cross-site scripting sinks. When taint from a network, inter-component communication, or some persistent service/document is used (for example, `DATABASE`, `FORM`, `GUI_FORM`, `ICC`, `ICC_CLOUD`, `JSON`, `LDAP`, `NAMING`, `NETWORK`, `WEBSERVICE`, `WEB` or `XML`), taints are added during a source along with `+XSS`. The variant of general taint flags along with `XSS` determines the type of cross-site scripting subcategory that is flagged. For example: `+XSS` and `+DATABASE` can mean that a `Cross-Site Scripting: Persistent` sink applies. `+XSS` and `+WEB` can mean that a `Cross-Site Scripting: Reflected` sink applies.

Specific Taint Flags

Specific taint flags include a declaration that describes the category of taint flag in the Secure Coding Rulepacks. The following table describes the specific taint flags.

Specific Taint Flag ID	Description
AES_ALGORITHM	Indicates a certain configuration object to have an underlying Advanced Encryption Standard algorithm.
AUTH_CHALLENGE	Indicates that an authentication challenge was sent to the client.
CLASS_NAME	Indicates a class name for sinks such as `Code Correctness: Erroneous Class Compare`.
CONTENTPROVIDERREAD	Indicates the entity requires an Android Missing Content Provider Read Permission. This is typically set on strings and fields associated with this type of permission.
CONTENTPROVIDERWRITE	Indicates the entity requires an Android Missing Content Provider Write Permission. This is typically set on strings and fields associated with this type of permission.
COOKIE_BROAD_DOMAIN	Source for finding `Cookie Security: Overly Broad Domain` issues.

Specific Taint Flag ID	Description
COOKIE_BROAD_PATH	Source for finding `Cookie Security: Overly Broad Path` issues.
COOKIE_NOT_SENT_OVER_SSL	Source for finding `Cookie Security: Cookie not Sent Over SSL` issues.
COOKIE_PERSISTENT	Source for finding `Cookie Security: Persistent Cookie` issues
COOKIE_PLIST	macOS/iOS cookie property list. Contains potentially private information from cookies.
CRYPTO_KEY	Indicates that the data is a cryptographic key. This is used together with PRIVATE.
DJANGO_SAFE_STRING	Set when a Django string is considered safe. Used to help with identifying XSS sinks in Django.
DJANGO_AUTOESCAPE	Set when Django auto-escaping is enabled. Used to help with identifying XSS sinks in Django.
DJANGO_DISABLED_ AUTOESCAPE	Set when Django auto-escaping is disabled. Used to help with identifying XSS sinks in Django.
DSA_ALGORITHM	Indicates a certain configuration object to have an underlying Digital Signature Algorithm.
FILESEPARATOR	Set to identify `Portability Flaw: File Separator` issues.
FLOATSTRING	Set when a float is converted to a string. Used to identify `Code Correctness: String Comparison of Float` issues.
HTML_RISKY_ATTR	Set to identify `Insecure Sanitizer Policy` issues.
HTML_RISKY_HREF	Set to identify `Insecure Sanitizer Policy` issues.
HTML_RISKY_IMGFRAME	Set to identify `Insecure Sanitizer Policy` issues.
INSECURE_PROTOCOL	Insecure Protocol Identifier.
INTENTRECEIVINGACTION	Indicates Android Intent Receiving Action.
INTENTSENDINGACTIVITY	Indicates Android Intent Sending Activity.
INTENTSENDINGBROADCAST	Indicates Android Intent Sending Broadcast.
INTENTSENDINGSERVICE	Indicates Android Intent Sending Service.
INTPTR_TO_PRIVATE	Set when taking a SecureString object and creating an IntPtr. This is then later used to attempt to find `Privacy Violation: Heap Inspection` issues.

Specific Taint Flag ID	Description
JAVASCRIPT	Indicates dynamically generated JavaScript.
JPANE_HTML_ON	Indicates that a JPane content type is set to HTML.
JSP_RESPONSE	Used to identify a JSP Response.
KEYBOARD_CACHE	Indicates Keyboard Cache.
LOCALSTORAGE	Indicates that HTML5 local storage is in use.
LOW_ENTROPY	Indicates that a source of low entropy was used. This might then lead to problems within cryptographic algorithms if random numbers are required.
MOBILE_DATA_ATTRIBUTE	Indicates mobile data lacking data protection.
MOBILE_PARTIAL_ PROTECTED_DATA	Indicates that a mobile platform has non-optimal file protection.
MOBILE_UNPROTECTED_DATA	Indicates that a mobile platform has no file protection.
NETWORKSTREAM	Identifies a general network output stream.
NOVALIDATION	Indicates that validation is disabled.
PASSWORD_IN_STR	Indicates a password in a string.
POORVALIDATION	Indicates a potentially incomplete validation against a cross-site scripting vulnerability, such as HTML encoding, URL encoding, and so on.
PRIVATE	Indicates private information.
PROCESSSTREAM	Indicates that an output stream is associated with a process object. For example, a `java.lang.Process` object in Java.
RSA_ALGORITHM	Indicates a certain configuration object to have an underlying RSA algorithm.
SESSIONSTORAGE	Indicates that HTML session storage is being used.
SHELL_PROCESS	Indicates that the process is run by a shell rather than directly.
SNAPSHOT_CACHE	Indicates Application Snapshot Cache.
SOAP_MESSAGE	Indicates a SOAP response message
SPRINGMODELMAP	Indicates that a map is a Spring model map. This might lead to a trust boundary violation.
SQLI_POORVALIDATION	Indicates a potentially incomplete validation against a SQL Injection

Specific Taint Flag ID	Description
	vulnerability, such as where escaping might not be enough protection from an attack.
SSL_URL	Indicates that an object creates an underlying URL connection, which might lead to insecure SSL/TLS configurations.
SYSTEM_KEYSTORE	Indicates that an object uses the default system keystore.
SYSTEMINFO	Indicates system information.
TRIPLEDES_ALGORITHM	Indicates a certain configuration object to have an underlying Triple DES (3DES) algorithm.
WEBSOCKET_STREAM	Identifies an output stream on a web socket.
WEBVIEW_JS_ENABLED	Identifies when a WebView has JavaScript enabled (leading to the possibility of certain types of attacks).
WEBVIEW_JS_DISABLED	Identifies when a WebView has JavaScript disabled, which prevents certain types of attacks.
WICKET_VALIDATION_OFF	Indicates that Wicket Validation is off.
XMLPROCESSOR	Identifies an XML processor. Used in conjunction with other taint to identify problems such as XEE and XXE.
XSSSTREAM	Indicates an output stream associated with an HTTP response.

Neutral Taint Flags

Neutral taint flags represent informational content. Neutral taint flags are most often used to note that a specific vulnerability category was validated. Neutral taint flags are useful to filter out false positives. The following table describes the neutral taint flags.

Neutral Taint Flag ID	Description
ANGULAR_TRUSTED_HTML	Indicates that dataflow is *trusted* HTML in an AngularJS application.
ANGULAR_TRUSTED_RESOURCE_URL	Indicates that dataflow is a *trusted* resource URL in an AngularJS application.
ANTISAMY_VALIDATION	Indicates that AntiSamy validation was performed. Requires manual auditing to identify whether this was sufficient.

Neutral Taint Flag ID	Description
APEX_VALIDATION_OFF	Indicates that Visualforce validation has been turned off.
BASE64_ENCODED	Indicates that the value has been Base64-encoded. Most vulnerabilities are not reported if the dataflow is Base64-encoded. This taint flag is removed when it is decoded again.
CONSTANTFILE	Indicates that the file object is instantiated based on the hardcoded file name.
CSS_ENCODE	Indicates that the tainted data was encoded for CSS.
CSV_ENCODE	Indicates data encoded for use in a CSV file.
DEOBFUSCATED	Prevents Password Management issues from appearing.
DYNAMIC	Indicates that dynamic Open SQL is used.
ENCRYPTED	Indicates encrypted data. This flag is added when data is encrypted, and then removed if the same data is then decrypted.
EXCEPTIONINFO	Indicates exception information.
HEALTH	Indicates that the information is health-related, such as with iOS HealthKit.
HIDDEN_FIELD	Indicates whether information comes from a hidden field. This affects the severity of `Access Control: Database` issues.
HTML_ATTR_ENCODE	Indicates HTML attribute encoding.
HTML_ENCODE	Indicates HTML encoding.
JAVA_ENCODE	Indicates Java encoding.
JS_ENCODE	Indicates JavaScript encoding.
LATERAL	Indicates a source of lateral SQL injection.
LDAP_DN_ENCODE	Indicates that encoding used is suitable for a Distinguished Name (DN).
LDAP_ENCODED	Indicates LDAP encoding.
LDAP_FILTER_ENCODE	Indicates that encoding used is suitable for a LDAP filter query

Neutral Taint Flag ID	Description
	argument.
LOCATION	Indicates that the information is location-related, such as GPS coordinates.
NO_NEW_LINE	Indicates no newline.
NOT_NULL_TERM_ TRUNCATE	Indicates that data is not null-terminated because of being truncated.
NOT_NULL_ TERMINATED	Indicates that data is missing a null terminator.
NULL_TERMINATED	Indicates a null-terminated string.
NUMBER	Indicates a numeric value.
OTHER_ENCODE	Indicates that data is encoded with an encoding that does not have a specific taint. This flag is also used for partial encodings that are very specific to an API.
PRIMARY_KEY	Indicates a primary key within a database. Affects the severity of `Access Control: Database` issues.
PROCESSOUTPUT	Indicates output from a process that is user-controlled.
RCONCATENATED	Indicates a right-concatenated string.
SALT	Indicates a salt. Used in cryptographic issues related to salts and salting.
SCRIPT_ENCODE	Indicates script (such as JavaScript or VBScript) encoding.
STRING_LENGTH	Indicates string length.
TAINTED_EXPECTED_ TYPE	Indicates that a user can control the expected types to be deserialized.
UNICODE_ENCODE	Indicates Unicode encoding.
URL_ENCODE	Indicates URL encoded data.

Neutral Taint Flag ID	Description
VALIDATED_ *<category>*_ *<subcategory>*	For each sink, there is a specific taint to check whether the dataflow validates against it. These taint flag identifiers use a naming convention of `VALIDATED_<category>_<subcategory>`. Any parentheses, hyphens, and spaces in the category and subcategory names are converted to underscores. For example, the VALIDATED flag against `Cross-Site Scripting: Reflected` is `VALIDATED_CROSS_SITE_SCRIPTING_REFLECTED`. The VALIDATED flag against `Log Forging (debug)` is `VALIDATED_LOG_FORGING__DEBUG_` (note the two underscores before `DEBUG`).
WEAKCRYPTO	Indicates data that was previously Base64 encoded. Used to identify when passwords used were just Base64 encoded (`Password Management: Weak Cryptography`).
XML_ATTR_ENCODE	Indicates encoded for use as an XML attribute.
XML_ENCODE	Indicates encoded for use in XML.
XPATH_ENCODE	Indicates encoded for use within an XPath expression.

Appendix B: Structural Rules Language Reference

This section contains the following topics:

Structural Syntax and Grammar 143
Types 144
Reference Resolution 146
Null Resolutions 146
Relations 147
Results Reporting 148
Call Graph Reachability 149

Structural Syntax and Grammar

The following is a simplified BNF-style grammar for the structural tree query language. Note that for readability purposes it is sometimes more and sometimes less strict than the actual grammar.

The following shows the structural tree query language:

```
<Rule> := <Label> <Expression>

<Label> := <TypeName> [ <Identifier> ] ':'

<Expression> := <Literal> | <Reference> | <RelationExpression> | 'not'
<Expression> | <Expression> 'and' <Expression> | <Expression> 'or'
<Expression> | '(' <Expression> ')'

<Reference> := [ <Reference> '.' ] <Identifier>

<RelationExpression> := [ <Reference> | <Literal> ] <Relation> (
<Reference> | <Literal> | <SubRule> )

<Relation> := 'is' | 'in' | 'contains' | 'reachedBy' | 'reaches' | '===' |
'==' | '!=' | '<=' | '>=' | '<' | '>' | 'startsWith' | 'endsWith' |
'matches'

<SubRule> := '[' [ <Label> ] <Expression> ']' [ '*' ]
```

```
<Literal> := 'true' | 'false' | <StringLiteral> | <NumberLiteral> |
<TypeSignatureLiteral>

<StringLiteral> := '"' <Text> '"'

<NumberLiteral> := ('0'-'9')+

<TypeSignatureLiteral> := 'T' '"' <Text> '"
```

Types

The rules language is strongly typed. Types in the rules language are called structural types to distinguish them from the language types of the source language. The types are organized into a hierarchy with source code constructs organized under the *Construct* base. Every type inherits the properties of each of its ancestors.

Each property has a fixed resolution type. As a result, the structural type of every subexpression in the rules language is known during rules specification. Static type-checking is performed when a rule is loaded.

For a full reference for the structural type hierarchy, see the *Fortify Structural Type and Properties Reference*. This information is included in the ZIP file from which you extracted this document.

The structural language also supports lists of objects. These objects do not have official type names. This means that they cannot appear as the subject of a rule. However, properties can still resolve to lists. The analyzer can access lists using the `contains` and `in` relations, just like constructs. For example, the `Function` construct has a property `paramaterTypes` that returns a list of `Type` objects.

The following rule matches functions that have any parameter of type `int`:

```
Function f: f.parameterTypes contains [Type t: t.name = "int"]
```

Interpret this rule as the following query: Select any function `f` from the structure of the program where the parameters of type `f` contain any type of `int`.

You can also reference with zero-based index notation, using standard, bracketed accessors.

The following rule matches functions where the first parameter has type `int`:

```
Function: parameterTypes[0] == T"int"
```

The `T"..."` syntax denotes a special type of constant in the structural language. It provides a convenient way to inspect language types. When the structural evaluator encounters such a constant, it converts the string between the quotes to a structural TypeSignature object (which is comparable with Type) using the rules of the source code language being examined (Java, C, and so on).

To match a nested type, use the dot notation (`OuterType.NestedType`). Below is a Java code example:

```
package com.example;

class Outer {
  class Nested {
    public void foo() {
      ...
    }
  }
}
```

The following rule matches a function declared inside the `Nested` class for the previous Java code example:

```
Function f: f.enclosingClass.name == "com.example.Outer.Nested"
```

The following example shows an equivalent rule:

```
Function f: f.enclosingClass == T"com.example.Outer.Nested"
```

Properties

The *Fortify Structural Type and Properties Reference* provides a list of all properties recognized by the structural analyzer. All structural types, including lists and primitive structural types, have associated properties. Every type inherits the properties of each of its ancestors. List types have only one property, `length`, which represents the number of items in the list.

Properties often resolve to subtypes of their declared types. The following is a Java code example:

```
x = 30;
```

This translates to an `AssignmentStatement` in the structural tree.

In the structural rules language, you can examine an assignment's right-hand side using the property `AssignmentStatement.rhs`, which nominally resolves to an `Expression`. In this case, it resolves to an `IntegerLiteral`, a subtype of `Literal`, which is itself a subtype of `Expression`.

The following example rule matches every assignment where the right-hand side has the expression of type `int`:

```
AssignmentStatement a: a.rhs.type == T"int"
```

You can use this rule because type is a property of all `Expression` objects. However, if you want to match every assignment, where the right-hand side is the integer literal `30`, you must cast `AssignmentStatement.rhs` using a subrule.

The following example subrule casts an `AssignmentStatement.rhs`:

```
AssignmentStatement a: a.rhs is [IntegerLiteral n: n.value == 30]
```

This is because value is not a property of `Expression`. To maintain type-safety, you must assert that `rhs` actually is an `IntegerLiteral` before you can access the property value.

Reference Resolution

A Reference (see "Structural Syntax and Grammar" on page 143) is an Identifier or chain of identifiers connected by dots, which resolves to a labeled object or a property of an object. Resolution of the first identifier follows the rules described here. Subsequent identifiers in the reference are always properties of the inner object.

To resolve the first identifier *ident* in a reference, the structural evaluator first checks to see if *ident* appears in a Label in the enclosing subrule, in a parent subrule, or in the initial label that starts the rule.

The following rule shows that `f` and `v` are resolved by examining the labels for the enclosing context:

```
Function f:
     f contains
          [Variable v: v.name == f.name]
```

In the case that *ident* does not resolve to a labeled object, *ident* is resolved as a property of the object selected by the immediately enclosing subrule (or the rule itself if ident does not appear in a subrule).

In the following example, name resolves in both cases to the name of the function:

```
Example 1: Function: name == "func"
Example 2: Variable v: v in [Function: name == "func"]
```

Null Resolutions

Some properties are valid only for certain instances of a structural type. For example, `TryBlock` has a property, `finallyBlock`, that resolves to the associated finally block of a try block. However, not all try blocks have associated finally blocks.

In these cases, properties resolve to null. There is no need for rules to check for this, because the Structural Analyzer handles operations on null in a well-defined manner:

- Every property of null resolves to null
- Every subrule relation on a null object resolves to false

The following shows how Boolean connectives resolve:

```
null and null -> null
null or null -> null
null and true -> null
null or true -> true
null and false -> false
null or false -> null
```

If the boolean value is determinate, it is resolved; otherwise it is null.

Relations

You can use the equality and inequality relations, == and != , to compare any two objects recognized by the Structural Analyzer. For equality to hold, the structural types of the objects must agree. Equality has the obvious meaning for primitive structural types; for constructs, the condition is that the two objects must be structurally identical.

The Structural Analyzer confirms the structural identity in one of two ways:

- Declarations are confirmed by comparing the canonical names of the symbols.
- Other constructs are confirmed by comparing the underlying nodes in the program representation

  Lists are equal if they enumerate equal elements in the same order.

The strict equality relation, === is true only if the objects being compared are the same object.

The order relations, < , > , <= , and >= have their usual meanings for strings, numbers, and booleans. You cannot compare types, lists, and constructs with order relations.

There are several special relations:

- `is` means the same thing as == except that you can use it to preface a subrule.
- You can use `in` and `contains` with strings and lists. For other constructs, these relations examine parent and child relationships. The `in` relation searches the parent and grandparents of the node to the top of the tree. The `contains` relation searches the children and *normally* the grandchildren of the node to the bottom of the tree. The exception to this behavior is for the *Class* and *CompilationUnit* structural types, for which `contains` only examines the first generation of children (this prevents writing queries that are unreasonably expensive to execute).
- You can only use `startsWith`, `endsWith`, and `matches` to relate two strings. The `matches` relation interprets the right-hand side of the relation as a Java regular expression, and it is true only if the left-hand side is matched by that regular expression.
- You can only use `reaches` and `reachedBy` to relate two `Functions` or two `Classes`. See "Call Graph Reachability" on page 149 for more information.

You can omit the left-hand side of any of these relations. If you omit it, the left-hand side defaults to the construct that the rule is currently matching.

The following example rule matches any class that has a proper superclass:

```
Class c:
     c.supers contains
          [Class c2: c2 != c]
```

Because `supers` resolves to a `Class[]`, you can abbreviate the previous rule to the following:

```
Class c: supers contains [!= c]
```

Although this rule is more compact, the first example is clearer and easier to read.

Results Reporting

Recall the following example, which matches return statements that appear inside a finally block:

```
ReturnStatement r: r in [FinallyBlock:]
```

The following rule is similar:

```
FinallyBlock f: f contains [ReturnStatement:]
```

However, there are two significant differences. First, if a single finally block contains multiple return statements, the first rule generates multiple vulnerabilities while the second rule produces just one.

The second difference is the way in which the rules report vulnerabilities. The primary source location, as reported in the analysis output, always points to the rule's outermost construct. The first rule highlights the return statement. The second rule highlights the block.

By default, the Structural Analyzer reports no information other than the source location of the outermost construct that it matches. For some rules, this is sufficient. Other rules require more information to create a complete report.

You can enable reporting for a subrule by appending an asterisk to the subrule:

```
ReturnStatement: in [FinallyBlock:]*
```

This rule is logically equivalent to the un-asterisked rule because it matches exactly the same code constructs. However, when the Structural Analyzer matches it, both the return statement and its enclosing finally block are reported. The return statement is still the primary reporting location.

The Structural Analyzer only reports asterisked subrule matches for subrules that contribute to a match. The following subrule shows this.

```
Function: contains [AssignmentStatement:]* and public or
contains [ReturnStatement:]* and private
```

This rule matches any public method containing an assignment statement, or any private method containing a return statement. The Structural Analyzer always reports the matching statement, because both subrules are asterisked. However, if a method contains both an assignment statement and a return statement, the analyzer reports as follows:

- Assignment statement—If the method is public
- Return statement—If the return statement of the method is private

Call Graph Reachability

Many structural rules apply only in certain contexts. For example, Enterprise JavaBeans (EJBs) should never call the `java.io` libraries. You can implement a rule that matches every call to `java.io`.

The following rule matches every call to `java.io`:

```
FunctionCall call:
     call.function.enclosingClass.name startsWith "java.io."
```

The issue with this rule is that it generates many false positives. This is because most calls to `java.io` do not involve EJBs. A better recommendation is to restrict the rule to function calls that appear within an EnterpriseBean. The enclosing class of the function call differs from the enclosing class of the function.

The following rule has an EnterpriseBean restriction:

```
FunctionCall call:
     call.enclosingClass.supers contains
           [Class c: c.name == "javax.ejb.EnterpriseBean"]
     and
     // The enclosing class of the function itself
     call.function.enclosingClass.name startsWith "java.io."
```

This improved rule misses cases in which an EJB indirectly calls `java.io`. For example, this rule misses when an EJB calls a utility method in a different class, and the utility method opens a file. This should be a violation. The Structural Analyzer provides two relations: `reaches` and `reachedBy` that traverse the call graph of a program. You can use these relations to handle this type of situation.

The following example shows a reaches relation:

```
f reaches [subrule]
```

This is true just if there is some path through the call graph originating with `f` and terminating at a function that matches the subrule. `reachedBy` is similar, with the path proceeding in the opposite direction.

The following example shows a `FunctionCall` that is the best way to encode the previous EJB rule:

```
FunctionCall call:
     call.enclosingClass.supers contains
     [Class: name == "javax.ejb.EnterpriseBean"]
     and
     call.function reaches
          [Function fnReached:
               fnReached.enclosingClass.name startsWith "java.io."]*
```

You can also use the `reaches` and `reachedBy` relations on classes. Class A reaches class B if some function of A reaches some function of B. For example, the following rule matches public fields in classes that an Applet can reach:

```
Field f:
     f.public and not f.final
     and f.enclosingClass reachedBy
          [Class a: a.supers contains
               [Class super: super.name == "java.applet.Applet"]]
```

A field cannot appear as part of a `reachedBy` relation. Only functions and classes can satisfy the `reaches` or `reachedBy` relation. For performance reasons, variable scopes do not extend across `reaches` or `reachedBy` predicates.

The following is an illegal rule:

```
Function f: reaches [Function g: g != f]
```

The variable `f` cannot appear in the subrule of a `reaches` relation.

Appendix C: Control Flow Rule Reference

This section contains the following topics:

Syntax and Grammar 151
Control Flow Rules 152

Syntax and Grammar

The following is a simplified BNF-style grammar for the Control Flow Rule Language. For readability, the grammar in this guide is stricter than it is in practice.

The following shows the Control Flow Rule Language:

```
<MachineSpecification> := <Declaration>* <Transition>*
<Declaration> := <StateDeclaration> | <PatternDeclaration> |
<VariableDeclaration>
<StateDeclaration> := 'state' <StateName> [ '(start)' | '(error)' ] ';'
<StateName> := <Identifier>
<PatternDeclaration> := 'pattern' <Identifier> '{' <StatementList> '}'
<VariableDeclaration> := 'var' <Identifier> ';'
<Transition> := <StateName> '->' <StateName> '{' <StatementList> '}'
<StatementList> := <Statement> [ '|' <StatementList> ]
<Statement> := <PatternUse> | <MetaFunction> | <Declaration> |
<AssignmentStatement> | <Expression>
<PatternUse> := 'pattern' <Identifier>
<MetaFunction> := '#end_scope' '(' <RuleVariable> ')'
     | '#end_function' '(' ')'
     | '#return' '(' [ <Expression> ] ')'
     | '#compare' '(' <RuleVariable> ',' ( <Literal> | <Wildcard> ) ')'
     | '#param' '(' <RuleVariable> ',' ( <Wildcard> | <NumberLiteral> ) ')'
     | '#ifblock' '(' <RuleVariable><IfBlockComparisonOperator> ( <Literal>
| <Wildcard> ) ',' ( 'true' | 'false' ) ')'
<IfBlockComparisonOperator> := '==' | '!=' | '<' | '<=' | '>' | '>='
<Declaration> := ( '#any_declaration' | '#simple_declaration' | '#complex_
declaration' | '#buffer_declaration' ) '(' <RuleVariable> ')'
<AssignmentStatement> := ( <RuleVariable> | <Wildcard> | <OpExp> ) '='
<Expression>
<Expression> := ( <Literal> | <OpExp> | <Call> | <QualifiedCall> |
<Wildcard> | <RuleVariable> )
<Literal> := <StringLiteral> | <NumberLiteral> | 'true' | 'false' | 'null'
```

```
<StringLiteral> := '"' <Text> '"'
<NumberLiteral> := ('0'-'9')+
<OpExp> := '&' <Expression> | '*' <Expression>
<RuleVariable> := <Identifier>
<Wildcard> := '?'
<QualifiedCall> := ( <RuleVariable> | <Wildcard> ) '.' <Call>
<Call> := ( <Identifier> | '#any_function' ) '(' [ <ArgumentList> ] ')'
<ArgumentList> := ( <Argument> [ ',' <ArgumentList> ] ) | '...'
<Argument> := [ '...' ',' ] <Expression>
```

Control Flow Rules

Control flow rules provide definitions of state machines that characterize unsafe behavior such as potentially dangerous sequences of operations.

Control Flow Rule Identifiers

Control flow rules can have multiple function identifiers. The function identifiers are used in the control flow definition. The definition uses the value of the reference identifier as a variable to access the function identifiers. See "XML Representation of Control Flow Analyzer Rules" on page 94 for descriptions of most of the control flow function identifiers. The function identifier for control flow rules also contains additional fields and functionality, described in this section.

Control Flow Rule Format

Unlike dataflow rules, a control flow rule does not specify a single function; instead, it specifies a sequence of program elements (function calls or other entities in a program). This definition, which goes in the `<Definition>` element of the rule, resembles a simple programming language.

Control flow rules support the following C++ and Java-style comments:

- // creates a comment to the end of the line
- /* creates a comment until a matching */

Each rule definition defines a state machine. Each state machine has exactly one start state, one or more error states, and any number of intermediate states. The machine always has a current state.

When the current state is an error state, the Control Flow Analyzer reports a vulnerability.

States are connected by transitions. Each transition has a source state, a destination state, and some number of patterns. If a transition's source state is the current state and one of that transition's patterns matches a fragment of the program, then the transition's destination state becomes the new current state. In this case, the machine is said to have transitioned from the source state to the destination state. The program fragment is referred to as the "input" to the pattern. The definition of a machine consists of two major parts: declarations and transitions.

This section contains the following topics:

Declarations 153
Transitions 153
Function Calls 156

Declarations

Machine definitions begin with declarations of the states of the machine. States are defined with the `state` keyword, followed by the state name, and optionally followed by `start` or `error` to designate the start and error states, respectively. A simple machine can have the following state definitions:

```
state state1 (start);
state state2;
state state3 (error);
```

Machines can also include variables, which are declared with the `var` keyword. A variable can match any expression in the program. At the first use of a variable, it is bound to the expression it matches. For subsequent uses of the same variable, the variable only matches if the input is the same as the expression to which the variable is bound.

The following shows a sample declaration:

```
var f;
```

Finally, you can give names to patterns to avoid entering the same pattern multiple times. Specify a name for the pattern with the `pattern` keyword, followed by the pattern enclosed in braces.

For example, the following line declares a pattern named `alloc`, that matches the `malloc` and `calloc` functions:

```
pattern alloc { malloc(...) | calloc(...) }
```

For more information on patterns, see "Transitions" below.

If a control flow rule contains a line of the form `limit <refid>;`, then that control flow rule only applies in the body of functions that match the function identifier with reference ID `<refid>`.

Transitions

Transitions define how the current state of the machine might change. As described in "Declarations" above, each transition has a source state, a destination state, and a pattern. You can have multiple transitions with the same source state; in this case, the new current state is the destination state of the first transition with a pattern that matches the input.

Transitions are defined by the name of the source state, the symbol `->`, the name of the destination state, and one or more patterns enclosed in braces. Separate multiple patterns in the same transition with the `|` character.

The following is an example of a transition with multiple patterns separated by the | character:

```
source -> destination { pattern1 | pattern2 }
```

A pattern consists of one of the following elements:

- Uses of a named pattern

  You can use patterns declared with the pattern keyword in the declaration section in transitions by specifying the pattern keyword followed by the pattern name, such as: state1 -> state2 { pattern alloc }.
- Assignment statements

  Control flow rules often refer to the return values of function calls, particularly object constructors and other functions that return handles to resources. You can match the return value of a function or any assignment statement with the name of a rule variable followed by the equal (=) symbol and an expression (see "Expressions" below) The left-hand side of the assignment operator must be a previously declared rule variable.
- Expressions

  An expression can be any one of the following:

  - A string, enclosed in double-quotes (C-style)
  - A character, in single-quotes (C-style)
  - An integer
  - A floating-point number
  - A boolean "true" and "false" (without quotes)
  - The value "null" (without quotes)
  - *<Expression>: A dereference of <Expression>
  - &<Expression>: A reference to <Expression> (C-style)
  - A function call (see "Function Calls" on page 156)
  - A ? character: Matches any expression in the input
  - The name of a rule variable: If the rule variable is unbound, matches any expression and binds the rule variable to that expression. If the rule variable is bound, matches the expression to which the variable was first bound.
- Language feature statements

  Some aspects of programs cannot be represented using the expressions. For these aspects, there are special types of patterns. These patterns resemble function calls in C or Java, but all the

function names begin with a # character.

The following table describes the valid language feature statements.

Statement	Description
`#end_scope(var)`	Matches the end of the enclosing scope for the expression bound to the rule variable `var`.
`#return(expr)`	Matches a return statement with a return expression matching `expr`.
`#return()`	Matches any return statement.
`#compare(var, const)`	Matches a comparison `(==, !=, <, >, <=, >=)` between `var` (a rule variable) and `const` (a string, character, integer, floating-point number, boolean, null, or '?' expression).
`#simple_declaration (var)`	Matches the declaration of a simple type—An integer, pointer, reference, or other primitive data type. Binds the rule variable `var` to the variable declared in the program.
`#declaration(var)`	Is identical to `#simple_declaration(var)`.
`#complex_declaration (var)`	Matches the declaration of a complex data type (struct or object) in C or C++. Pointers to structs, pointers and references to C++ objects, and references to Java objects are not matched; use the `#simple_declaration` pattern for these data types.
`#buffer_declaration (var)`	Matches the declaration of a stack buffer in C or C++.
`#any_declaration (var)`	Matches any of the preceeding.
`#ifblock (var, const, which)`	Matches a comparison between `var` and `const` as defined for `#compare`, with the additional restrictions that the comparison operator must be an equality test (`==`, `!=`, or a similar operator), and that the comparison must occur within the predicate of a branching or looping construct (such as if statements, for loops, and while loops). The specified state transition only occurs on the branch where `var == const` evaluates to `which`.

Function Calls

Most interesting security properties involve the use of function matching syntax based on function identifiers. Control flow rules use the reference ID field from function identifiers to specify functions for transitions. For example, if there is a function identifier with a reference ID of `allocator`, then the control flow pattern `v = $allocator(?)` would assign the rule variable `v` to the return value of any function that matched the `$allocator` function identifier and took exactly one argument.

In general, the arguments to the rule function should exactly match the expected arguments to the input function. Therefore, to write a rule that binds the second argument to the link system call to the rule variable var, the rule would read `$link(?, var)`, assuming a function identifier matching the link system call had already been defined with a reference ID of link. There is one exception to the "one expression per argument" rule: an ellipsis (...) in the arguments to a function matches zero or more expressions. It is therefore possible to match the last argument of a function by specifying `function(..., var)`, `function(...)`, and `$function(...,var,...)` matches any invocation of the specified function, without paying attention to the arguments to that function.

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email.

Note: If you are experiencing a technical issue with our product, do not email the documentation team. Instead, contact Micro Focus Fortify Customer Support at https://www.microfocus.com/support so they can assist you.

If an email client is configured on this computer, click the link above to contact the documentation team and an email window opens with the following information in the subject line:

Feedback on Custom Rules Guide (Fortify Static Code Analyzer 23.1.0)

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 fortifydocteam@microfocus.com.

We appreciate your feedback!