Secure Coding Assistant

Enforcing Secure Coding Practices Using the Eclipse Development Environment

Benjamin White, Jun Dai, and Cui Zhang
Computer Science Department, California State University Sacramento
ben_white@att.net, jun.dai@csus.edu, zhangc@csus.edu

ABSTRACT

Developing secure software in a world where companies like Anthem Blue Cross, Twitter, Facebook, and Target have had massive amounts of data stolen by hackers is as challenging as it is important. Insecure coding practices are major contributors to software security vulnerabilities. What is missing is an open-source secure coding enforcement tool utilizing well-documented rules that software developers can use to predict potential pitfalls, learn from their mistakes and aid in the construction of secure programs as they build them. To address the need, we have designed a new tool called Secure Coding Assistant for the Eclipse Development Environment that semi-automates several secure coding rules set forth by the CERT division at Carnegie Mellon University. The tool detects violations of the CERT rules for the Java programming language but it is easily extensible to other languages supported by Eclipse. It is an open-source tool with an emphasis on educating software developers in secure coding practices. The tool is disseminated via github at http://benw408701.github.io/SecureCodingAssistant/.

1. INTRODUCTION

Finding secure coding standards is not difficult but following them is. In a 2011 study [20], Veracode analyzed over 6,750 web applications and found that a third of these had SQL code injection vulnerabilities. According to the study, secure coding experts documented how to address these vulnerabilities over a decade ago and it involves something as simple as parameterized SQL statements [20]. A study in India [5] found that less than 1% of engineering students are skilled in secure programming. Even the most "security aware" professionals are writing their code first then adding security as an afterthought [12]. The evidence indicates that there is an overwhelming lack of knowledge and experience when it comes to developing secure software.

Coding for software security is an extremely important issue. In 2013 Facebook, Twitter and Apple were all targets of large-scale attacks. The Twitter attack resulted in 250,000 stolen usernames, passwords and other personal information [18]. Later that year Target was a victim of a security breach and as many as 40 million credit and debit card accounts were compromised [6]. Home Depot's 2,157 stores fell prey to a data security breach in 2014 [2]. CNN [3] also reported two alarming attacks on our government. In July 2014, the Department of Energy was hacked and the attackers stole 100,000 records of personally identifiable information. Earlier in the year, the Army Corps of Engineers lost information on 85,000 dams across the nation. Lastly, the medical industry has been a large target; Anthem Blue Cross had a staggering "millions" of personal health records stolen [4]. If these companies had software systems developed to a higher degree of secure coding standards, these incidents would have been less likely to have occurred.

Every year there are thousands of newly documented software vulnerabilities. The Common Vulnerability Enumeration (CVE) is a database of known security vulnerabilities that is commonly cross-referenced by security tools and is one of the most recognizable vulnerability databases today. The list of vulnerabilities may be accessed online at [16] in a raw format. The published vulnerabilities count in the thousands year over year, starting with a mere 1,500 in 1999 when CVE was founded and leaping past 7,000 in 2014. There is no possible way that a software developer could be expected to learn thousands of CVE's and understand how to write secure code that is resilient against them. The Software Engineering Institute (SEI) of Carnegie Mellon University has made it so that they do not have to. SEI's CERT division documents secure coding rules and recommendations that are language-specific and help protect against these thousands of known vulnerabilities [14]. For instance, there are only 160 secure coding Java rules published by CERT as opposed to the tens of thousands of published CVE's.

The goal of the Secure Coding Assistant is to alert developers when they have violated a CERT rule, educate them on proper secure coding practices and provide an open-source tool to the development community. Though other tools exist that implement some of the CERT rules, the Secure Coding Assistant is the only tool that specializes in CERT rules and is open source. The initial version focuses on Java and educates developers in secure coding practices.

2. RELATED WORK

There are many tools available to developers for building secure applications. In Table 1, several of these tools are compared. The first three are widely used commercial tools and the remaining eight represent a comprehensive list of vulnerability detection plugins for the Eclipse Development Environment. Like the Secure Coding Assistant, several of these tools provide early-detection mechanisms. All of these tools are static analysis tools.

Even though several of the tools available provide an early-detection mechanism, most of them are closed source and do not disclose detailed information on what vulnerabilities are detected and which detection mechanisms are used. The one tool that is open source, FindBugs, focuses on byte code

Table 1: Static analysis tools that scan for security vulnerabilities compared.

Company	*Tool*	*Early/Late Detection*	*Open/ Closed*
Vericode	White Box Testing/Binary Static Analysis	Late	Closed
HP	Fortify Static Code Analyzer	Late	Closed
WhiteHat Security	Sentinel Source	Late	Closed
Klockwork	Klockwork Insight	Early	Closed
Cigital, Inc.	SecureAssist	Early	Closed
The Code Master	Early Security Vulnerability Detector	Early	Closed
Towson University	Static Security Vulnerability Analyzer	Early	Closed
Contrast Security	Contrast for Eclipse	Late	Closed
Sonar Source	SonarLint	Early	Closed
Checkmarx	CxSuite	Late	Closed
Red Lizard Software	Goanna Studio	Early	Closed
Univ. of Maryland	FindBugs	Late	Open
Coverity, Inc.	Coverity Prevent	Early	Closed
Univ. of N. Carolina	ASIDE	Early	Open

and does not alert the developer when they write their source code. There are free tools such as the Early Security Vulnerability Detector (ESVD) and Static Security Vulnerability Analyzer, but these are graduate student research projects that are no longer maintained [1, 13]. The most notable and similar tool available is the ASIDE tool developed by the University of North Carolina [19]. This is a well-developed and advanced detection tool that focuses on OWASP rules and web development whereas the Secure Coding Assistant focuses on CERT rules and any type of Java development. The Secure Coding Assistant is an education-driven, CERT-based, open-source early-detection tool that will be maintained by the university and the development community.

3. DESIGN

3.1 Goals

In addition to being an open-source development tool that evolves with public contribution, the Secure Coding Assistant has two goals. The first is to provide software developers with instant feedback as they write their code. Similar to the way a word processor would alert a writer when they have a grammar or spelling mistake, the Secure Coding Assistant provides messages to the developer that are easy to understand and integrate well into their workflow.

The second goal is to educate on the CERT secure coding practices. Deployed in computer science programming-relevant courses, the tool would provide a foundation of secure coding principles with little effort. To accomplish this goal, the alerts that the programmers receive must provide a message that clearly indicates what rule was violated and what measures they can take towards remediation. This information is provided in the alert messages along with various examples of secure code violations. These mechanisms create a natural learning environment for secure coding practices during the student coding process.

3.2 Architecture

The Secure Coding Assistant runs in the background of the development environment and looks for violations to secure coding rules. The high-level flow is outlined in Figure 1. The only portion of the workflow that is language-specific is the rule violation detection which is outlined in bold. The workflow assumes that a syntax tree of the code segment being analyzed has been built. A syntax tree is a representation of the source code that is easily traversed by a tool like the Secure Coding Assistant. Changes to the syntax tree initiate the code analysis process. Once the process begins, any existing secure coding violations tied to the tree are cleared before the tree is traversed. Each node of the tree is analyzed and if the node contains a rule violation then a new marker is created in the source code where the rule violation is detected. Markers alert the programmer that a violation has occurred and contain the name of the rule, the CERT description and the recommendation from CERT to fix the violation. After the syntax tree traversal is completed the application returns to the initial start state and waits to run again. The markers in the source code display in a tooltip fashion. As the programmer makes changes to the source code the tool runs again in the background, removes all existing markers and only adds new ones if violations exist.

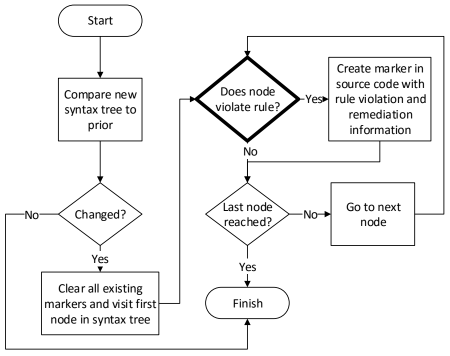

Start
Compare new syntax tree to prior
Changed?
No
Yes
Clear all existing markers and visit first node in syntax tree
Does node violate rule?
Yes
No
Create marker in source code with rule violation and remediation information
Last node reached?
No
Go to next node
Yes
Finish

Figure 1: High-level flow of Secure Coding Assistant, language-specific node in bold.

4. IMPLEMENTATION

4.1 Rule Selection

The CERT website references 185 secure coding rules for the Java programming language [14]. Before selecting which rules to include in the tool, each was classified whether or not automation would be possible. Some rules cannot be automated since they require knowledge of the problem domain. NUM03-J, for instance, states that integer types in Java cannot be used to represent unsigned data [10]. Java programs that need to interoperate with languages like C and C++ must use integer types that can represent the range of unsigned data. This type of rule is very difficult to detect using an automated tool. The tool would need to know that the application is going to be used with components that use unsigned data. The only feasible way to detect this type of vulnerability is to have knowledge of the intended use of the code segment which is not practical for an automated tool. Furthermore, there are entire categories that require some type of metadata for an automated tool to function. An example of this is the “Thread-Safety” category. Without knowledge that a code segment is intended to be run in a multi-threaded environment the tool cannot adequately detect rule violations. Rules like these are infeasible to implement using a tool like the Secure Coding Assistant.

Many of the rules on the CERT website clearly state if they are automatable or not [14]. Others do not say. Out of the total 160 rules available there are 85 that appear to be automatable. Also, the CERT website divides the secure coding rules into 20 categories. Three out of the 20 categories do not contain any rules that can be automated leaving 17 categories with eligible rules to automate. Rules were chosen from these categories based on the severity of the potential vulnerability and an effort was made to sample from as many rule categories as possible. A total of 21 rules were chosen covering 15 categories which represents 88% of the eligible categories.

4.2 Plugin Implementation Details

Eclipse provides a Plugin Development Environment (PDE) that gives plugin developers the ability to extend and customize the development environment. The plugin structure itself is defined using a markup language that contains information on what attributes of the environment are being customized. For instance, a plugin that adds a custom command to one of the menus would extend `org.eclipse.ui.menus` as well as `org.eclipse.ui.commands`. Along with the extension points there are usually other attributes that are defined such as the menu name or the name of the class that contains an execution path when the command is invoked. The Secure Coding Assistant extends two points. The first is `org.eclipse.jdt.core.compilationParticipant` and the second is `org.eclipse.core.resources.problemmarker`. These extension points allow the plugin to participate in the compilation process and create markers that will alert the user a potential vulnerability exists.

4.3 The Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a common representation of a block of source code. Syntax trees are traversed depth-first and define the order of operations.

The Eclipse development environment provides a Java Development Tools (JDT) library. These tools contain a Java language compiler and many other helpful compilation tools including the AST representation of the source code that is being compiled. Eclipse also provides a mechanism for traversing the syntax tree. To traverse the tree, a class may extend ASTVisitor and override one of the many `visit()` methods. ASTVisitor defines a `visit()` method for each type of node (method declaration, assignment, method invocation, etc.) as well as a `preVisit()` and `postVisit()` method which occurs before and after visiting every node. The Secure Coding Assistant uses the `preVisit()` method in its SecureNodeAnalyzer class which attaches to the AST from the SecureCompilationParticipant. There is also a second custom ASTVisitor that is used by the Utility Library that supports the rule detection methods.

4.4 Utility Library

The rule detection logic for many of the CERT rules can be reduced to several sub-problems. These problems are shown in Table 2 along with the methods from the Utility Library that have been developed to solve the given problem. These methods use the ASTNodeProcessor to traverse the AST a second time and gather data on the nodes that occur before and after the node being processed. With this library of reusable code, future rules may be built much easier.

Table 2: Utility Library methods by problem solved.

Problem Solved	*Method*
Was a call made to method x?	calledMethod()
Was method x called prior to method y?	calledPrior()
Was a variable x modified after a call to method y?	modifiedAfter()
Was class c instantiated with argument a?	containsInstanceCreation()
What block b encloses node n?	getEnclosingNode()
Is argument a in a list of arguments l?	argumentMatch()
Retrieve method declaration d from a superclass when method m is overriding it.	getSuperClassDeclaration()

The Utility Library evolved throughout the implementation process. When a rule was chosen for implementation, the pseudo-code for rule was added as comments to the source code. If a step in the pseudo code appeared to be common enough to be reusable in other rules, then it was added to the utility library rather than implemented in the rule logic itself. Even though the Utility Library operates alongside the rule detection logic which is language-specific, the parameters of the methods in the library are designed to be used for multiple programming languages. There were also several instances where method overloading was helpful. For instance, `calledMethod()` was implemented three times. Once to check to see if a method is called from a given class, again to see if it is called from a base class and lastly to see if it is called with particular arguments.

4.5 Rule Logic

Each rule implements the interface IRule and uses the "protected" class modifier so they cannot be instantiated directly. A call to `RuleFactory.getAllRules()` returns an ArrayList of references to each rule that has been fully implemented. The IRule interface provides a level of abstraction that can be used in marker creation and node checking since all rules share the same fundamental properties. These fundamental properties implemented by all secure coding rules in the tool are shown in Table 3.

Table 3: The IRule interface.

Method Signature	*Description*
boolean violated(ASTNode)	Checks to see if the rule has been violated in a given node
String getRuleText()	The description of the rule that was violated
String getRuleName()	The name of the rule violated
String getRuleRecommendation()	The recommended action that will satisfy the rule
int securityLevel()	The security level of the violatedrule, values are defined as LOW, MEDIUM, and HIGH in the Global.Markers class

The `IRule.violated()` method has one parameter, the node that is being evaluated, and returns true if a rule violation was detected at the node location and false otherwise. This makes iterating through a large set of rules very straightforward as shown in Figure 2. In this code segment a collection of rules, built by `RuleFactory.getAllRules()`, each tests a node in the syntax tree. Since this is in the overridden preVisit method, it is run against each node in the syntax tree in a depth-first traversal.

```
public void preVisit (ASTNode node) {
  // Iterate through rules
  for (IRule rule : m_rules)
    if(rule.violated(node))
      m_insecureCodeSegments.add(new
           InsecureCodeSegment(node, rule,
           m_context)); }
```

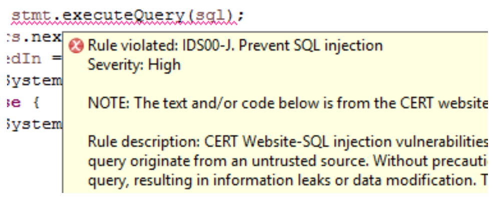
Figure 2: Iterating through rule collection.

5. EVALUATION

5.1 Accuracy

5.1.1 CERT Validation

The CERT website lists several code samples for each secure coding rule along with the rule definition. The samples are presented in pairs, first is an example of a violation of the rule and next is the corrected code segment. Figure 3 shows the Secure Coding Assistant detecting an IDS00-J violation in a code segment taken from the CERT website [9]. In this example the query string is built using parameters supplied by the user. The alert window cites CERT's solution to use a `PreparedStatement` instead. Rule logic was not considered to be complete until all secure coding violation examples shown on the CERT website for that particular rule could be detected by the tool.



```
stmt.executeQuery(sql);
:s.nex
:dIn =
;ystem
se {
;ystem
```

Rule violated: IDS00-J. Prevent SQL injection
Severity: High

NOTE: The text and/or code below is from the CERT website

Rule description: CERT Website-SQL injection vulnerabilities
query originate from an untrusted source. Without precauti
query, resulting in information leaks or data modification. T

Figure 3: IDS00-J violation from CERT detected with Secure Coding Assistant

5.1.2 False Positive Study

The Stanford SecuriBench [7] was used for the false positive study. It consists of applications that have various types of documented vulnerabilities. The Stanford group identified 30 vulnerabilities in 2005 when SecuriBench was first made public. After running seven of the eight programs through the Secure Coding Assistant several thousand potential CERT violations were detected.

The Secure Coding Assistant generated 4,172 secure coding alerts, but the overall distribution shown in Table 4 is quite interesting. Only 8 out of the 21 implemented rules detected violations. Of those 8 rules, 77% of the violations detected were all in one rule, EXP00-J, which states that a programmer should never ignore a value returned by a method [8]. The reason for this is that method return values are often indicators of whether or not the call was successful or they contain some other output that is beneficial to the caller of the method. According to CERT, "Ignoring method return values can lead to unexpected behavior." Upon further investigation, it is not always clear what is to be done with the return value. The next highest rule violation detected was ERR08-J which cautions developers against catching a `NullPointerException` or any of its ancestors [15]. This type of exception is thrown when an application is running and attempts to dereference a pointer that has not been initialized to a value. According to CERT, when this type of runtime error is ignored the application becomes unstable. Rather than catching the exceptions, CERT advises that the application terminate immediately.

Since EXP00-J and ERR08-J both contain a large number of exceptional cases, they have been excluded from the false positive study. To identify the false positive results, each alert was visually inspected and only categorized as a "true positive" if the code segment was a true reflection of the secure coding rule outlined by CERT; all others are classified as a "false positive." The results of this study in Table 5 reflect an overall false positive rate of 8.6% which are isolated to the IDS00-J, IDS11-J and MSC02-J CERT rules.

The largest false positive result was found in detecting the MSC02-J rule. This rule states that a cryptographically secure random number generator should always be used in applications where security is important. The false positive results logged were instances where the random number was being used for purposes besides security. Visual inspection showed the numbers were used for a randomly sorted list which was not related to application security. Fixing this issue with MSC02-J would be difficult since it requires knowledge of how the random number is used. Adding a set of meta tags to the tool to allow programmers to disable security warnings for a line of code would solve this issue. For example, putting `@SuppressSecurity` before the line that generates the alert would cause that rule to be ignored when evaluating the following line for potential vulnerabilities.

The next highest false positive rate is seen in the IDS00-J rule detection which checks for correct usage of the `PreparedStatement.setString()` method. All of the false positive results stemmed from query strings that did not require user input. In these cases, the value being inserted into the query string was a constant value. Additional analysis on how the query string is built would be required to reduce the false positive rate for IDS00-J. This would include parsing the expression into subcomponents and tracing their origin in the source code. In cases where the input is coming from other services or modules this type of a trace would not be feasible.

5.1.3 False Negative Study

A false negative analysis of the Secure Coding Assistant requires segments of Java source code with known vulnerabilities. Due to unavailability of documented CERT violations outside of the CERT website, a limited false negative analysis of the Secure Coding Assistant was performed. It was done by looking for examples of insecure Java code from organizations that document vulnerabilities like the Open Web Application Security Project (OWASP) and Common Weakness Enumeration (CWE). The first test is the example shown on the OWASP website [11] for preventing SQL injection attacks in Java. Our tool accurately detected the violation as shown in Figure 4.

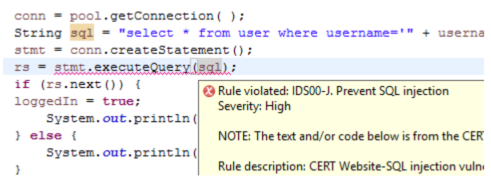
Next, the CWE library was searched for code that would relate to the IDS01-J rule to normalize strings before val-

Table 4: SecuriBench test results.

Level	*Full Name*	*Total*	*Percent*
L2	EXP00-J. Do not ignore values returned by methods	3,211	77.0%
L1	ERR08-J. Do not catch NullPointerException or any of its ancestors	740	17.7%
L2	MET04-J. Do not increase the accessibility of overridden or hidden methods	138	3.3%
L1	IDS00-J. Prevent SQL injection	42	1.0%
L1	MET06-J. Do not invoke overridable methods in clone()	25	0.6%
L1	IDS11-J. Perform any string modifications before validation	7	0.2%
L1	MSC02-J. Generate strong random numbers	7	0.2%
L1	IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method	2	0.0%
L1	IDS01-J. Normalize strings before validating them	0	0.0%
L1	FIO08-J. Distinguish between characters or bytes read from a stream and -1	0	0.0%
L1	SEC07-J. Call the superclass's getPermissions() method when writing a custom class loader	0	0.0%
L1	SER01-J. Do not deviate from the proper signatures of serialization methods	0	0.0%
L1	STR00-J. Don't form strings containing partial characters from variable-width encodings	0	0.0%
L2	ENV02-J. Do not trust the values of environment variables	0	0.0%
L2	EXP02-J. Do not use the Object.equals() method to compare two arrays	0	0.0%
L2	NUM09-J. Do not use floating-point variables as loop counters	0	0.0%
L2	OBJ09-J. Compare classes and not class names	0	0.0%
L3	DCL02-J. Do not modify the collection's elements during an enhanced for statement	0	0.0%
L3	LCK09-J. Do not perform operations that can block while holding a lock	0	0.0%
L3	NUM07-J. Do not attempt comparisons with NaN	0	0.0%
L3	THI05-J. Do not use Thread.stop() to terminate threads	0	0.0%
	Total	*4,172*	

Table 5: False positive analysis.

Rule	*Total Count*	*True Pos. Count*	*True Pos. %*	*False Pos. Count*	*False Pos. %*
MET04-J	138	138	100.0%	0	0.0%
IDS00-J	42	29	69.0%	13	31.0%
MET06-J	25	25	100.0%	0	0.0%
IDS11-J	7	5	71.4%	2	28.6%
MSC02-J	7	3	42.9%	4	57.1%
IDS07-J	2	2	100.0%	0	0.0%
Total	*221*	*202*	*91.4%*	*19*	*8.6%*



```
conn = pool.getConnection( );
String sql = "select * from user where username='" + userna
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
loggedIn = true;
    System.out.println(
} else {
    System.out.println(
}
```

Rule violated: IDS00-J. Prevent SQL injection
Severity: High

NOTE: The text and/or code below is from the CER

Rule description: CERT Website-SQL injection vuln

Figure 4: Output of SQL injection detection.

idation. Figure 5 from the CWE Dictionary [17] is in the "Validate Before Canonicalize" section but is similar to the IDS01-J rule to validate before normalizing a string. In this example the `path` variable is being tested to see if it begins with `/save_dir/` but there is no guarantee that the path name is in canonical form. To correct this code, the `path` string needs to be converted to canonical form before the comparison. Unfortunately, the violation went undetected by the Secure Coding Assistant. The key difference between the IDS01-J rule on the CERT website and the CWE example is that the CWE example includes canonicalization in the category of normalization but the CERT rule only gives the example of the normalize method. This type of vulnerability would be difficult to detect since there is no indication that the text string represents a path and the canonical form of the path depends on the operating system.

Another code segment from CWE is shown in Figure 6 which illustrates a vulnerability that should be detected under the CERT OBJ09-J rule. OBJ09-J states that class comparison should be done using the `==` operator on the class objects themselves and not the class names. In the example

```
String path = getInputPath();
if (path.startsWith("/safe_dir/")) {
  File f = new File(path);
  return f.getCanonicalPath(); }
```

Figure 5: Validate before canonicalize example from CWE [17].

```
public boolean equals(Object obj) {
  boolean isEquals = false;
  // first check to see if the object is of the same
      class
  if (obj.getClass().getName().equals(
      this.getClass().getName()))
    // then compare object fields
    if (...) { isEquals = true; }
  return isEquals;}
```

Figure 6: Comparison of classes by name from CWE [17].

given, changing the comparison line to `obj.getClass() == this.getClass()` would rectify the problem. The Secure Coding Assistant successfully detected the vulnerability.

5.2 Efficiency

The Eclipse development environment has a responsiveness monitoring tool that will log delays over a certain threshold. The efficiency analysis for the Secure Coding Assistant was done by setting the monitor threshold to 10 milliseconds then loading 5 SecuriBench source code files 3 times with the plugin enabled and 3 times with the plugin disabled. The difference between the average load time without the plugin and the average load time with the plugin was recorded as the increase in load. The results of the study in Table 6 show that the plugin added an additional 0.03 to 0.20 seconds to the load time for each source file. There appeared to be a correlation between the amount of additional processing time and the number of detected alerts. The last column in the table shows the additional time per alert and ranges from 2 to 4.5 milliseconds.

Table 6: Plugin efficiency analysis.

Application	*Source File*	*Alerts*	*Increase (sec.)*	*Time per Alert (ms)*
pebble	SimpleBlog.java	46	0.2037	4.428
roller	WebLogEntry-FormAction.java	16	0.0713	4.458
webgoat	CreateDB.java	49	0.1923	3.925
snipsnap	Configuration-Map.java	23	0.0270	1.174
snipsnap	Configura-tionProxy.java	19	0.0380	2.000

6. LIMITATIONS, CONCLUSION AND FUTURE WORK

The Secure Coding Assistant has demonstrated practical, efficient and accurate applications for education in computer science. Future development work will focus on fine-tuning the existing rule detection logic, building logic for additional rule detection, expanding the tool to support additional programming languages and adding additional features.

The SecuriBench testing showed that some rules like the EXP00-J rule need additional documentation on exception cases. The Secure Coding Assistant can help detect such cases and aid in fine-tuning the CERT rule library. The false positive and false negative study showed that there are a few adjustments that could be made to the rule logic to improve performance. There are also several rules that cannot be automated because the rule itself is context-specific. For instance, whether or not an application is running in a multi-threaded environment and requires thread safety or whether or not the Java application is interoperating with programs developed in other programming languages. These types of things cannot be identified through code inspection but a system of meta tags could be developed to indicate whether or not a block of code requires a certain type of security.

There are many static analysis tools that are available to the programming community. Several of these are Eclipse plugins, a few of them provide early-detection techniques but none of them are open-source learning tools for the CERT secure coding rules. The Secure Coding Assistant provides the development community with an educational tool in secure coding practices. It is open source, extensible and will be maintained. For more detailed information, please visit the project website at http://benw408701.github.io/SecureCodingAssistant/.

7. REFERENCES

[1] J. Dehlinger, Q. Feng, E. Oestrich, and M. Smith. SSV Checker - An Eclipse plug-in interface static security vulnerability checker, Aug. 26 2012. URL: http://ssvchecker.sourceforge.net/ [accessed: 2015-11-15].

[2] B. Elgin, M. Riley, and D. Lawrence. Hacked wide open.(home depot fails to improve security). *Bloomberg Businessweek*, pages 39–40, Sept. 22 2014.

[3] C. Frates and C. Devine. Government hacks and security breaches skyrocket. *CNN Wire*, Dec. 19 2014.

[4] J. J. Gelsomini and K. H. Garcia. Anthem's data breach impacts many anthem and non-anthem plans: Necessary employer actions now. *Employee Benefit Plan Review*, 69(11):5–7, 2015.

[5] HT Media Ltd. Fewer than 1% of engineering students skilled in secure programming. *Mint*, Feb. 2014.

[6] T. F. Lindeman. Target acknowledges security breach; 40 million accounts compromised. *McClatchy - Tribune Business News*, Dec. 20 2013.

[7] B. Livshits, M. Martin, M. Lam, J. Whaley, D. Avots, M. Carbin, and C. Unkel. Stanford SecuriBench, Dec. 21 2005. URL: http://suif.stanford.edu/~livshits/securibench/intro.html [accessed: 2015-12-23].

[8] D. Mohindra. EXP00-J. Do not ignore values returned by methods, Nov. 03 2015. URL: https://www.securecoding.cert.org/confluence/display/java/EXP00-J.+Do+not+ignore+values+returned+by+methods [accessed: 2015-12-23].

[9] D. Mohindra. IDS00-J. Prevent SQL injection, Nov. 03 2015. URL: https://www.securecoding.cert.org/confluence/display/java/IDS00-J.+Prevent+SQL+injection [accessed: 2015-12-22].

[10] D. Mohindra. NUM03-J. Use integer types that can fully represent the possible range of unsigned data, June 03 2015. URL: https://www.securecoding.cert.org/confluence/display/java/NUM03-J.+Use+integer+types+that+can+fully+represent+the+possible+range+of++unsigned+data [accessed: 2015-11-01].

[11] OWASP. Preventing SQL Injection in Java, Aug. 14 2014. URL: https://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java [accessed: 2016-02-06].

[12] M. K. Pandit. Developing secure software using aspect oriented programming. *IOSR Journal of Computer Engineering*, 10(2):28–34, 2013.

[13] L. Sampaio. The Code Master, June 17 2015. URL: http://thecodemaster.net/ [accessed: 2015-11-15].

[14] S. Shrum. 2 - Rules - CERT Oracle Coding Standards for Java, Apr. 07 2015. URL: https://www.securecoding.cert.org/confluence/display/java/2+Rules [accessed: 2015-11-14].

[15] D. Svoboda and A. Hicken. ERR08-J. Do not catch NullPointerException or any of its ancestors, Nov. 03 2015. URL: https://www.securecoding.cert.org/confluence/display/java/ERR08-J.+Do+not+catch+NullPointerException+or+any+of+its+ancestors [accessed: 2015-12-23].

[16] The Mitre Corporation. CVE - Download CVE, Nov. 13 2015. URL: https://cve.mitre.org/cve/cve.html [accessed: 2015-11-15].

[17] The Mitre Corporation. CWE-2000: Comprehensive CWE Dictionary, Dec. 08 2015. URL: http://cwe.mitre.org/data/slices/2000.html [accessed: 2016-02-06].

[18] A. Vamialis. Online service providers and liability for data security breaches. *Journal of Internet Law*, 16(11):23–33, 2013.

[19] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton. ASIDE: IDE support for web application security. *ACSAC 2011*, Dec. 2011.

[20] J. Zhu, J. Xie, H. R. Lipford, and B. Chu. Supporting secure programming in web applications through interactive static analysis. *Journal of Advanced Research*, 5(4):449–462, 2014.