A framework for creating custom rules for static analysis tools

Eric Dalci **John Steven**
Cigital Inc.
21351 Ridgetop Circle, Suite 400
Dulles VA 20166
(703) 404-9293
{edalci,jsteven}@cigital.com

Abstract

Code analysis tools have only a limited standard set of rule they enforce out of the box. Some code analysis tools have built in extension capabilities in the form of rule customization. Companies adopting code analysis tools gain their full benefit only through customization. This paper describes experiences with custom rules written for the Fortify Software Source Code Analyzer [1]. We explain a process that we followed to achieve reasonable accuracy and coverage. The process we describe is "tool agnostic;" we believe that it can be adopted for other code analysis tools as long as they offer a customization mechanism.

1- Introduction

Code analysis tools scan source code for implementation bugs without actually executing the source code, unlike penetration testing tools. In addition to enforcing a set of core rules out of the box some code analysis tools offer the capability to look for additional security vulnerabilities by writing custom rules. Every organization has its own specific corporate security standards. Every organization also possesses a wealth of incident data in its operations department. Both corporate standards and incident data represent essential custom rules to be created. Because of how organization-specific these data points are, tool vendors are very unlikely to create these rules as they iterate their tool. This paper introduces a testing framework for custom rules that we have created and used during a custom rule creation process for Java source code. This framework possesses three benefits. First, it improves the instances of a particular vulnerability a rule identifies, since a rule violation may appear with different code constructs within a source code—increasing true positive results. Second, the framework improves the accuracy of the rules—reducing false positives. Finally, this framework helps identify the limitations of the tool and provide new insights for the code reviews—identifying false negatives.

2- The need for custom rules

The majority of code analysis tools are based on rules that describe desirable or undesirable characteristics for a piece of software. Tool vendors will likely continue to provide updates to their set of predefined rules over time. For example the new rules may test for newly discovered software vulnerabilities. A company's own penetration testing and incident response data may be an excellent place to look for such new vulnerabilities. These breaches are pulled from one's own systems. There is automatically an applicability, feasibility, and high priority to detecting these same findings earlier.

As the tools' analysis capabilities become more mature, organizations expect more from them. Central to that expectation is customization and extension. Custom rules may be used to enforce corporate standards—which like incidents are to a certain extent necessarily different from organization to organization. Every organization seems to have a corporate standard for use and configuration of a particular set of strong cryptographic algorithms. It is unlikely that tool vendors will ever include such rules in their core set because they are not in the business of taking a stance on what is sufficient—like corporate security groups are.

3- Creating the rule, from the idea to the battle field

This section describes, step by step, our test-driven framework for creating custom code analysis rules.

Rules can be expressed at different levels of abstraction. They can be made so specific that they contain the exact code that constitutes a violation, but highly specific rules can be cumbersome and they may not even work well since the same vulnerability may manifest itself in many different ways, depending on who wrote the code.

At the opposite end of the spectrum, highly general rules may be violated by any suspicious use of a method. The typical example of this is a semantic rule flagging any occurrence of a potentially dangerous method, but without having constraints on the input and output parameters. An overly general rule typically causes an unmanageable number of false alarms (i.e., signaling security vulnerabilities where none exist). Such rules still require extensive human effort to ferret out the genuine issues and separate them from the false alarms.

In addition to that, there are different types of rules. Some rules simply look at simple semantics, and define a C function such "gets()" as unsafe. Other rules demand analysis of data flow, control flow, or configuration files. More complex code analysis tools can express rules as state machines and some can even create "partial models" of how code might execute that allow for more powerful and accurate statements about vulnerability.

Step one of rule creation involves documenting a vulnerability that can found statically. It greatly helps this first step if the performer is familiar with the custom rule creation features of the code analysis tool because there are limits to what's feasibly identified statically by each code analysis tool.

The rule can originate from multiple sources such as programmers' bug repository, corporate coding standards, incident's, published best practices, and other sources. A cryptographic rule defined in step (a) is used to illustrate our step by step process.

a. Scoping the rule.

The first step is to define and scope the rule that we want the tool to enforce. This first definition will be conceptual and not tied to a particular code construct. However the rule should be specific enough to check itself against a source code implementation.
For instance a security policy may mandate the use of strong cryptographic algorithms for secure data transmission. At the implementation level, we want to enforce the use of AES (CBC mode) and 3DES (CBC-EDE3 mode) regardless of the language, toolkit, or platform being used. Any use of unapproved algorithms would violate our rule.

b. Drafting high level axioms (optional)

The second step is to express the rule using a high level description language. Our previous cryptographic rule (described in step (a)) can be expressed with axioms that cover the different implementations that a programmer may write. The high level axioms for our rule might be as in Listing 1.

```
If [Cipher.instance]
and     (
        [used_Cipher != AES(CBC mode)] and
        [used_Cipher != 3DES(CBC-EDE3 mode)]
        )
Then
        Issue_Alarm("CipherMisused");
```

Listing 1

The rules created in this stage are just preliminary drafts; writing complete and well defined axioms will require some further exploratory work. In particular, these rules will need to be revisited after writing a first set of test cases. In fact, it may be difficult or impossible to write any axioms at all without having some test cases on hand already. In such cases, the first step may have to be omitted entirely.

c. Packaging of the test cases

To test a code analysis rule, we use code fragments which either contain rule violations (to test detection ability) or correct code (to test for false alarms). The test cases need to be organized consistently. We package the test cases within an Abstract Class or an Interface containing the java methods illustrated in *Listing 2*.

```
        void trueNegativeExamples();
        void truePositiveExamples();
        void falsePositiveExamples();
        void falseNegativeExamples();
```

Listing 2

The method `trueNegativeExamples()` will host the true negatives test cases. The method `truePositiveExamples()` will host the true positive test cases. Before the first round of testing, the content of these two first methods are hypothetical. For example, when testing a rule that scans for unauthorized cryptographic methods `trueNegativeExamples()` might contain uses of authorized cryptographic algorithms, which should

not generate any violations. At the same time, `truePositiveExamples()` might contain uses of unauthorized algorithms which should lead to violations if the rule is working correctly.. The last two methods, `falsePositiveExamples()` and `falseNegativeExamples()`, are initially empty because their content is tool dependent and therefore not predictable before a code scan. Indeed, two different code analysis tools may not report the same false positives and false negatives. Identifying false negatives can be a difficult subtle game, but it is an important one. Actually catching vulnerabilities classified in our test suite, as false negatives will require manual code review, dynamic testing, or some combination. Failing to identify false negatives means you are missing vulnerabilities present in the code.

d. Writing test cases

The next step is to write test cases which will implement the correct and incorrect way to implement the rule. If an axiom has been written in the previous step, the test cases writing will be facilitated. To illustrate this step we wrote test cases for our previous example in step (a). In Java, there are many possible source code constructs to implement the use of allowed cryptographic algorithm. Therefore we can start to list all the possible correct ways to implement the use of the permissible algorithms. In the Java Cryptographic Extension (JCE) framework [2], in order to use cryptographic algorithm we should get an instance of the Cipher Object. The following code samples in *Listing 3* are all valid implementations.

```
public void trueNegativeExamples()
{
// true negative #1
// Use of AES (CBC mode)
Cipher.getInstance("AES/CBC/PKCS5Padding");

// true negative #2
// Use of 3DES (CBC-EDE3 mode)
Cipher.getInstance("DESede/CBC/PKCS5Padding");

// true negative #3
// Use of String parameters
String cipherSpec1="DESede/CBC/PKCS5Padding";
Cipher.getInstance(cipherSpec1);

// true negative #4
// Load the algorithm name from a property file
which has an authorized algorithm
Properties p = new Properties();
p.loadFromXML(new
FileInputStream(PROPERTIES_FILE));

cipherSpec2 = p.getProperty("cipherSpec");
Cipher.getInstance(cipherSpec2);

// true negative #5
String cipherSpec3 = "DES/CBC/PKCS5Padding";
if (cipherSpec3.startsWith("DES"))
{
cipherSpec3 = cipherSpec3.replaceFirst("DES",
"DESede");
}
Cipher.getInstance(cipherSpec3);

// true negative #N
// etc.
}
```

Listing 3

From a static analysis perspective (with Fortify's Source Analyzer), the previous examples are considered true negatives. The code analysis tool should not report them as findings because they are all valid implementation respecting the corporate mandate on cryptographic algorithm.

Similarly, we have to list all the possible violations of the rule that we are trying to enforce. That list will be our list of true positives, the ones that the tool should recognize as violating our authorized algorithms rule.

Writing these two lists may require imagination and experience. Most of the time programmers are thinking about the right way to program things. Almost oppositely, writing test cases requires to come up with, not strictly speaking abuse case, but data (in this case source code) that will cause the code analysis tool to fail. In essence, we are stress-testing the tool. For instance, the use of an unauthorized algorithm would violate the rule as illustrated by the following code *Listing 4*.

```
private String cipherSpec1;

void init()
{
//unauthorized algorithm
cipherSpec1 = "DES/CBC/PKCS5Padding";
}
…
public void truePositiveExamples()
{
String cipherSpec2 = "AES/ECB/PKCS5Padding";

// true positive #1
// interprocedural
Cipher.getInstance(cipherSpec1);

// true positive #2
Cipher.getInstance(cipherSpec2);

// true positive #3
// concatenating Strings
StringBuffer cipherSpec3 = new
StringBuffer("IDEA");
cipherSpec3.append("/CBC/ISO10126Padding");

Cipher.getInstance(cipherSpec3.toString());
```

```
// true positive #4
// from a system property which has an
// unauthorized algorithm value.
cipherSpec3 = System.getProperty("cipherSpec");

// true positive #N
// etc.
}
```

Listing 4

The goal of having multiple test cases with similar effects is to try to cover all the different implementation variants. It bears clarifying: here we are talking about variations in syntax that might trick the code analysis tool rather than purely alternative implementations of a code construct. For instance the true positive test case #1 and #2 have the same result, their Cipher Object take an unauthorized algorithm as parameter, but the parameter passing is done differently. Specifically, one tests the tool's interprocedural parameter modeling.

Some of the test cases are intentionally too complex for the tool to recognize as true positives or true negatives, but they represent control or data flow that might occur in real application's source code in a less contrived form. For instance true positive #4 takes an environment variable which has an unauthorized algorithm as value. Static analysis tools face tremendous difficulty identifying examples like #4 because an environment variable can be resolved deterministically only at runtime. This test case can have its true negative counterpart which would take an authorized algorithm as environment variable, but again it is unlikely for the tool to be accurate unless the tool's user can provide it hints during analysis. While some vendors' tools allow such 'hints', Fortify's product does not currently.

Maturity of test cases gradually elevates as the tester can define more complex code constructs that define the tool's limits. We did not use a quantitative scale for evaluating the complexity of the test case. Instead, we relied on several years of static analysis experience. For instance we know that some code analysis tools have no inter-procedural analysis checks. Therefore we can add a test case that hides a vulnerability using an inter-procedural call.

The list of false negatives and false positives should now permit us to write well defined axioms specifying the rule at the source code level.

e. Writing/Revisiting the source code level axioms.

Iterating test cases allows us to iteratively refine the accuracy of axioms that will specify what code constructs would violate our custom rule. An accurate axiom would typically describe the rule constraints so the rate of false positives is reduced. In order to expedite rule writing, we used a common grammar for axiom writing. We defined the syntax of this common grammar as pseudo code similar to the specification language through which one writes certain types of custom rules for the Fortify product to facilitate translation. But ideally we would want to have larger common grammar that could cleanly express rules that rely heavily on other analysis such as control flow, data flow, or state machine specification. The axiom corresponding to the cryptographic rule in step (a) is mapping to the code construct in *Listing 5*.

```
// true negative #1
// Use of AES (CBC mode)
Cipher.getInstance("AES/CBC/PKCS5Padding");
```

Listing 5

For our cryptographic example, the Java source code axiom would look like the following Listing 6.

```
FunctionCall:

function.name == "getInstance"
and
function.parameters.length != 0
and
function.enclosingClass.supers contains [Class
name == "javax.crypto.Cipher"]
and
function parameters[0].type=="java.lang.String"
and
not(
arguments[0].constantValue is [String:
startsWith "AES/CBC"] or

arguments[0].constantValue is [String:
startsWith "DESede/CBC"]
)
```

Listing 6

Translation into axioms crucially maps the high level requirements of a security standard to possible implementations in a particular language's source code. It is necessary to ensure that all the rule constraints are captured properly and no constraints are lost during this translation phase.

f. Implementing the axioms using the tool extension mechanism

Next, the rule writer implements a rule by translating axioms into whatever form the tool-specific

extension mechanism requires. Some code analysis tools use a proprietary rule description language; others use programming languages as extension mechanism (C++, python, etc.)

g. Running the static analysis tool against the test cases.

The code analysis tool should take the new custom rules as input and be run against all the true positives and true negatives that we have constructed. In our experiments, this was done in a loop, iteratively, and we used our test cases as well as code “from the wild” to collect measurements on our ability to find additional true positive results and increase a rule’s accuracy through reducing false positives. We believe it is reasonable to expect a 100% increase in both measures when customizing a tool’s existing core rule.

h. Analyzing the results

The test scan creates two new expected categories of test cases: the false negatives and false positives (see *Figure 1*). One of the goals of code analysis is to minimize the number of findings in those two new categories. False positives create noise and require time consuming verification. This is only tolerable if the number of false positives is low. But on the opposite we want to avoid false negatives. In this case, false negatives are true rule violations that the tool missed.

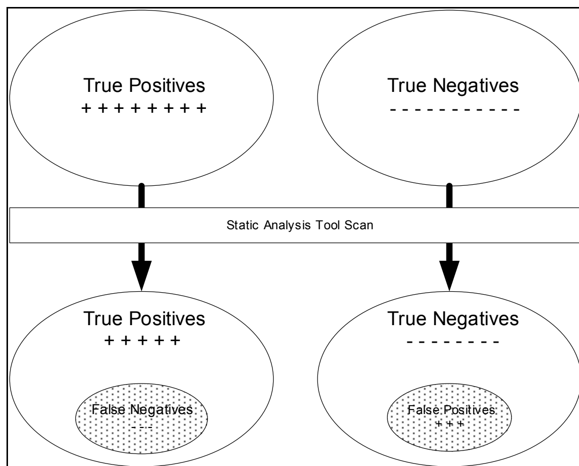

Figure 1

We can therefore reclassify the test cases according to the tool’s findings. We move those test cases belonging to the new categories to their respective methods `falsePositiveExamples()` and `falseNegativeExamples()` from the true negatives and true positives methods.

At this point it is useful to try to understand what confused the tool. Why did the tool report the false positives? Why did it not catch the false negatives? Errors may be caused by the tool itself or by the implementation of the custom rule. The tool has limitations, for instance in our previous example the tool may not be capable of recognizing the value of the `String` input parameter which represents an authorized algorithm. Usually the tool user does not have much control of the tool’s implementation limitations (this applies to commercial tools, where source code is not available). However, the user has control of the custom rule implementation which uses the tool’s extension mechanism. Problems caused by faulty rules can be fixed, and fixing them is the purpose of the next step.

i. Feedback loop, return to step one (axioms) and stop when low false positive and false negative residuals

One of the goals of this framework is to have the static analysis tool reporting all true positives and have a low rate of false positives and false negatives. Therefore after the first iteration, the scan result may not be satisfactory. At a higher level the axioms can be incorrect and may need to be revised. The process may need multiple iterations before being accepted by the user with tolerable error levels.

It is useful to note that some code constructs are more frequently used than others, and reporting the most frequently used code constructs first will lead to the fastest results. This property is illustrated in the following Java code (*Listing 7*).

```
private String cipherSpec1;

void init()
{
//unauthorized algorithm
cipherSpec1 = "DES/CBC/PKCS5Padding";
}
…
public void truePositiveExamples()
{
String cipherSpec2 = "AES/ECB/PKCS5Padding";

// true positive #1
// interprocedural
Cipher.getInstance(cipherSpec1);

// true positive #2
Cipher.getInstance(cipherSpec2);

}
```

Listing 7

The previous *Listing 7* demonstrates that there are many ways to violate the rule. The tool is supposed to catch all true positives, but the test cases that we really care about are the first two test cases (#1 and #2). The remaining true positives test cases (*Listing 4*) are less a concern because they are unlikely to occur but we still desire that the tool covers them. We assume here that most of the programmers would use the case #1 and #2 in a real application. Trying to cover the most likely code constructs for a rule violation can be a wise choice when the possible code constructs are too numerous.

j. Integration with other process

As mentioned earlier, false negatives provide valuable insight into other security activities. False negatives represent what the tool does not find as rule violation, but should ideally. Other techniques can be used to find those false negatives depending on their severity. False negatives should feed manual code review standards, security testing efforts, and in some cases, may guide application deployment or penetration testing efforts.

Custom rules can be further tested by running them against wild code to find out if they behave as predicted in the test framework. The rules can be continually fine-tuned to achieve greater efficiency.

Conclusion

We have described a step by step test methodology that we have used to write efficient custom rules for automated software scanning. This test-driven approach has several benefits. It can expand the state of the art of static tool analysis. Identifying the undesired residual results such as false negatives and false positives can be used to improve the accuracy and coverage of existing tools. The false negatives test the tool's limits and create new technical challenges for tool providers. Being able to isolate the false negatives and positives is also crucial knowledge for the scan results reviews and manual reviews.

References

[1] Fortify Software
http://www.fortifysoftware.com/

[2] Java Cryptography Extension (JCE),
http://java.sun.com/products/jce/

[3] SAMATE NIST Project
http://samate.nist.gov

[4] McGraw G. Software Security: Building Security In, Addison-Wesley Professional, 1st Edition 2006