An early detection tool in Eclipse to support secure coding practices

Benjamin White

California State University Sacramento,
Sacramento, CA 95819, USA
Email: benwhite@csus.edu
and
Mother Lode Holding Company,
Roseville, CA 95747, USA

Jun Dai* and Cui Zhang

California State University Sacramento,
Sacramento, CA 95819, USA
Email: jun.dai@csus.edu
Email: zhangc@csus.edu
*Corresponding author

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. Even though several static analysis tools are available that can search for and identify security holes in software applications, this process usually runs too late and any remediation will be more costly after large portions of the software have been built. The early detection tools that do exist are closed source and utilise proprietary software vulnerability rule sets. What is missing is an open-source secure coding enforcement tool utilising 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 and a tool demo is disseminated via github at http://benw408701.github.io/SecureCodingAssistant/.

Keywords: secure coding; development tool; Java; Eclipse; static analysis; education.

Reference to this paper should be made as follows: White, B., Dai, J. and Zhang, C. (2018) ‘An early detection tool in Eclipse to support secure coding practices’, *Int. J. Information Privacy, Security and Integrity*, Vol. 3, No. 4, pp.284–309.

Biographical notes: Benjamin White is an IT Division Manager for Mother Lode Holding Company (MLHC) and part time Lecturer at California State University Sacramento (CSUS) in the Computer Science Department. At MLHC he supports over 1,000 users that comprise several divisions of title insurance companies. As part of the MLHC IT management team he oversees software development and support and serves as a Systems and Software Analyst. He received his MS in Computer Science from the CSUS and lectures now for one introductory Java programming course.

Jun Dai is an Assistant Professor in the Department of Computer Science at California State University Sacramento. He holds a PhD in Information Science and Technology from the Penn State University. His expertise includes network security and system security. His research includes hardware-enforced virtualisation (such as QEMU-KVM), network-wide information flow monitoring and tracking, attack graph-based vulnerability analysis, intrusion and malware detection, cyber situation awareness, and secure programming.

Cui Zhang is a Professor in the Department of Computer Science at California State University Sacramento. She holds a PhD in Computer Science from the Nanjing University. Her research and teaching interests include formal methods for secure software engineering, secure coding, computer-aided software engineering (CASE), software architecture, and programming language theories and paradigms.

This paper is a revised and expanded version of a paper entitled ‘Secure coding assistant: enforcing secure coding practices using the Eclipse development environment’ presented at National Cyber Summit, Huntsville, AL, 8–9 June 2016.

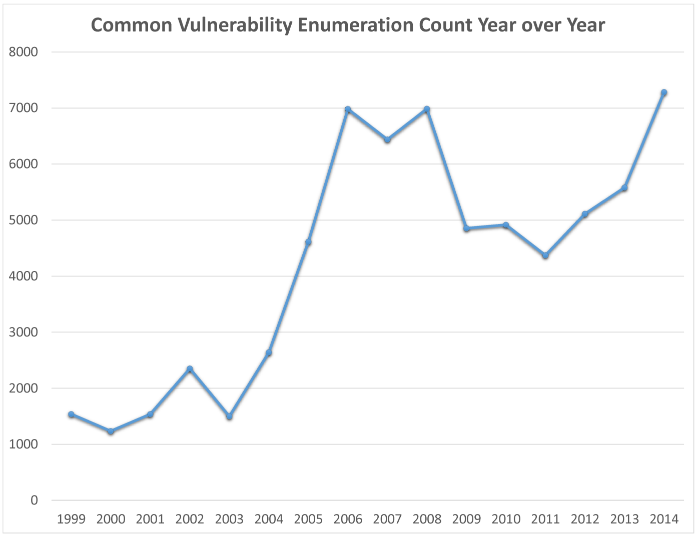
1 Introduction[1]

Finding secure coding standards is not difficult but following them is. In a 2011 study (Zhu et al., 2014), Veracode analysed 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 parameterised SQL statements (Zhu et al., 2014). A study in India (HT Media Ltd., 2014) 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 (Pandit, 2013). The evidence indicates that there is an overwhelming lack of knowledge and experience when it comes to developing secure software.

Coding for security and especially software security is an extremely important issue. In 2013 Facebook, Twitter and Apple were all targets of large-scale attacks. The Twitter attack detailed in the Journal of Internet Law (Vamialis, 2013) resulted in 250,000 accounts compromised and stolen usernames, passwords and other personal information. Later that year target was a victim of a security breach and as many as 40 million credit and debit card accounts were compromised (Lindeman, 2013). Home Depot’s 2,157 stores fell prey to a data security breach in 2014 (Elgin et al., 2014). CNN (Frates and Devine, 2014) 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, hackers hit the Army Corps of Engineers and took information on 85,000 dams across the nation. Lastly, the medical industry has been a large target as well and we see in Gelsomini and Garcia (2015) that Anthem Blue Cross had a staggering 'millions' of personal health records stolen. If these companies had software systems developed to a higher degree of secure coding standards, then 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 recognisable vulnerability databases today. The list of vulnerabilities may be accessed online at The Mitre Corporation (2015a) in a raw format. Figure 1 is a compilation of these documented vulnerabilities after removing all items marked as 'reject', 'reserved' or 'deprecated'. The remaining 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 to 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 Shrum (2015). For instance, there are only 185 secure coding Java rules published by CERT as opposed to the tens of thousands of published CVE's.

Figure 1 CVE's published year over year (see online version for colours)



The goal of the secure coding assistant by California State University Sacramento, commonly referred to as 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 specialises in CERT rules and is open source. The initial version of the secure coding assistant detects 21 secure coding rules for Java, see Section 5.1 for rule selection, but is designed to allow for the addition of rules and programming languages.

The preliminary version of this paper has already appeared in White et al. (2016), which is a short report of a graduate student project designed as a sustainable open-source contribution to the secure programming education community (White, 2016). The short version briefly introduced the concept of a secure coding assistant and its practical applications in education and security vulnerability detection. This paper extends the topics and the Secure Coding Assistant design and implementation in much greater depth, including additional descriptions of static analysis tools and their relevance to secure coding (Section 2), detailed project methodologies (Section 3), implementation-specific details on rule IDS00-J (Section 4), additional information on the plugin extension points that were used in the plugin implementation (Section 5.2), detailed explanation of the concept of a 'compilation participant' (Section 5.3), graphical illustration of the concept of an abstract syntax tree (AST) (Section 5.4), additional rules in the false positive analysis (Section 6.1.1), and additional analysis of the false positive study results (Section 6.1.1).

2 Related work

There are many tools available to developers for building secure applications. In Table 1, several of these tools are compared and classified as early- or late-detection as well as open- or closed-source. The first three are widely used commercial tools and the remaining 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.

Static analysis tools as described by Díaz and Bermejo (2013) all follow the same basic workflow: transform the code, analyse certain properties and display results. In this same article, they also touch on the static analysis counterpart, dynamic analysis, which analyses properties of an application while it is running. Dynamic tools are capable of detecting vulnerabilities that static tools cannot but provide even later detection than static tools and are not included in the comparison table. The cost of catching and fixing vulnerabilities later in the software lifecycle is so significant that companies such as Microsoft have begun to hold "developers [personally] liable for the security and integrity of the code they write" (Fisher, 2003). The solution is to minimise the security vulnerabilities in application code at the time the developer is writing it. The process described in Díaz and Bermejo (2013) must be running in the background while the developer is typing their code. The tools in Table 1 listed as 'early' detection provide the live feedback necessary for the developer to write secure code and learn as they do it. The tools listed as 'late' detection require the developer to finish writing their code, launch the tool and load the code, then review the results and make changes to the source as necessary. The secure coding assistant falls into the 'early' category since it alerts the developer when they are at risk of violating a secure coding rule. This type of tool provides feedback at the earliest possible stage in the development process and teaches developers secure coding practices at the same time.

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

Company	*Tool*	*Early/late*	*Open/closed*
Veracode	White box testing/binary static analysis (Veracode, 2018)	Late	Closed
Micro Focus	Fortify static code analyser (Micro Focus, 2018)	Late	Closed
WhiteHat Security	Sentinel source (WhiteHat Security, 2018)	Late	Closed
Rogue Wave Software	Rogue wave (Rogue Wave Software, 2018)	Early	Closed
Synopsys	SecureAssist (Synopsys, Inc., 2018)	Early	Closed
The Code Master	Early security vulnerability detector (Sampaio, 2015)	Early	Closed
Towson University	Static security vulnerability analyser (Dehlinger et al., 2012)	Early	Closed
Contrast Security	Contrast for Eclipse (Contrast Security, 2018)	Late	Closed
Sonar Source	SonarLint (SonarSource S.A., 2018)	Early	Open
Checkmarx	CxSAST (Checkmarx, Ltd., 2018)	Late	Closed
Red Lizard Software	Goanna studio (Red Lizard Software, 2015)	Early	Closed
University of Maryland	FindBugs (University of Maryland, 2018)	Late	Open
University of North Carolina	ASIDE (Xie et al., 2011)	Early	Open

Even though several of the tools available provide an early-detection mechanism, most of them are closed-source and only one, Goanna Studio by Red Lizard Software (2015), mentions validation against the CERT secure coding rules. Goanna Studio also only provides support for the C and C++ programming languages whereas the Secure Coding Assistant currently supports Java but extensible to others. In addition, even though Goanna Studio lists CERT as being one of the sources for secure coding rules, it also lists several others and it does not indicate specifically which rule is violated and the rule source when a violation is detected. There are other tools such as the early security vulnerability detector (ESVD) and static security vulnerability analyser, available at Dehlinger et al. (2012) and Sampaio (2015). These tools were developed by graduate students and share many of the same goals as the secure coding assistant. Neither of them are open-source, and it is uncertain whether they will be maintained since the graduate research is completed. One tool that is open source, FindBugs, focuses on byte code and does not alert the developer when they write their source code. A notable and similar tool available is the ASIDE tool developed by the University of North Carolina (Xie et al., 2011). 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. Another similar tool is SonarLint by SonarSource S.A. (2018), but the focus of this tool is on software quality and bug detection rather than vulnerability detection. At this time there does not exist an open-source tool that detects vulnerabilities in source code that the development community can build upon and optimise as new vulnerabilities are documented and new detection methods are discovered.

The secure coding assistant has also been expanded to include enforcement of the design by contract methodology as presented in Li et al. (2017). Additionally, in a new effort reported in Aldausari et al. (2018), the design by contract methodology is combined with inference rules of programming logic to assist in locating coding errors in Java applications. This combination can be included in the secure coding assistant. There is also further work in progress to expand the functionality of the secure coding assistant to the C programming language (Melnik, 2018). These exciting and ongoing expansions on the secure coding assistant make the tool much more versatile than before.

3 Methodology

In addition to being an open-source development tool that evolves with public contribution, thesecure coding assistant has two goals. The first is to provide software developers with instant feedback as they write their source 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 is to educate on the CERT secure coding practices. Initially the C# programming language was considered but the lack of well-documented secure programming rules for C# was the driving force behind developing a tool that focuses on the CERT secure coding rules for Java Shrum (2015). A decision was also made regarding which development environment to initially support. Currently NetBeans and Eclipse are both very popular development environments for the Java programming language. They both provide plugin development tools and support multiple programming languages. Eclipse, however, is widely used by the student body at California State University Sacramento and since the goal was to develop a tool that could be used by the students, the decision was made to develop a plugin for Eclipse.

Some of the static analysis tools described earlier are categorised as ‘early’ detection tools. This means that they provide feedback to the programmers as they are typing their code. The other tools require the developer to complete a section of code, send it to the tool and then receive feedback. The secure coding assistant follows the early detection methodology and provides live feedback as the source code is being typed. This type of feedback is already quite common in a development environment. Modern development environments provide live syntax checking and type compatibility checking that validates that the code adheres the rules of the programming language. For instance, a rule in many programming languages is that the addition operator requires two operands on either side that are numerical expressions. For example, trying to add the number 1 to the text string ‘hello’ would result in an error under these rules as would putting the addition operator at the beginning of the operands rather than in-between. This type of instant feedback saves the developer time and helps them write syntactically correct and type-compatible program code before compilation rather than waiting and having to fix those problems later. This is exactly why the secure coding assistant needs to be designed similarly. The types of mistakes that lead to insecure program code are best identified and corrected early, while the developer is writing their code which reduces time and cost of remediation later in the development process.

The second goal of the secure coding assistant, education, is made possible through the CERT website and their thorough documentation of the Java secure coding rules (Shrum, 2015). The alerts that the programmers receive must provide a comprehensive message that clearly indicates what rule was violated and what measures they can take towards remediation. The CERT website provides this information along with various examples of secure code violations and proposed fixes. They are incorporated into the secure coding assistant alert messages. This creates a natural learning environment for secure coding practices during the development process.

4 Design

The secure coding assistant continuously runs in the background of the development environment and looks for violations to secure coding rules. The high-level flow is outlined in Figure 2. The workflow assumes that a syntax tree of the code segment being analysed 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 or any other tool that participates in parsing code in a programming language and is used extensively in the implementation. 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 analysed and if the node contains a rule violation then a new marker is created in the source code where the rule violation is detected. The rule violation logic is the only component that is language-specific. Markers alert the programmer that a violation has occurred and contain the name of the rule, a description from the CERT website (Shrum, 2015) 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 that remain 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. In this manner, rule violations that have been fixed will no longer show.

Since Eclipse supports multiple programming languages, the design allows for future support of any programming language supported by Eclipse. To accomplish this, the only portion of the workflow that is language-specific is the rule violation detection. This component of the workflow is outlined in bold in Figure 2. An example of a Java-specific rule is IDS00-J which is 'prevent SQL injection' and shows how the Java `PreparedStatement.setString()` method is the most effective way to sanitise data being passed to a SQL query string (Mohindra, 2015a). In Figure 3 this Java secure coding rule is translated into a workflow to detect violations. The logic used for the rule violation assumes that if a SQL query is being run then there must be at least one parameter that is obtained from the user so there must be at least one call to `PreparedStatement.setString()`. There is a possibility of a false positive when the query string does not require any parameters and a false negative when the programmer uses the `setString()` method once but not for subsequent parameters, but the emphasis is on the general case since capturing the number of user parameters required would not be feasible.

Figure 2 High-level flow of secure coding assistant

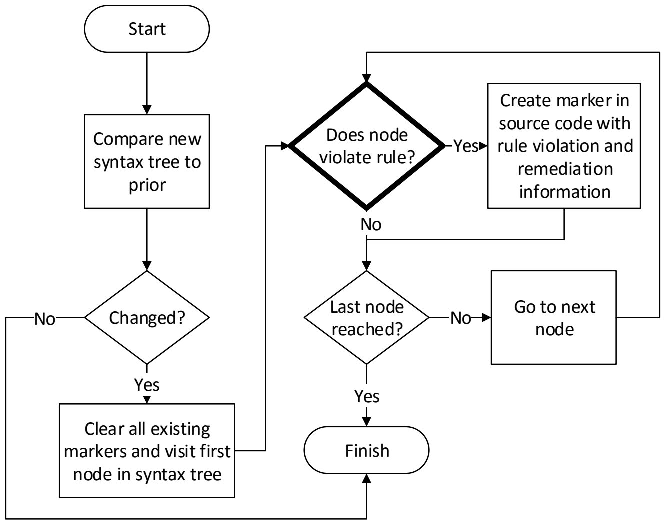
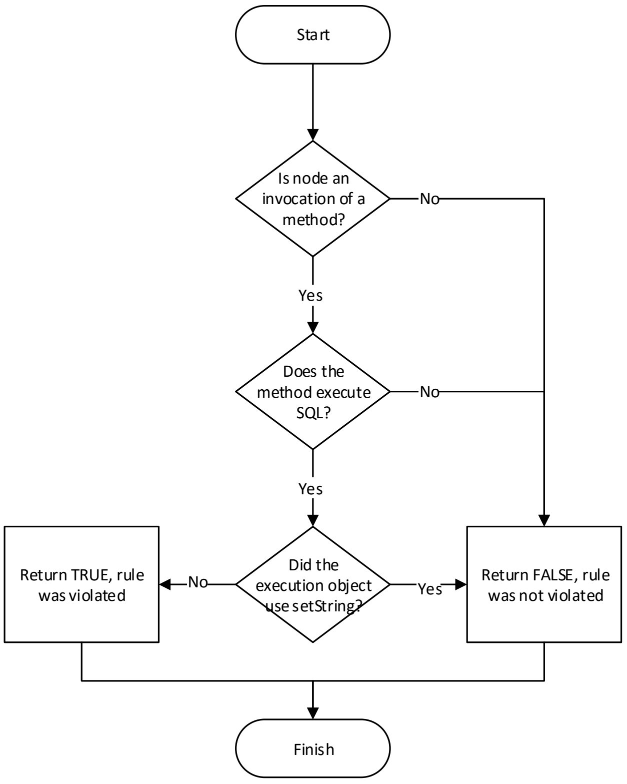

Start
Compare new syntax tree to prior
Changed?
No
Yes
Clear all existing markers and visit first node in syntax tree
Does node violate rule?
Yes
Create marker in source code with rule violation and remediation information
No
Last node reached?
No
Go to next node
Yes
Finish

Figure 3 Sample flow of SQL injection violation detection


Start
Is node an invocation of a method?
No
Yes
Does the method execute SQL?
No
Yes
Did the execution object use setString?
No
Yes
Return TRUE, rule was violated
Return FALSE, rule was not violated
Finish

Additional secure programming rules are all crafted in a similar fashion using logic that looks for a node which is a call to a method, instantiation of an object, inheritance from a base class or some other structure identified in the CERT rule. Once identified, the context in which that node is executed is evaluated programmatically. For instance, many of the data sanitisation rules require that strings be normalised before they are processed. Detecting a rule violation of this type reduces to finding a call to the method that processes the data then checking to see if a call to a normalise method occurred prior and in the same scope.

5 Implementation

5.1 Rule selection

The CERT website references 185 secure coding rules for the Java programming language (Shrum, 2015). Before selecting which rules to include in the tool, each rule was reviewed and classified as to 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 (Mohindra, 2015b). 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.

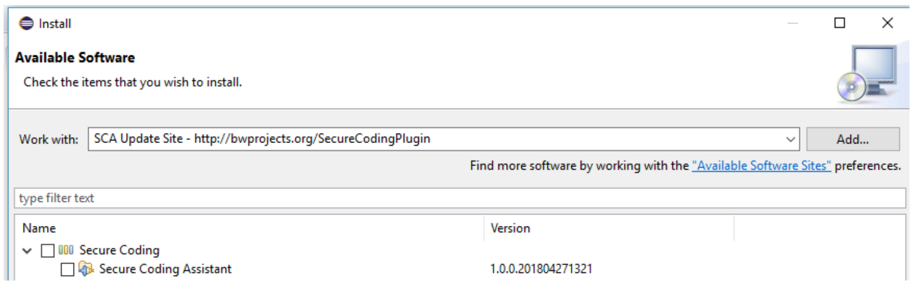
Many of the rules on the CERT website clearly state if they are automatable or not (Shrum, 2015). Others do not say. Out of the total 185 rules available there are 85 that either state explicitly that they may be automated or appear to be automatable. Also, the CERT website divides the secure coding 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 and 25% of the total eligible rules. As with similar static analysis tools, the secure coding assistant cannot be a solution to catch every potential secure coding violation. Instead, the goal must be to capture potential vulnerabilities whenever and wherever possible and direct the developer to resources that may educate them further.

5.2 Plugin implementation details

Eclipse provides a plugin development environment (PDE) that gives plugin developers the ability to extend and customise the development environment. The plugin structure itself is defined using a markup language that contains information on what attributes of the environment are being customised. 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 as well 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.

The first extension point, compilationParticipant, allows the plugin to participate in the compilation process. Part of this extension point also requires a class definition that extends a super class, called CompilationParticipant, which receives notifications at various stages of the compilation process. The second extension point, problemmarker, allows for the creation of custom markers. A marker in Eclipse is a warning, alert, task, or error that developers use to track issues in their code or leave reminders to revisit sections of the source code. The secure coding assistant used generic 'warning' markers in early stages of development but the need to track the various severity levels of the CERT secure coding rules necessitated the change to a customised marker. With a customised marker the base marker type can be extended to have additional fields of any types. The secure coding rule violation markers have an additional enumerated field for severity to capture the three severity levels reflected on the CERT website (Shrum, 2015).

Figure 4 Secure coding assistant update site (see online version for colours)



SCA Update Site - http://bwprojects.org/SecureCodingPlugin

Find more software by working with the "Available Software Sites" preferences.

Name	Version
Secure Coding	
Secure Coding Assistant	1.0.0.201804271321

In addition to the plugin definition itself, Eclipse provides the ability to package similar plugins as a 'feature' and deploy them on an update site. Deploying plugins through an update site is beneficial for anyone wishing to install the plugin for two reasons. First, Eclipse has an 'install new software' feature in the Help menu that you can use to install new plugins using the update site. Since the feature is installed through the Eclipse update tool, it is equally convenient to uninstall the plugin if it is no longer needed. Secondly, using the update site lets users quickly check for updates using the 'check for updates' feature which is also in the Help menu. The update site is online at http://bwprojects.org/SecureCodingPlugin and the display of the update tool is shown in Figure 4. Note that the current version number 1.0.0.201804271321 reflects the date and time of the build (4/27/2018 at 1:21 PM) which is generated automatically by Eclipse when releasing a new build of the feature.

5.3 Compilation participants

The compilationParticipant extension point directs compilation events to any custom class that extends the CompilationParticipant super class. The SecureCompilationParticipant is such a class and is the top-most component in the design of the secure coding assistant. The SecureCompilationParticipant's `reconcile()` method (inherited from CompilationParticipant) is called every time a 'reconcile' event occurs. This type of event is triggered by any modification to the application source code. The only parameter to the `reconcile()` method is an object that conveys information about the context of the reconciliation such as the name of the file that was modified, the types of changes that occurred and a copy of the AST.

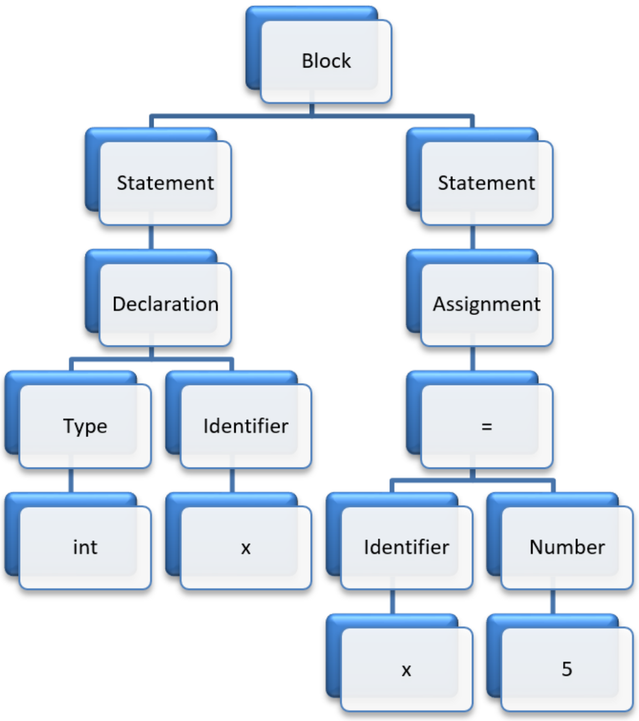
The SecureCompilationParticipant uses the following approach to handling reconciliation events: look for a change to the AST, if there was a change then get a reference to the new AST, clear existing markers, then traverse the new AST and create new markers as necessary. In general, compilation participants are not compiling the source code into an assembly language but only responding to compilation events which include the dynamic syntax checking that is so common in modern development environments like Eclipse. This is exactly what the Secure Coding Assistant is doing, checking the source code, but rather than checking the syntax the emphasis is on the semantics and how the compiled application would affect the application security. The SecureCompilationParticipant is the backbone to the Secure Coding Assistant plugin and is responsible for managing rule detection and marker creation and management.

5.4 The AST

An AST is a common representation of a block of source code. Syntax trees are traversed depth-first and define the order of operations. A simple example is shown in Figure 5, where a block of code contains one declaration of a variable x of type integer and one assignment of the value 5 to x.

The Eclipse development environment provides a Java development tools (JDT) library. The core components of this library are found in `org.eclipse.jdt.core`. 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. At the time that the `reconcile()` method is called the AST has already been built since it is required by any compilation participants. Eclipse also provides a mechanism for traversing the syntax tree, the ASTVisitor class. Any application that wishes to traverse the syntax tree and execute code at any given node in the tree may implement a class that extends 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. Note that even though the `visit()` method is defined for each node type, the `preVisit()` and `postVisit()` methods are defined once, generically, for any node type. The Secure Coding Assistant uses the `preVisit()` method in its SecureNodeAnalyser 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. This ASTVisitor is called ASTNodeProcessor and defines several `visit()` methods that are used to gather data on the context of a node while evaluating whether or not it contains a CERT rule violation.

Figure 5 The AST representation of a code segment (see online version for colours)



5.5 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 the high-level rule logic 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 block of 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.

5.6 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. A rule, for instance, may be violated at a particular node and has several properties like the rule name, level of severity and the recommendation when a violation is detected. 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 level of abstraction makes iterating through a large set of rules very straightforward as shown in Figure 6. 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.

Rule violation detection is reduced to implementing the logic for the rule's `violated()` method. Figure 7 shows the implementation for the IDS00-J rule which states that a SQL query in Java must use calls to the `PreparedStatement.setString()` method to properly sanitise and place query parameters in a query string before sending it to the database for execution (Mohindra, 2015a). The implementation is simplified by using utility library. First the node is checked to see if it is an invocation of a method. If it is, then the one of the methods from the Utility Library are used to check if `PreparedStatement.executeQuery()` is called and if it is the rule is violated only if there is not a call to `PreparedStatement.setString()` prior to `PreparedStatement.executeQuery()`. Compare this to the high-level flow in Figure 3.

Figure 6 Iterating through rule collection (see online version for colours)

```
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 7 Implementation of the IDS00-J rule (see online version for colours)

```
public boolean violated(ASTNode node) {
  boolean ruleViolated = false;
  MethodInvocation method;

  if (node instanceof MethodInvocation) {
    method = (MethodInvocation) node;
    if (Utility.calledMethod(method,
        PreparedStatement.class.getCanonicalName(), "executeQuery"))
      ruleViolated = !Utility.calledPrior(method,
          PreparedStatement.class.getCanonicalName(), "setString");
    else
      ruleViolated = Utility.calledMethod(method,
          Statement.class.getCanonicalName(), "executeQuery");
  }

  return ruleViolated;
}
```

6 Evaluation

6.1 Accuracy

6.1.1 False positive study

The Stanford SecuriBench (Livshits et al., 2005) 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 eight out of the 21 implemented rules detected violations. Of those eight 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 (Mohindra, 2015c). 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" (Mohindra, 2015c). Upon further investigation, it is not always clear what is to be done with the return value. For instance, a large number of alerts were generated on calls to the `StringBuffer.append()` method. This method returns a reference to the modified buffer but the buffer that calls it is also modified. It is not clear in the Java documentation the reason for the return value but it mentions that developers should be using the `StringBuilder` class now instead. There were other circumstances where ignoring the return value seemed appropriate. The `Properties.setProperty()` method returns a reference to the previous value before it overwrites it. Though there are many cases where capturing the previous value is useful, ignoring it is certainly not always a security risk. The high count of EXP00-J violations, even though they are correctly categorised according to CERT, are very likely due to the CERT rule itself needing some additional restrictions. Many other CERT rules have exceptions and this one would benefit from excluding methods that return a reference to the calling object or certain types of return values that are for convenience for the programmer like capturing a value before overwriting it.

The next highest rule violation detected was ERR08-J which cautions developers against catching a `NullPointerException` or any of its ancestors (Svoboda and Hicken, 2015). This type of exception is thrown when an application is running and attempts to dereference a pointer that has not been initialised 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. The rule also states that the ancestors `RuntimeException`, `Exception` and `Throwable` should never be caught since catching one of these could implicitly catch the `NullPointerException`. The secure coding assistant identified 740 violations of this rule which accounted for 17.7% of the rule violations detected, but a majority of those were an ancestor and not the `NullPointerException`. Quite frequently developers will catch an exception as a generic `Exception` rather than the more specific type of exception that is being thrown (e.g., `IOException`). Unfortunately, this means that the tool cannot accurately detect this violation when developers catch generic exceptions. Of the remaining rules, 1.2% fell into the IDS category which accounts for data sanitisation and helps prevent SQL injection attacks and 4.1% were in other categories.

To identify the false positive results, each secure coding alert was visually inspected and only categorised 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 0.5%.

The largest false positive result was 57.1% and found in detecting the MSC02-J rule (Katsis, 2016). 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 encryption or security. Visual inspection showed the numbers were used for a randomly sorted list which is not related to application security so they could not be counted as a true positive result. Fixing this issue with MSC02-J would be difficult since it requires knowledge of how the random number is used. The standard Java development kit does not provide any standard methods to support cryptography, but there is a Java cryptography extension (JCE) that does. The tool should not assume, however, that a developer is using JCE when needing a secure random number. This fact was shown in the false positive analysis when pseudo-random numbers were used for key generation. In general, secure random numbers may always be used even when not necessary. For this reason, advising the developer to never use a pseudo-random number in all cases may increase false positives but will also urge the developer to shy away from pseudo-random number generators completely. Another solution to the MSC02-J false positives is to add a set of meta tags to the tool to allow programmers to disable security warnings for a line of code. For example, putting `@SuppressSecurity:MSC02` 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.

The last notable result was the 28.6% false positive rate on the IDS11-J rule which recommends to 'Perform any string modifications before validation' (Mohindra, 2016). The method of detection for IDS11-J is to look for strings that are modified after they have been validated [using the `Pattern.matcher()` method], but in two instances in SecuriBench the strings were validated a second time after modification thus fixing the potential security vulnerability. A modification to the rule implementation that looks for additional validation after a string is modified would eliminate the IDS11-J false positives in SecuriBench.

6.1.2 False negative study

A false negative analysis of the secure coding assistant requires segments of Java source code with known vulnerabilities. The SecuriBench programs have a large number of known vulnerabilities but a detailed listing of where they are in the source code does not exist. For this reason, a limited false negative analysis of the secure coding assistant was performed by looking for examples of insecure Java code from organisations 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 (OWASP Foundation, 2016) for preventing SQL injection attacks in Java in Figure 8. The structure of the code is almost identical to that of the CERT examples so it was not a surprise that the tool detected the vulnerability without an issue, as shown in Figure 9.

Figure 8 SQL injection example from OWASP website (see online version for colours)

```
 String query = "SELECT account_balance FROM user_data WHERE user_name
     = "
   + request.getParameter("customerName");
 try {
  Statement statement = connection.createStatement( âŁ¦ );
  ResultSet results = statement.executeQuery( query );
}
```

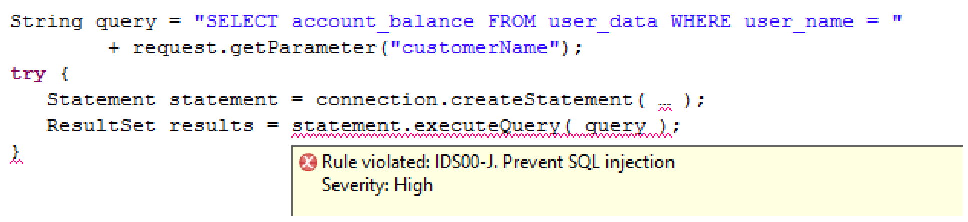
Source: OWASP Foundation (2016)

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. Sanitise untrusted data passed to the Runtime.exec() method	2	0.0%
L1	IDS01-J. Normalise 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 serialisation methods	0	0.0%
L1	STR00-J. Do not 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*	

Next, the CWE library was searched for code that would relate to the IDS01-J rule to normalise strings before validation (Mohindra, 2014). Figure 10 from the CWE dictionary (The Mitre Corporation, 2015b) is in the 'validate before canonicalise' section but is similar to the IDS01-J rule to validate before normalising 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 canonicalisation in the category of normalisation but the CERT rule only gives the example of the normalise method. With a small modification to the rule detection logic canonicalisation could be detected as well.

Figure 9 Output of SQL injection detection (see online version for colours)



```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
        + request.getParameter("customerName");
try {
   Statement statement = connection.createStatement( ... );
   ResultSet results = statement.executeQuery( query );
}
```

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

Figure 10 Validate before canonicalise example from CWE (see online version for colours)

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

Source: The Mitre Corporation (2015b)

Figure 11 Comparison of classes by name from CWE (see online version for colours)

```
public class TrustedClass {
  @Override
  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;
  }
}
```

*Source:*The Mitre Corporation (2015b)

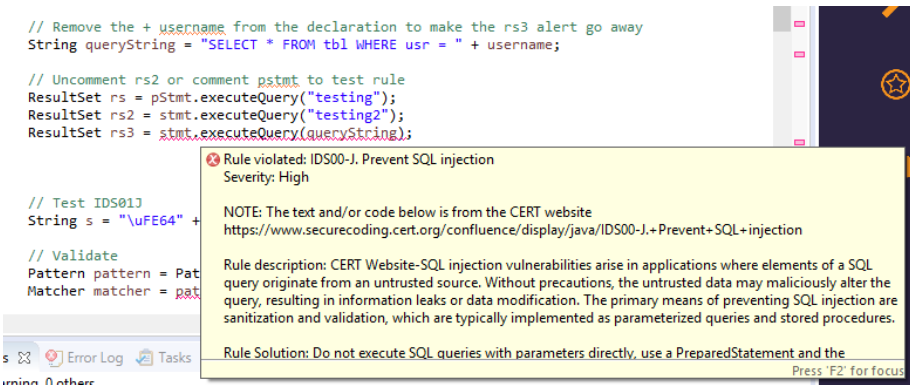
Another code segment from CWE is shown in Figure 11 which illustrates a vulnerability that should be detected under the CERT OBJ09-J rule (Gale, 2016). OBJ09-J states that class comparison should be done using the == operator on the class objects themselves and not the class names. In the example given, changing the comparison line to `obj.getClass() == this.getClass()` would rectify the problem. In this example the secure coding assistant successfully detected the vulnerability.

6.2 Validation

6.2.1 CERT

The CERT website lists several code samples for each secure coding rule along with the rule definition (Shrum, 2015). The samples are presented in pairs, first is an example of a violation of the rule and next is the corrected code segment. Figure 12 shows the secure coding assistant detecting an IDS00-J violation in a code segment taken from the CERT website (Mohindra, 2015a). 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.

Figure 12 IDS00-J violation from CERT detected with secure coding assistant (see online version for colours)



```
// Remove the + username from the declaration to make the rs3 alert go away
String queryString = "SELECT * FROM tbl WHERE usr = " + username;

// Uncomment rs2 or comment pstmt to test rule
ResultSet rs = pStmt.executeQuery("testing");
ResultSet rs2 = stmt.executeQuery("testing2");
ResultSet rs3 = stmt.executeQuery(queryString);


// Test IDS01J
String s = "\uFE64" +

// Validate
Pattern pattern = Pat
Matcher matcher = pat
```

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

NOTE: The text and/or code below is from the CERT website
https://www.securecoding.cert.org/confluence/display/java/IDS00-J.+Prevent+SQL+injection

Rule description: CERT Website-SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures.

Rule Solution: Do not execute SQL queries with parameters directly, use a PreparedStatement and the

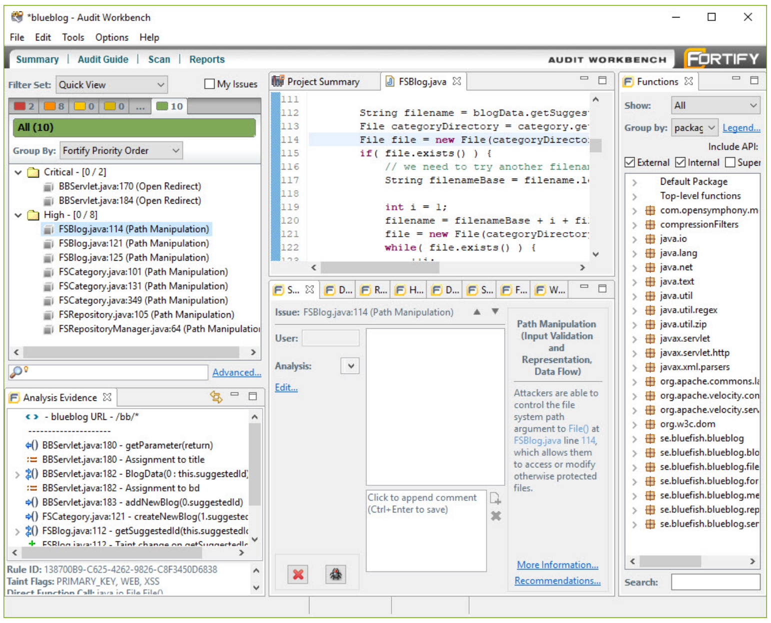
6.2.2 Fortify static code analyser

The Fortify static code analyser by Micro Focus (2018), formerly a software division of Hewlett Packard Enterprise, is a widely used and well-maintained tool which has similar features as the secure coding assistant. This is a sophisticated commercial product that assists with secure software development in many ways. Fortify is bundled with seven reports, a centralised workspace for developers to assign and collaborate tasks from static scan results, and very detailed analysis of code alerts. A screen shot of the Fortify audit workbench is included to illustrate the features that are included as Figure 13.

To compare the secure coding assistant to fortify, the Stanford SecuriBench (Livshits et al., 2005) was used. This project has seven applications written in Java that are known to have vulnerabilities. Using the OWASP top 10 (OWASP Foundation, 2018) vulnerabilities categories was necessary to compare the results of Fortifu to the secure coding assistant. Fortify already has a grouping for OWASP top 10, but the secure coding assistant alerts needed to be categorised first. In Section 6.1.1, a false positive study of the secure coding assistant is done using SecuriBench. The CERT rules (Shrum, 2015) that have at least one alert from this study, see Table 4, have been classified according to the OWASP Top 10 in Table 6.

Rules IDS00-J, IDS11-J and IDS07-J (Shrum, 2015) are in the ‘A1 - injection’ category as they all pertain to sanitisation of data which help to prevent injection attacks. ERR08-J, MET04-J, MET06-J and MSC02-J (Shrum, 2015) are in the ‘A3 - sensitive data exposure’ category since violating any of these rules could potentially expose sensitive data to an adversary. One of the rules, EXP00-J (Mohindra, 2015c), could not be categorised and has been listed with an ‘NA’ reference. There are some rules that Fortify has put in this category as well.

Figure 13 The Fortify results of blueblog scan viewed through their audit workbench (see online version for colours)



```
Critical - [0 / 2]
    BBServlet.java:170 (Open Redirect)
    BBServlet.java:184 (Open Redirect)
High - [0 / 8]
    FSBlog.java:114 (Path Manipulation)
    FSBlog.java:121 (Path Manipulation)
    FSBlog.java:125 (Path Manipulation)
    FSCategory.java:101 (Path Manipulation)
    FSCategory.java:131 (Path Manipulation)
    FSCategory.java:349 (Path Manipulation)
    FSRepository.java:105 (Path Manipulation)
    FSRepositoryManager.java:64 (Path Manipulatio
```

```
 - blueblog URL - /bb/*
 ---------------------
 BBServlet.java:180 - getParameter(return)
 BBServlet.java:180 - Assignment to title
 BBServlet.java:182 - BlogData(0 : this.suggestedId)
 BBServlet.java:182 - Assignment to bd
 BBServlet.java:183 - addNewBlog(0.suggestedId)
 FSCategory.java:121 - createNewBlog(1.suggestec
 FSBlog.java:112 - getSuggestedId(this.suggestedIc
 FSBlog.java:112 - Taint change on getSuggestedIc
```

Rule ID: 138700B9-C625-4262-9826-C8F3450D6838
Taint Flags: PRIMARY_KEY, WEB, XSS
Direct Function Call: java.io.File.File()

```
111
112         String filename = blogData.getSugges
113         File categoryDirectory = category.ge
114         File file = new File(categoryDirecto
115         if( file.exists() ) {
116             // we need to try another filena
117             String filenameBase = filename.l
118
119             int i = 1;
120             filename = filenameBase + i + fi
121             file = new File(categoryDirector
122             while( file.exists() ) {
```

Issue: FSBlog.java:114 (Path Manipulation)

Path Manipulation (Input Validation and Representation, Data Flow)

Attackers are able to control the file system path argument to File() at FSBlog.java line 114, which allows them to access or modify otherwise protected files.

Table 5 False positive analysis

Rule	*Total count*	*True pos. count*	*True pos. %*	*False pos. count*	*False pos. %*
EXP00-J	3,211	3,211	100.0%	0	0.0%
ERR08-J	740	740	100.0 %	0	0.0%
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	*4,172*	*4,153*	*99.5%*	*19*	*0.5%*

The Fortify static code analyser has three primary run modes: ‘show me all issues that may have security implications’, ‘show me likely problems’, and ‘show me only remotely exploitable issues’. In the analysis results these three modes have been labeled ‘A’, ‘B’, and ‘C’, respectively. The purpose of the multiple run modes is to allow the developer or security auditor to tailor the types of alerts to the type of environment in which the application is running. If, for instance, the application is running on a highly secure network then options ‘B’ or ‘C’ may be more appropriate. If, on the other hand, the application is running on the internet in an unsecured environment then option ‘A’ which produces the most scan results would be most appropriate. The secure coding assistant only has one run mode and is labeled as ‘SCA’ in the analysis results.

Table 6 OWASP Top Ten 2017 descriptions (OWASP Foundation, 2018) and secure coding assistant alert mappings

Category	*Description*	*SCA mapping*
A1	Injection	IDS00-J, IDS11-J and IDS07-J
A2	Broken authentication	
A3	Sensitive data exposure	ERR08-J, MET04-J, MET06-J and MSC02-J
A4	XML external entities (XXE)	
A5	Broken access control	
A6	Security misconfiguration	
A7	Cross-site scripting (XSS)	
A8	Insecure deserialisation	
A9	Using components with known vulnerabilities	
A10	Insufficient logging and monitoring	
NA	Could not categorise	EXP-00

In Table 7, the scan results are categorised using relative severity. With the Fortify tool, all results from SecuriBench were either in a ‘critical’ or ‘high’ category. The CERT rules have a similar ranking but with levels 1 through 3 (Shrum, 2015). The alerts from the secure coding assistant were all in the level 1 ‘L1’ or level 2 ‘L2’ categories. Relative to Fortify, the secure coding assistant identified many more potential rule violations in the ‘high/L2’ category than the ‘critical/L1’ category. The comparison using OWASP top 10 (OWASP Foundation, 2018) is shown in Table 8. The Fortify tool was able to identify potential violations in seven categories and the secure coding assistant was able to identify violations in two categories. Though it is evident through this analysis that the secure coding assistant does not perform as well as a commercially available product, the intention is not to compete commercially. Rather, the goal is to provide a free and open source tool to the community with an emphasis educating software developers in secure coding practices. Additionally, for those rules that mapped to the OWASP categories the secure coding assistant had close to the same number of alerts as Fortify. The results of this analysis will be used to identify CERT rules that can be implemented in a subsequent release.

Table 7 Fortify priority and CERT level comparison table

Fortify priority/CERT level	*Fortify A*	*Fortify B*	*Foritfy C*	*SCA*
Crtical/L1	1,184	1,153	551	823
High/L2	447	435	369	3,349
Total	*1,631*	*1,588*	*920*	*4,172*

Table 8 OWASP Top Ten 2017 comparison table

OWASP top ten 2017	*Fortify A*	*Fortify B*	*Foritfy C*	*SCA*
A1	224	214	145	51
A2	1	1	1	0
A3	423	423	422	910
A4	1	1	1	0
A5	165	138	119	0
A6	6	6	6	0
A7	772	772	199	0
A8	0	0	0	0
A9	0	0	0	0
A10	0	0	0	0
NA	39	33	27	3,211
Total	*1,631*	*1,588*	*920*	*4,172*

6.3 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 five SecuriBench source code files three times with the plugin enabled and three times with the plugin disabled. When testing with the plugin enabled the alert list was cleared after each test. After each load, the total delay was recorded and then the total delay for all three loads were averaged together. 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 9 show that the plugin added an additional 0.03 to 0.20 seconds to the load time for each source file. The additional processing is in a separate thread so the impact to the user is minimal. While the plugin is processing the source file the alert window is filling with secure coding alerts which does not interfere with the user's ability to scroll through the file and make edits. 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 9 Plugin efficiency analysis

Application	*Source file*	*Alerts*	*Increase (sec.)*	*Time per alert (ms)*
pebble	SimpleBlog.java	46	0.2037	4.428
roller	WebLogEntryFormAction.java	16	0.0713	4.458
webgoat	CreateDB.java	49	0.1923	3.925
snipsnap	ConfigurationMap.java	23	0.0270	1.174
snipsnap	ConfigurationProxy.java	19	0.0380	2.000

The additional time per alert is at most less than 5 milliseconds which would not be noticable unless there were several hundred alerts in a single source code document. Additionally, while coding with and without the plugin enabled there is no noticeable difference in performance. No lag time could be observed while coding with the plugin enabled.

7 Limitations, conclusions and future works

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. Future plans also include incorporating the tool in the computer science course curriculum to test effectiveness. Specifically, the students in the introductory Java programming courses will benefit from this tool the most.

The SecuriBench testing showed that some rules like EXP00-J 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 several little 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 specialised security. There are still several more rules that can be implemented though and the data obtained in Section 6.2.2 will be used as a guide for identifying which rules to add to the tool.

The tool could also benefit from a few small improvements to the functional design. The markers, for instance, contain information from the CERT website (Shrum, 2015) but they do not have a hyperlink back to the website itself. A precursory review of the marker structure found that customising the marker text to have hyperlinks would be possible but requires a fair amount of additional design and implementation work. Also some additional controls to the user interface to control the scanning of the source code files would be useful. For instance, the plugin is always scanning the open file in the background and a programmer may want to scan an entire workspace at once, pause it and monitor the status. Lastly, there are instances where a programmer may see an alert and not agree that it is a security concern as was the case in the many EXP00-J and MSC02-J alerts in the SecuriBench test. In these cases, it would be very helpful for the programmer to have a way to indicate that they would like to ignore a particular rule in a block of code.

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 continue to be maintained. For more detailed information, including a tool demo, please visit the project website at http://benw408701.github.io/SecureCodingAssistant/.

Acknowledgements

This publication incorporates portions of the “SEI CERT Oracle Coding Standard for Java” ©2017 Carnegie Mellon University with special permission from its Software Engineering Institute.

References

Aldausari, N., Zhang, C. and Dai, J. (2018) ‘Combining design by contract and inference rules of programming logic towards software reliability’, in *Proceedings of the 15th International Conference on Security and Cryptography (SECRYPT 2018) (to appear, accepted)*, July.

Checkmarx, Ltd. (2018) *Static Application Security Testing* [online] https://www.checkmarx.com/products/static-application-security-testing/ (accessed 4 May 2018).

Contrast Security (2018) *Contrast for Eclipse* [online] https://marketplace.eclipse.org/content/contrast-eclipse (accessed 4 May 2018).

Díaz, G. and Bermejo, J.R. (2013) ‘Static analysis of source code security: assessment of tools against SAMATE tests’, *Information and Software Technology*, Vol. 55, No. 8, pp.1462–1476.

Dehlinger, J., Feng, Q., Oestrich, E. and Smith, M. (2012) *SSV Checker – An Eclipse Plug-in Interface Static Security Vulnerability Checker*, 26 August [online] http://ssvchecker.sourceforge.net/ (accessed 15 November 2015).

Elgin, B., Riley, M. and Lawrence, D. (2014) ‘Hacked wide open.(home depot fails to improve security)’, *Bloomberg Businessweek*, 22 September, pp.39–40.

Fisher, D. (2003) ‘Appscan tests for vulnerabilities during the development cycle’, *eWeek*, Vol. 20, No. 7, p.23.

Frates, C. and Devine, C. (2014) *Government Hacks and Security Breaches Skyrocket*, CNN Wire, 19 December.

Gale, A. (2016) *OBJ09-J. Compare Classes and not Class Names*, 24 May [online] https://www.securecoding.cert.org/confluence/display/java/OBJ09-J.+Compare+classes+and+not+class+names (accessed 12 July 2016).

Gelsomini, J.J. and Garcia, K.H. (2015) ‘Anthem’s data breach impacts many anthem and non-anthem plans: necessary employer actions now’, *Employee Benefit Plan Review*, Vol. 69, No. 11, pp.5–7.

HT Media Ltd. (2014) ‘Fewer than 1% of engineering students skilled in secure programming’, *Mint*, February.

Katsis, G. (2016) *MSC02-J. Generate Strong Random Numbers*, 13 May [online] https://www.securecoding.cert.org/confluence/display/java/MSC02-J.+Generate+strong+random+numbers (accessed 12 July 2016).

Li, C., White, B., Dai, J. and Zhang, C. (2017) ‘Enhancing secure coding assistant with error correction and contract programming’, in *Proceedings of the National Cyber Summit*, June.

Lindeman, T.F. (2013) ‘Target acknowledges security breach; 40 million accounts compromised’, *McClatchy – Tribune Business News*, 20 December.

Livshits, B., Martin, M., Lam, M., Whaley, J., Avots, D., Carbin, M. and Unkel, C. (2005) *Stanford SecuriBench*, 21 December 2005 [online] http://suif.stanford.edu/~livshits/securibench/intro.html (accessed 23 December 2015).

Melnik, V. (2018) *Enforcing Secure Coding Rules for the C Programming lLanguage using the Eclipse Development Environment*, in Master Project, California State University Sacramento.

Micro Focus (2018) *Static Analysis, Static Application Security Testing*, https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview (accessed 27 April 2018).

Mohindra, D. (2014)*IDS01-J. Normalize Strings Before Validating Them*, 27 November [online] https://www.securecoding.cert.org/confluence/display/java/IDS01-J.+Normalize+strings+before+validating+them (accessed 12 July 2016).

Mohindra, D. (2015a) *IDS00-J. Prevent SQL Injection*, 3 November [online] https://www.securecoding.cert.org/confluence/display/java/IDS00-J.+Prevent+SQL+injection (accessed 22 December 2015).

Mohindra, D. (2015b) *NUM03-J. Use Integer Types that can Fully Represent the Possible Range of Unsigned Data*, 3 June [online] https://www.securecoding.cert.org/confluence/display/java/NUM03-J.+Use+integer+types+that+can+fully+represent+the+possible+range+of++unsigned+data (accessed 1 November 2015).

Mohindra, D. (2015c) *EXP00-J. Do not Ignore Values Returned by Methods*, 3 November [online] https://www.securecoding.cert.org/confluence/display/java/EXP00-J.+Do+not+ignore+values+returned+by+methods (accessed 23 December 2015).

Mohindra, D. (2016) *IDS11-J. Perform any String Modifications Before Validation*, 2 March [online] https://www.securecoding.cert.org/confluence/display/java/IDS11-J.+Perform+any+string+modifications+before+validation (accessed 12 July 2016).

OWASP Foundation (2016) *Preventing SQL Injection in Java*, 25 May [online] https://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java (accessed 26 June 2016).

OWASP Foundation (2018) *Top 10-2017 Top 10*, 27 March [online] https://www.owasp.org/index.php/Top_10-2017_Top_10 (accessed 8 April 2018).

Pandit, M.K. (2013) 'Developing secure software using aspect oriented programming', *IOSR Journal of Computer Engineering*, Vol. 10, No. 2, pp.28–34.

Red Lizard Software (2015) *Red Lizard Software – Goanna C/C++ Static Analysis* [online] http://redlizards.com/ (accessed 15 November 2015).

Rogue Wave Software (2018) *Introducing Static Code Analysis with Continuous Integration* [online] https://www.roguewave.com/programs/ppc/static-analysis-code (accessed 4 May 2018).

Sampaio, L. (2015) *The Code Master*, 17 June [online] http://thecodemaster.net/ (accessed 15 November 2015).

Shrum, S. (2015) *2 - Rules - CERT Oracle Coding Standards for Java*, 7 April [online] https://www.securecoding.cert.org/confluence/display/java/2+Rules (accessed 14 November 2015).

SonarSource S.A. (2018) *SonarLint* [online] https://www.sonarlint.org/ (accessed 4 May 2018).

Svoboda, D. and Hicken, A. (2015) *ERR08-J. Do not Catch NullPointerException or Any of its Ancestors*, 3 November [online] https://www.securecoding.cert.org/confluence/display/java/ERR08-J.+Do+not+catch+NullPointerException+or+any+of+its+ancestors (accessed 23 December 2015).

Synopsys, Inc. (2018) *SecureAssist Overview and Datasheet* [online] https://www.synopsys.com/software-integrity/resources/datasheets/secureassist.html (accessed 4 May 2018).

The Mitre Corporation (2015a) *CVE - download CVE*, 13 November [online] https://cve.mitre.org/cve/cve.html (accessed 15 November 2015).

The Mitre Corporation (2015b) *CWE-2000: Comprehensive CWE Dictionary*, 8 December [online] http://cwe.mitre.org/data/slices/2000.html (accessed 6 February 2016).

University of Maryland (2018) *FindBugs – Find Bugs in Java Programs* [online] http://findbugs.sourceforge.net/ (accessed 4 May 2018).

Vamialis, A. (2013) 'Online service providers and liability for data security breaches', *Journal of Internet Law*, Vol. 16, No. 11, pp.23–33.

Veracode (2018) *Static Analysis (SAST)*, 2017 [online] https://www.veracode.com/products/binary-static-analysis-sast (accessed 4 May 2018).

White, B., Dai, J. and Zhang, C. (2016) 'Secure coding assistant: enforcing secure coding practices using the eclipse development environment', in *Proceedings of the National Cyber Summit*, June.

White, B. (2016) *Secure Coding Assistant: Enforcing Secure Coding Practices using the Eclipse Development Environment*, in Master Project, California State University Sacramento.

WhiteHat Security (2018) *Static Application Security Testing (SAST)* [online] https://www.whitehatsec.com/products/static-application-security-testing/ (accessed 4 May 2018).

Xie, J., Chu, B., Lipford, H.R. and Melton, J.T. (2011) 'ASIDE: IDE support for web application security', *ACSAC 2011*, December.

Zhu, J., Xie, J., Lipford, H.R. and Chu, B. (2014) 'Supporting secure programming in web applications through interactive static analysis', *Journal of Advanced Research*, Vol. 5, No. 4, pp.449–462.

Notes

1 Certain acknowledgements and attributions have been made to Carnegie Mellon University and its Software Engineering Institute in the 'Acknowledgements' section of this paper. This is an extended version of a paper published in the National Cyber Summit conference proceedings (White et al., 2016) as well as a result of a master's project (White, 2016), for more information see White (2016).