SecureAssist Custom Rule Tutorial

October 2014

www.Cigital.com
North America: +1 800 824 0022
EMEA: +44 203 427 6601
support@cigital.com

Introduction

Cigital SecureAssist is a lightweight IDE plugin that points out common security vulnerabilities in real time as the developer is coding. The detected vulnerabilities are identified by analyzing source code against a set of rules. Cigital provides a number of these rules in rulepacks and updates them regularly for active customers. However, in many cases, an organization may wish to scan for issues not included in the default Cigital rulepacks. For example, a rule enforcing a company's specific cryptography standards could be used in addition to the standard SecureAssist cryptography rules.

This tutorial walks you through the process of creating a custom SecureAssist rule. Beginning with a code sample containing a security vulnerability, you will create a rule to detect that particular vulnerability. This includes the rule itself, propagator, source, and filter to locate the vulnerability and provide remediation advice whenever the vulnerability is detected in code.

Intended Audience

Developing rules requires an understanding of the SecureAssist rule schema, a basic knowledge of regular expressions (regex,) and a programming background. Deciding what the rule itself entails (what code needs to be flagged, what vulnerability the rule is for, and user guidance to help remediate the vulnerability) requires an understanding of software security.

The Example Code

```
package com.cigital.rulepack.example;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.servlet.http.HttpServletRequest;

public class UserDao {

 public User getUser(HttpServletRequest request) {

        User user = null;
        StringBuffer query = new StringBuffer();
        query.append("SELECT * FROM users WHERE id=");
        query.append(request.getParameter("userid"));

        try {
               String connStr = "jdbc:msql://10.10.10.1:1114/Demo";
               Connection conn = DriverManager.getConnection(connStr, "", "");
               Statement stmt = conn.createStatement();
               ResultSet rs = stmt.executeQuery(query.toString());
               user = new User(rs.getString(0), rs.getString(1));
        }
        catch (SQLException e) { }

        return user;
 }
```

```
class User {
        public User(String name, String title) {

        }
}
}
```

This method returns some information about a User from the database when given a user ID from a request parameter. To get the user information from the database, it takes an HttpServletRequest as an argument, retrieves the value of the parameter named “userid,” links this value to a query String, and then executes this query on the database.

This method is insecure because it exposes an SQL injection vulnerability. A parameter may contain any arbitrary character, including those that are syntactically significant to SQL. For example, a malicious attacker can pass the parameter ‘1; drop table users;’.

The vulnerability is comprised of these elements, on the subsequent lines of code:

- **The Sink at which the vulnerability occurs.** On line 24, the call to executeQuery will perform a database query on an unvalidated query String. In SecureAssist, these are described as **Rules**.
- **An entry point in the code for the untrusted data.** The HttpServletRequest argument on line 13 is an instance of this. In SecureAssist, these are described as **Sources**.
- **The means by which the tainted data in the Source arrives at the Sink**. On line 18, the tainted data is linked to a StringBuffer, so SecureAssist must be aware that the StringBuffer now contains the taint. Likewise, on line 24, SecureAssist must be aware that the StringBuffer#toString call propagates the taint from the StringBuffer into the returned String. In SecureAssist, these are described as **Propagators**.

Rules also require filters and guidance:

- A **Filter** organizes rules into categories. A Rule must be added to a filter and activated for SecureAssist to display the Rule’s results.
- **Guidance** provides advice to a SecureAssist user about how to fix the vulnerability. When code triggers a finding, guidance supplies the developer with the information necessary to understand and remediate the vulnerability.

In this example, you will be writing all of these components of a custom rule. In practice, this is not always necessary. SecureAssist provides an extensive library of components available for reuse, and custom rule developers often only need to add a new Rule or a new Source.

Working with the Rulepack Configurator

The Cigital SecureAssist Rulepack Configurator is an application for creating and modifying custom SecureAssist rules.

For this tutorial, we will modify the default SecureAssist rulepack. To begin:

- Login to the SecureAssist Enterprise Portal and navigate to the Rulepack page.
- Click on “CSA Default Rulepack” (version 2.3 or later) and select a location to download the JAR file.
- Launch the SecureAssist Rulepack Configurator and select ‘File > Open Rulepack’ and navigate to the JAR file downloaded from the Portal.

After you’ve made changes to a rulepack, you can save these changes by choosing ‘File > Save Rulepack’. If any of your custom elements are malformed, you will be prompted to fix them before the Configurator will allow you to save your rulepack.

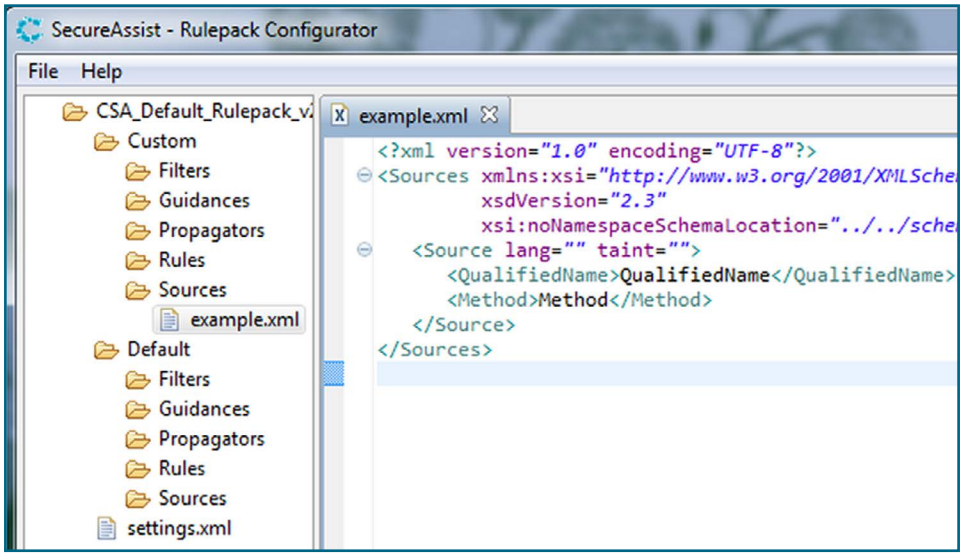
The contents of a rulepack are organized into two directories: Custom and Default. Default elements are provided with SecureAssist and cannot be modified. The Custom directory provides a location for users to create custom Rules, Sources, Propagators, Filters, and Guidances.

For more detailed instructions, please refer to the Rulepack Configurator User Guide.

Writing a Source

A Source informs SecureAssist where untrusted data can potentially enter an application. When SecureAssist finds a Source, it is marked as such, and used to determine which lines of code are insecure during taint flow analysis.

To begin, create a new Source in the configurator by right clicking the Custom > Sources directory, select New > Source and name your Source 'example.' This will create an XML file with a Source that contains the required attributes and elements pre-generated as shown below.



This Source has two attributes (lang and taint) and two child elements (QualifiedName and Method).

The **lang** attribute describes the programming language to which this Source applies. In our case, this applies to Java source code, so we will use 'java.

The **taint** attribute describes the kind of untrusted data that the Source can introduce. Taint can be of type **WEB** if it originates via HTTP protocol, **DB** if it originates from a database, **PRIVATE** if it is passed to the application as private data, or **FILE** if it comes from a file. Since this taint originates from an HttpServletRequest, we will set the value to WEB.

The **QualifiedName** element specifies the fully qualified name of the Object responsible for introducing the taint. This can be expressed as either an exact text match or a Java regular expression. (A reference to valid regular expressions can be found at http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html)

The **Method** element specifies the name of the method responsible for introducing the taint. If taint is introduced by a constructor, there will be no Method element. This can be expressed as either an exact text match or a Java regular expression.

Our Source is complete and appears as follows:

```
<Sources>
  <Source lang="java" taint="WEB">
 <QualifiedName>javax.servlet.HttpServletRequest</QualifiedName>
    <Method>getParameter</Method>
  </Source>
</Sources>
```

Note: Enhancing Sources

The previous Source can be made more general, applicable to more situations where untrusted data may enter code. We can make the source more useful in two ways:

First, this Source only applies to HttpServletRequest. Because Java allows inheritance, a class may extend from HttpServletRequest and inherit the getParameter method, which is capable of introducing untrusted data. This Source would be better if it were to apply to any of those classes as well. This is done by adding the extends attribute to the QualifiedName element and setting its value to 'true'.

```
<Sources>
  <Source lang="java" taint="WEB">
 <QualifiedName extends="true">javax.servlet.HttpServletRequest</QualifiedName>
  <Method>getParameter</Method>
  </Source>
</Sources>
```

The Source will now locate untrusted data from objects of or extending from HttpServletRequest.

Second, getParameter is not the only method in the HttpServletRequest API capable of introducing arbitrary and potentially dangerous data into source code. We will write our Method as a simple regular expression that matches each of these potential sources of taint.

```
<Sources>
  <Source lang="java" taint="WEB">
 <QualifiedName extends="true">javax.servlet.HttpServletRequest</QualifiedName>

 <Method>(getRemoteHost|getParameterValues|getParameterNames|getParameterMap|getHeader
 |getQueryString|getRemoteUser|getRequestedSessionId|getRequestURI|getRequestURL
 |getContentType|getCookies|getHeaders|getLocalName|getParameter)</Method>
  </Source>
</Sources>
```

Our source is now more versatile and will find additional points of untrusted data. While writing Sources, Rules, and Propagators, it is important to consider whether they can be enhanced to enable SecureAssist to locate more vulnerabilities.

For a comprehensive guide to Source features, please consult the Sources section of the Rulepack Configurator Users Guide.

Writing a Taint Flow Rule

SecureAssist allows users to write a variety of different kinds of rules. For this example we will write a taint flow rule. Taint flow rules describe conditions under which code is exploited using untrusted data.

Create a new rule file in the Java rules directory and the Configurator will provide the outline we will use to construct our taint flow rule.

example.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsdVersion="2.3"
       xsi:noNamespaceSchemaLocation="../../../schemas/java_ru
   <Rule id="" lang="">
      <Category>Category</Category>
      <Title>Title</Title>
      <Description>Description</Description>
      <Match/>
      <Standards>
         <Standard file=""/>
      </Standards>
   </Rule>
</Rules>
```

The Rule element has two attributes:'id' and 'lang'.

'id' represents the unique name by which this rule is identified in SecureAssist. We will give this rule the identifier SQL_001.

The **'lang'** attribute represents the programming language to which this rule applies, as it does for Sources. Similarly, we will give this attribute the value 'java'.

The Rule element has five children: Category, Title, Description, Match, and Standards.

Category allows like rules to be grouped together. Before creating an ad hoc category, check the default and custom filters to see if an existing one already suits your purpose. For our example, we will use the category 'Dynamic Database Query.'

Title provides a means to specify a human-readable identifier for the rule. We will name our rule 'SQL Injection.'

The **Standards** element is the means by which you can specify which guidance the developers will receive when SecureAssist finds a vulnerability. This contains the information necessary to understand and remediate the vulnerability, and will often include a description, a language-specific code sample of the vulnerability, a language-specific code sample of secure code, recommendations, and classifications of the vulnerability. We will reuse an existing Guidance, which resides in the default Standards folder within the file 'about-sql-injection.xml'. Additionally, we will provide the standard with a child node named **'Context'** and set its value to 'J2EE'. This will cause this Guidance to render Java code samples whenever Rule triggers.

```
<Rules>
  <Rule id="SQL_001" lang="java">
    <Category>Dynamic Database Query</Category>
    <Title>SQL Injection</Title>
    <Description>Identifies dangerous method calls of the java.sql.Statement class.</
Description>
    <Match />
```

```
    <Standard file="about-sql-injection.xml">
      <Context>J2EE</Context>
    </Standard>
  </Rule>
</Rules>
```

Our rule has all of its metadata defined, but still requires us to fill out its Match element. Unlike the metadata and guidance we've described, the Match element supplies the condition upon which the rule will fire.

First, we will define a QualifiedName and Method for our rule, using the same elements as we did when defining a Source.

```
<Rules>
  <Rule id="SQL_001" lang="java">
    <Category>Dynamic Database Query</Category>
    <Title>SQL Injection</Title>
    <Description>Identifies dangerous method calls of the java.sql.Statement class.</
Description>
    <Match>
      <QualifiedName extends="true">java.sql.Statement</QualifiedName>
    <Method>executeQuery</Method>
    </Match>
    <Standard file="about-sql-injection.xml">
      <Context>J2EE</Context>
    </Standard>
  </Rule>
</Rules>
```

Because we are only interested in instances where the first argument of calls to java.sql.Statement#executeQuery contains taint, we will specify an Arguments condition in the Match where this is the case. This is done by giving **Arguments** an **Argument** child, with an **Index** and **Taint** child.

Index should contain the index of interest, which in our example is 0. Taint should contain the child Type where we specify the taint type we are interested in tracking. When we specified a Taint type in our Source, we used the value WEB. However, it may be advantageous to fire this rule when a possible SQL injection attack comes from a file or database as well. Accordingly, we will use the special value 'UNTRUSTED', which will cause the rule to match tainted data from files, the web, or databases.

Our completed rule appears as follows:

```
<Rules>
  <Rule id="SQL_001" lang="java">
    <Category>Dynamic Database Query</Category>
    <Title>SQL Injection</Title>
    <Description>Identifies dangerous method calls of the java.sql.Statement class.</
Description>
    <Match>
      <QualifiedName extends="true">java.sql.Statement</QualifiedName>
      <Method>executeQuery</Method>
      <Arguments>
        <Argument>
          <Index>0</Index>
          <Taint>
            <Type>UNTRUSTED</Type>
          </Taint>
          <Argument>
        </Arguments>
    </Match>
    <Standard file="about-sql-injection.xml">
```

```
        <Context>J2EE</Context>
      </Standard>
    </Rule>
  </Rules>
```

This is the complete rule; however our rule writing syntax is flexible and offers several options. Please consult the Rules section of the Custom Rulepack Users Guide for a comprehensive description of CSA's rule writing capabilities.

In this example, we wrote a taint flow rule to determine where untrusted data is used to exploit insecure code. SecureAssist supports a variety of other types of rules as well. See the Configurator Users Guide for comprehensive details on SecureAssist's rule writing capabilities.

Writing a Filter

The rule that we wrote is disabled by default and must be enabled. This is done by creating or modifying a filter.

Create a filter in the custom filters section of the rulepack.

example-Custom_v1405546815672.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Filter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        desc=""
        ide="eclipse"
        name="example"
        version="1405546815672"
        xsdVersion="1.2"
        xsi:noNamespaceSchemaLocation="../../schemas/filter_1_2.xsd">
   <Problems/>
</Filter>
```

The Problems section is blank, so we must add the problem that we are attempting to solve, which is SQL Injection. We do so by creating a Problem element with the name 'SQL Injection'.

```
<Filter desc="" name="" version="" ide="">
  <Problems>
    <Problem name="SQL Injection">
    </Problem>
  </Problems>
</Filter>
```

Next, we add a **Rules** section to the **Problem** element and assign it a **Rule**.

This Rule will have four attributes.

The **'id'** attribute should be assigned the unique identifier given to the rule. Ours is 'SQL_001'.

The **'active'** attribute specifies whether the rule should fire or not and, in our case, will be given the value 'true'.

The **'importance'** attribute describes the importance of the rule relative to other rules and can be given the value 'HIGH', 'MEDIUM', or 'LOW. Since SQL injection is a serious vulnerability, we will assign this rule the value 'HIGH'.

'markerShow' informs SecureAssist whether to represent this Rule's findings as markers within open editor windows. We will give this the value 'true'.

Our filter is now complete:

```
<Filter desc="" name="" version="" ide="">
  <Problems>
    <Problem name="SQL Injection">
      <Rules>
   <Rule id="SQL_001" active="true" importance="HIGH" markerShow="true"/>
 </Rules>
    </Problem>
  </Problems>
</Filter>
```

Writing Propogators

Propagators track how taint travels through a system and describe conditions on which taint is passed between objects.

In our example, taint from our source is linked to a StringBuffer, so SecureAssist must recognize that the StringBuffer is then tainted. This StringBuffer's taint is then passed into a String when StringBuffer's toString method is called. Likewise, this requires a propagator.

Writing the toString Propagator

To begin, create a new custom propagators file to hold our propagation rules. A PropagationRule must be given a language and identifier, much like a Rule or Source. Also, like a Rule or Source, a PropagationRule contains a QualifiedName and a Method. Since toString can transfer taint for many classes that are unrelated to StringBuffer, we will describe the QualifiedName as '.*', a regular expression that will match any class.

```
<PropagationRule ruleID="JAVA_TEST_PROPAGATOR_001" lang="java">
    <QualifiedName>.*</QualifiedName>
    <Method>^toString$</Method>
</PropagationRule>
```

A propagation rule must describe how taint flows through the class. In this case, we want to describe the following situation: "If an Object contains taint, and the toString method is called on this object, then the value this method call returns should also contain taint".

The transfer of taint is described by the Propagate element. Propagate's children describe where taint is propagating TO, and their children express where the taint is propagating FROM.

In our example, taint is propagating TO the return value of the toString method, so we will use a ReturnValue element. Taint is propagating FROM the object on which the toString method is being called, so we will add a Caller child with the value 'true'.

Our complete propagation rule appears as follows:

```
<PropagationRule ruleID="JAVA_TEST_PROPAGATOR_001" lang="java">
    <QualifiedName>.*</QualifiedName>
    <Method>^toString$</Method>
    <Propagate>
      <ReturnValue>
        <Caller>true</Caller>
      </ReturnValue>
    <Propagate>
</PropagationRule>
```

Writing the StringBuffer Propagator

In our code example, the untrusted data is linked to a StringBuffer. It is necessary to create a propagation rule that is capable of propagating taint into and out of a StringBuffer object.

We do not want this rule to match when the first index is a non-String. Accordingly, in addition to QualifiedName and Match, we will add an “Arguments” element, so that the StringBuffer only matches when it has a String as its first argument. This Arguments element will have one Argument element with two child elements: Type and Index.

Next, we will describe where taint originates from and propagates to. Our second propagator is only slightly more complicated than the first. Here, there are multiple ways taint is propagating. Ifa StringBuffer#append call has a tainted String as its first argument, then taint should propagate to the StringBuffer.

First, taint flows TO the StringBuffer itself FROM the argument at index 0 of the StringBuffer#append call. Additionally, taint flows TO the StringBuffer itself FROM the argument at index 0. This means the Propagate element must have an element for

Caller as well as ReturnValue, since taint can propagate to both.

Additionally, the return value of a tainted StringBuffer should be tainted even if the argument being appended to it is not. Taint flows TO the ReturnValue FROM the Caller, so a Caller element must be added to a ReturnValue element.

This can all be expressed in the same rule.

```
<PropagationRule ruleID=”JAVA_TEST_PROPAGATOR_002” lang=”java”>
    <QualifiedName>java.lang.StringBuffer</QualifiedName>
    <Method>^append$</Method>
    <Arguments>
      <Argument>
        <Type>java.lang.String</Type>
        <Index>0</Index>
      </Argument>
    </Arguments>
    <Propagate>
      <Caller>
        <Argument>0</Argument>
      </Caller>
      <ReturnValue>
        <Caller>true</Caller>
        <Argument>0</Argument>
      </ReturnValue>
    <Propagate>
</PropagationRule>
```

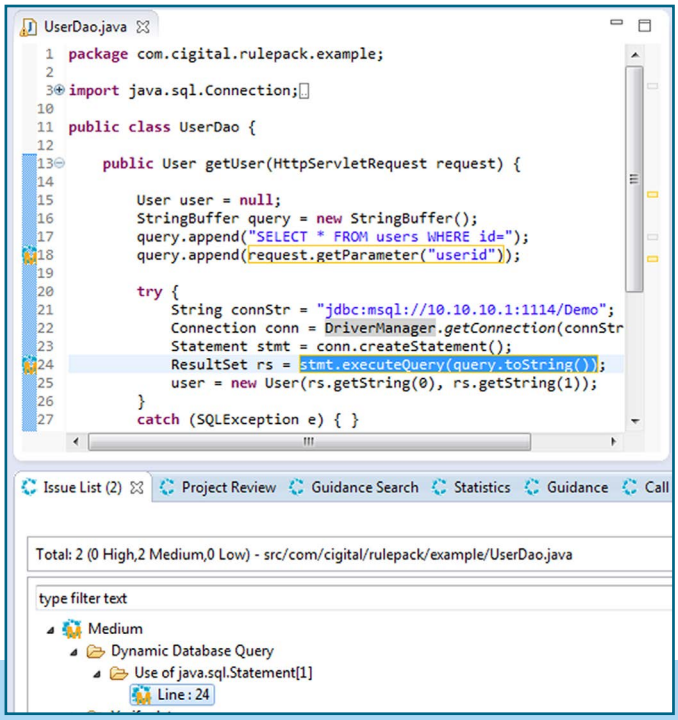
Deploying the Rulepack

Now that all elements are written, choose ‘File > Save Rulepack’ and provide the rulepack with a name and version number.

Upload your rulepack to the enterprise portal, set the rulepack to enabled, and restart your eclipse instance. Instructions on adding new rulepacks can be found in the Enterprise Portal User Guide available at http://www.cigital.com/products/support/.

When file scan is run on our example again, a result should appear on line 24.

This document discussed each of the components that are used to create custom SecureAssist elements. For a more detailed description of SecureAssist’s capabilities, consult the Custom Rulepack Users Guide.



About Cigital, Inc.

Cigital, Inc., a leading software security and quality consulting firm, was established in 1992 with a single focus of helping organizations improve software. Our consultants are thought leaders who specialize in programs that help organizations ensure their applications are secure and reliable while also improving how they build and deploy software. We provide advice across the enterprise using a combination of proven methodologies, tools, and best practices that are tuned to meet each client’s unique requirements. Cigital has enabled some of the most well-known organizations in financial services, communications, insurance, hospitality, e-commerce and government to reduce their mission-critical software business risks. Cigital is headquartered near Washington, D.C. with regional offices in the U.S., Europe and India.

For help, contact us:

SecureAssistHelp@cigital.com