Enforcing Secure Coding Rules for the C Programing Language Using the Eclipse Development Environment

Victor Melnik[1], Jun Dai[1], Cui Zhang[1], Benjamin White[12]

[1] Computer Science, California State University, Sacramento, CA 95819
[2] Mother Lode Holding Company, Roseville, CA 95747
jun.dai@csus.edu

Abstract. Creating secure software is challenging, but necessary due to the prevalence of large data breaches that have occurred for organizations such as Equifax, Uber, and U.S. Securities and Exchange Commission. Many static analysis tools are available that can identify vulnerable code, however many are proprietary, do not disclose their rule set or do not integrate with development environments. One open source tool that integrates well with the Eclipse development environment is the *Secure Coding Assistant* that was developed at California State University, Sacramento (CSUS), which is featured by early error detection. The tool provides support for secure coding rules for the Java programming language that were developed at the CERT division of the Software Engineering Institute at Carnegie Mellon University. The tool also provides error correction and contract programming support. To provide secure coding assistance in C programming, we further extend the tool to support the C programming language by semi-automating a subset of the CERT secure coding rules for C. The tool detects rule violations for the C programming language in the Eclipse development environment and provides feedback to aid and educate developers in secure coding practices. The tool is *open source* to the community and maintained at *GitHub* (http://benw408701.github.io/SecureCodingAssistant/).

Keywords: Secure Coding, Software Security, C Programming

1 Introduction

Developing software using secure coding practices is becoming increasingly important as the frequency and severity of data breaches continue to rise. According to the Identity Theft Resource Center, 2017 set a record of the highest number of data breaches in the United States of America, with an increase of 44.7% compared to the previous year [1]. In 2017 the world also observed some of the largest data breaches to date. For instance, in the beginning of 2017, Uber disclosed that 57 million Uber users and driver's information was stolen, which included "names, email addresses, phone numbers, driver's license numbers", and other personal information [2]. Later that year the largest data breach to date occurred at Equifax, a consumer credit reporting agency. Hackers were able to steal "145.5 million records containing social security numbers, names, ad-

dresses, credit card numbers and other personal information" [7]. Lastly, the U.S. Securities and Exchange Commission's Electronic Data Gathering, Analysis, and Retrieval (EDGAR) system was infiltrated and information regarding mergers, acquisitions and other company data was exfiltrated [3]. The severity of this data breach is difficult to assess, because the data retrieved could be used in the future to make millions to billions of dollars for criminal organizations. Many of these attacks could have been mitigated or prevented if the organizations enforced more stringent coding practices.

There are many vulnerabilities that are reported and published on the Common Vulnerability Enumeration (CVE) website. It would take a good deal of effort to keep up with ever newly published vulnerability. In 2017 alone, 14,712 CVEs were published [12]. This was an unprecedented spike in code vulnerabilities compared to 2016, where only 6,447 CVEs were published [12]. According to IEEE Senior Member Gary McGraw, "there has been too much focus on common bugs and not enough on secure design and avoidance of flaws" [13].

To stay ahead of the curve of newly published vulnerabilities, various tools were developed to provide code weakness detection and secure coding assistance. Our tool named *Secure Coding Assistant* is one of these efforts, which is open source and implements the CERT secure coding rules for Java programming language [7] [18-19]. It is a static analysis tool that was developed in 2016 [18-19] and later enhance in 2017 at [7]. The tool, featured by early detection, provides support for the CERT secure coding rules for the Java language. It also provides error correction and contract programming for the Java language. The rules were developed at the CERT division of the Software Engineering Institute at Carnegie Mellon University. By enforcing the rules throughout coding, newly developed software can avoid common security pitfalls.

This paper is focused on the enhancement of the tool by semi-automating the secure coding rules for C programming language. To achieve this goal, a subset of the CERT secure coding rules for C will be carefully selected and implemented. Specifically, the tool will flag unsecure code segments similar to problem markers generated during the compilation process. These markers will provide the developer with the name of the violated rule and information on how to remediate the vulnerable code. These problem markers will help educate software developers on secure coding principles.

Throughout this paper, the enhancement to provide support for the C language to the Secure Coding Assistant will be referred to as the *Secure Coding Assistant for C*. *Secure Coding Assistant for Java* will be used to refer to the original software that was developed for the Java language. The Secure Coding Assistant for C and Secure Coding Assistant for Java are integrated as part of the same tool but are mutually exclusive components within the tool, due to their inherent difference in programming language.

2 Related Work

There are currently many static analysis tools that are available to aid developers in making secure software. Table 1 provides a list of some of these available tools. The first five are commercial tools while the rest are open source ones.

All the tools that are closed source do not disclose the rule set or the methodologies that are used to detect vulnerabilities in the developer's source code. The first four open source tools, scan source code for vulnerabilities but do not disclose which rule set the tool is based on. Also, two of these open source tools have not been updated for a few years. *VisualCodeGrepper* has not been updated in the past two years, while *PreFast* has not been updated since 2005. The tool that is most closely related to our tool in *Flawfinder*. *Flawfinder* is an open source tool that is available for download on GitHub. *Flawfinder* is based on the Common Weaknesses Enumeration (CWE) database and detects vulnerable code segments by matching code against a database of C/C++ functions with known problems. Unlike *Flawfinder*, Secure Coding Assistant is based on an established secure coding rule set and does not rely on new vulnerabilities to be published to update the tool. Secure Coding Assistant will be maintained and further developed by the Department of Computer Science at CSUS.

Table 1. Current Secure Code Analysis Tools.

Company	Tools	Rule Set	Open/Closed
Synopsys	Coverity Static Analysis Tool	Proprietary	Closed
Veracode	Static Analysis SAST	Proprietary	Closed
Rouge Wave Software	KlocWork	Proprietary	Closed
Viva64	PVS-Studio Analyzer	Proprietary	Closed
Micro Focus	Fortify Static Code Analyzer	Proprietary	Closed
Microsoft	PreFast	Custom	Open
NCC Group	Visual Code Grepper	Custom	Open
Michael Scovetta	Yasca	Custom	Open
Daniel Marjamäki	CPPCheck	Custom	Open
David Wheeler	Flawfinder	CWE	Open

3 Design

3.1 Goals

There are two goals that are expected by enhancing the Secure Coding Assistant. The first goal is to provide developers with feedback when compiling their source code. This will be similar to warnings and error problem reports that are generated during the compilation process. This feedback will allow developers to mitigate security vulnerabilities during the development of their software.

The second goal is to educate developers on secure coding practices for the C language. This goal will be accomplished by providing developers with problem alerts that provide a clear message that specifies the violated rule and guideline on how to remediate the unsecure code segment. These two implemented goals will create a learning environment that will educate software developers on the secure coding practices for the C language.

3.2 Architecture

The Secure Coding Assistant for C runs when the *build* command in Eclipse is called. The *build* command is used to compile all the C source code files within an open project. Eclipse refers to source code files that are inputted into a compiler as translation units. As the *build* command runs, all the nodes in the translation unit are analyzed to determine if any rules are violated. Fig. 1 shows the high-level flow on the overall design for the Secure Coding Assistant for C. When the *build* command is called all the pre-existing markers in the source code are cleared, and the first node within the first translation unit is visited. If a rule is violated in the node, a *marker* is generated with the name of the rule violated and its remediation information. Then the next node in the translation unit is visited. This process continues until all the nodes in the translation unit have been visited and analyzed. If there are more translation units in that need to be compiled, the next translation unit is visited, and all its node are subsequently analyzed. Once all the translation units within the project are visited and analyzed, the Secure Coding Assistant for C displays all the *markers* that have been created during the *build* processes. The Secure Coding Assistant for C will run and display all the problem markers in the project's translation units, even if the *build* fails to compile the project successfully.

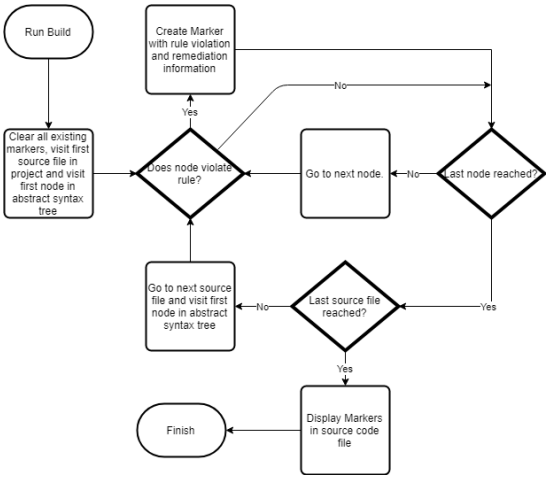

Fig. 1. Secure Coding Assistant for C High-level Flow Chart.

4 Implementation

The idea to use the Eclipse Development Environment as the common platform decides that the Secure Coding Assistant for C and the Secure Coding Assistant for Java could share methodologies for implementation. The difference between the two analyzers is mainly that they utilize a different Eclipse tooling library. Specifically, the Secure Coding Assistant for C utilizes the Eclipse C/C++ Development Tooling (CDT) library, while the Secure Coding Assistant for Java utilizes the Eclipse Java Development Tooling (JDT) library.

4.1 Rule Selection

The CERT secure coding standard provides a total of 120 rules for C which are divided into 17 specific categories. To determine which rules are to be incorporated into the Secure Coding Assistant for C, the rules are first divided into two categories: rule that could be automated and rules that could not be automated. For a majority of the C secure coding rules, the CERT website provides information on whether the rule can be automated or not.

An example of a rule that could not be automated is the *FIO32-C* rule, which states to not perform file operations on devices that are only appropriate for files [15]. In the UNIX and Windows operating systems, special files are used to represent devices. To determine if this rule was violated, the tool would require a mechanism of identifying each file as it was inputted into a file operation function. Since this information could only be gathered during runtime, this rule could not be automated in a static analysis tool.

Additionally, the CERT secure coding standard for C contained three rule categories that did not contain any rules that could be automated. One of these rule categories is the *Preprocessor* category. The *Preprocessor* rule category could not be automated due to the limitation of the Eclipse CDT library. The library did not provide a method to analyze preprocessor code segments in a translation unit. This limitation prevented the tool from being able to automate any of the rules within this rule category.

From the 120 CERT rules for C, 38 were determined to be automatable. From the 38 rules that were determined to be automatable 20 rules were selected to be automated in the tool. The 20 rules that were selected for this tool were determined based on their severity, and the likelihood that the rule violation would occur. The CERT website provided the classification for each rule. Additionally, rules were also selected to represent all the 17 rule categories that did contain automatable rules.

4.2 Plugin Implementation

To develop the Secure Coding Assistant for C, the Eclipse Plugin Development Environment (PDE) was utilized. The Eclipse PDE provides developers with extension points that can be used to improve and customize the existing development environment. Extension points are a combination of XML mark-up language and a Java interface, that allow for one plugin to extend and customize the functionality of another plugin [4].

The Secure Coding Assistant for C extends one extension point. The extension point is *org.eclipse.cdt.core.ErrorParser*. This extension point allows the plugin to fulfil two functions. First, it allows the plugin to interact with the C *build* process. *Build* is used to compile and link the source files in an open project. Second, it allows for the generation of problem markers. Problem markers are used to mark the segment of code that contains a rule violation and provide a tool-tip that contains information on the violated rule and how to remediate the unsecure code.

4.3 Abstract Syntax Tree

Each translation unit in a C project is represented as an *Abstract Syntax Tree* (AST). An AST is a tree model that is used to represent the structure of a programming language's source code file. An AST can be traversed depth-first from top to bottom or bottom to top.

The Eclipse CDT library provides a mechanism to examine the AST through the *org.eclipse.cdt.core.dom.ast* package. To traverse the AST, the *org.eclipse.cdt.core.dom.ast* package provides the class *ASTVisitor*. *ASTVisitor* provides a *visit()* method for each of the different types of nodes (variable declaration, expression statement, function parameters, etc.). The *visit()* method allows for each node within a translation unit to be visited and examined.

The Secure Coding Assistant for C has two classes that extend the *ASTVisitor* class: *SecureCodingNodeVisitor_C* and *ASTNodeProcessor*. *SecureCodingNodeVisitor_C* class is used to access the AST during the build process. *ASTNodeProcessor* class is used by the *Utility_C* library to aid in the detection of rule violations.

4.4 Rule Detection

The Secure Coding Assistant for C uses two Java classes to fulfil the task of detecting rule violations: *ASTNodeProcessor_C*, and *Utility_C*.

ASTNodeProcessor_C is at the heart of rule detection. *ASTNodeProcessor_C* traverses the AST of a translation unit a second time and creates collections of various node types such as variable declarations, function definitions, assignment statements, etc. *ASTNodeProcessor_C* also assigns a numerical value to each node to keep track of the order in which the nodes appear in the source code. These collections of nodes allowed for easy retrieval of nodes that were called before and after the node being currently analyzed.

Table 2. *Utility_C* Library.

Utility	Method
Get scope of node	getScope(IASTNode)
Determine if inner node is contained within outer node	isEmbedded(IASTNode, IASTNode)
Get list of all variables in the same scope as the node	allVarNameType()
Get list of function call parameter	getFunctionParamaterVarName()
Get list of function call parameters for printf functions	getFunctionParameterVarNamePrintf()

Utility_C library is a collection of methods that are used by more than one rule. Since many of the CERT rules share common rule detection logic, *Utility_C* library was used to simplify the logic for each rule. This library created a list of methods that could be used by future developers to expand the tool. The list of methods in the *Utility_C*, along with the purpose they serve is show in Table 2. The *Utility_C* library was expanded

during the development of the Secure Coding Assistant for C tool. A new method was added when more than one rule was determined to share similar rule detection logic. Using both the *ASTNodeProcessor_C* class and the *Utility_C* library simplified the rule logic for each rule and allows for code reusability.

4.5 Rule Interface

Each rule implements the *SecureCodingRule_C* interface. The interface provides methods for detecting a rule violation and for provide feedback to the user of the tool. Table 3 provides the methods contained in the *SecureCodingRule_C* interface.

Table 3. *SecureCodingRule_C* Interface [18].

Method Signature	Description
Boolean violated_CDT(IASTNode)	Checks to see if a rule has been violated for a node
String getRuleText()	The description of the violated rule
String getRuleName()	The description of the violated rule
String getRuleID()	The ID of the violated rule
String getRuleRecommendation()	Suggestions to remediate the insecure node
Int securityLevel()	The security level of the violated rule: HIGH, MEDIUM, LOW
String getRuleURL()	The URL to the rule on the CERT website

This interface is borrowed from the Secure Coding Assistant for Java developed by [18-19]. However, since both tools use different Eclipse development libraries, the *SecureCodingRule_C.violated()* function is modified to accommodate the difference.

The *SecureCodingRule_C.violated()* method takes one parameter, i.e. the node that is currently being processed by the *SecureCodingNodeVisitor_C*. The node is analyzed by the method and returns true if the rule has been violated. This method made the code required for running each rule against all the nodes in a translation unit simple. Fig. 2 displays the rule traversal logic used in *SecureCodingNodeVisitor_C*.

5 Evaluation

5.1 Accuracy

5.1.1 CERT Validation

The CERT website provides a list of example code as well as the definitions for each of the CERT rules. Each rule contains a pair of code samples: one with a rule violation and one with the rule violation remediated. Some of the rules contained more than one pair of code examples. To initially develop the Secure Coding Assistant for C, the tool focused on detecting the rule violation in the unsecure code segments. It also made sure that any false positives were remediated during this process. Once the Secure Coding Assistant for C was able to detect all the rule violation in the CERT's rule sample code, the rule logic was considered to be complete.

5.1.2 False Positive

```
public void traverseRule(IASTNode checkNode)
{
    for (IRule_C rule : c_rules)
    {
        if(rule.violate_CDT(checkNode))
        {
            Globals.insecureGlobalNode = checkNode;
            Globals.cdt_InsecureCodeSegments.add(
                    new InsecureCodeSegment_C(checkNode,rule, localITU));
        }
    }
}
```

Fig. 2. Rule Detection Logic in *SecureCodingNodeVisitor_C*.

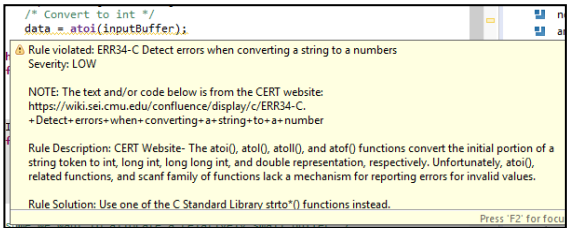

Fig. 3. *ERR34-C* rule violation from Juliet Test Suite for C/C++ detected by Secure Coding Assistant [11].

The *Juliet Test Suite* for C/C++ developed by the NSA Center for Assured Software was used to conduct a false positive study [11]. This test suite consists of 64,099 C/C++ source code files which are categorized under 118 different CWEs. Each source code file contains an unsecure code example paired with a secure code correction. The authors of the files provide comments within each file to identify the code segments that contain weaknesses. Many of the weaknesses that were documented in the Juliet Test Suite for C/C++ were not detected by the Secure Coding Assistant for C because most CWEs do not directly translate over to any CERT rules. For example, CERT does not include any rules for code weaknesses such as unchecked return values or unreachable code segments.

The Secure Coding Assistant for C generated 11,021 secure coding warning which are shown in Table 4. Ten of the 20 rules that were implemented in the tool detected rule violations. The top two rules that were detected are the *ERR34-C* and *MEM31-C* rules, which collectively account for 68% of all the rule violations. The *ERR34-C* rule states to detect errors when converting strings to a number [5]. This rule detects rule violations when using *string* to *integer* conversion functions that lack error reporting mechanism such as *atoi*, *atoll*, and *atoll* [5]. Fig. 3 shows an example of a rule violation for the *ERR34-C* rule with its accompanied problem alert window. The rule *MEM31-C* states that dynamically allocated memory should be freed once it is no longer needed by the program [16]. This rule was detected, since many CWEs are associated with memory leakage and corrupt memory pointers.

Table 4. Juliet Test Suite for C/C++ Results.

Level	Rule Name	Total	Percent
L3	ERR34-C. Detect errors when converting a string to a number	3784	34.3%
L2	MEM31-C. Free dynamically allocated memory when no longer needed	3750	34.0%
L2	INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors	2010	18.2%
L2	MSC30-C. Do not use the rand() function for generating pseudorandom numbers	812	7.4%
L1	ENV33-C. Do not call system()	557	5.0%
L2	ARR36-C. Do not subtract or compare two pointers that do not refer to the same array	36	0.3%
L2	FIO45-C. Avoid TOCTOU race conditions while accessing files	36	0.3%
L1	SIG30-C. Call only asynchronous-safe functions within signal handlers	18	0.2%
L2	SIG31-C. Do not access shared objects in signal handlers	18	0.2%
L2	FIO47-C. Use valid format strings	18	0.2%
L2	DCL36-C. Do not declare an identifier with conflicting linkage classifications	0	0.0%
L3	DCL38-C. Use the correct syntax when declaring a flexible array member	0	0.0%
L3	DCL41-C. Do not declare variables inside a switch statement before the first case label	0	0.0%
L2	EXP32-C. Do not access a volatile object through a nonvolatile reference	0	0.0%
L2	FLP30-C. Do not use floating-point variables as loop counters	0	0.0%
L2	STR34-C. Cast characters to unsigned char before converting to larger integer sizes	0	0.0%
L3	STR37-C. Arguments to character-handling functions must be representable as an unsigned char	0	0.0%
L1	STR38-C. Do not confuse narrow and wide character strings and functions	0	0.0%
L2	FIO47-C. Use valid format strings	0	0.0%
L2	CON40-C. Do not refer to an atomic variable twice in an expression	0	0.0%
L2	POS33-C. Do not use vfork()	0	0.0%
	Total	11,021	

Each rule detection in Table 4 was manually inspected to determine if the alert was a true positive or false positive. Table 5 displays the false positives that were identified. False positives accounted for 25% of all of the rule detections. Only two rules were determined to have false positive detections: the *INT33-C* and the *MSC30-C* rules.

The highest false positive result was attributed to the *INT33-C* rule. This rule states that "division and modular operations should not result in a divide-by-zero error" [14]. These false positives stem from floating point division, where a conditional statement checks to see if the divisor is greater than the value of .00001 before performing division. The rule logic in the tool is structured to check if the divisor is greater than zero, greater than or equal to one, or not equal zero. It would be difficult to account for the different variations of conditional statements that can be satisfied to check if a floating-point number is not equal to zero. This makes avoiding false positives for this rule difficult. This rule highlights that the rule detection logic for this rule should be revisited.

The second highest false positive result is attributed to the *MSC30-C* rule. This rule states to not use the function *rand()* to generate pseudorandom numbers for application that have a strong pseudorandom number requirement [9]. The false positive results found were in source files that were using *rand()* for purposes that did not need strong pseudorandom values. It would be difficult to fix the false positives that were generated by this rule, because it requires context into how these random number will be used in an application. Future release of the Secure Coding Assistant for C could provide the

option to hide a secure coding rule violation if there is disagreement with the tool. This would help minimize the number of false positive detections.

Table 5. False Positive Results.

Rule	Total Count	True Pos. Count	True Pos. (%)	False Pos. Count	False Pos. (%)
INT33	2,010	1,519	75.57	491	24.43
MSC30	812	505	62.19	307	37.81
Total	2,822	2,024	71.72	798	28.28

5.1.3 False Negative

To conduct a false negative study on the Secure Coding Assistant for C, the Juliet Test Suite for C/C++ [11] and the CWE website database [10] were used. These sources were used because they contained code segments that provided documented vulnerabilities. The false negative study was done by looking through both sources and determining if the documented vulnerability should have been picked up by the tool. The tool failed to detect rule violations for the *FIO45-C* and *STR34-C* rules.

The false negative instance for the *FIO45-C* rule was found in the Juliet Test Suite for C/C++. The *FIO45-C* rule states that a TOCTOU (time-of-check, time-of-use) race conditions should be avoided when more than one concurrent process is operating on a shared file system [17]. The code segment that should have been picked up by the tool is shown in Fig. 4. The Secure Coding Assistant for C did not flag this code segment as a vulnerability because the *#define* preprocessor directive was used to rename the file operations *stat* and *open* to *STAT* and *OPEN*, respectively.

```
if (STAT(filename, &statBuffer) == -1)
{
    exit(1);
}
fileDesc  = OPEN(filename, O_RDWR);
if (fileDesc == -1)
{
    exit(1);
}
```

Fig. 4. Code segment from Juliet Test Suite for C/C++ [11].

The false negative instance for the *STR34-C* rule was discovered on the CWE website under *CWE-843*: Access of Resource Using Incompatible Type [10]. *CWE-843* does not relate to the CERT rule *STR34-C*, however the CWE code example contained a segment of code that violated the *STR34-C* rule. The *STR34-C* rule states that *char* should be cast to an *unsigned char* before converting the value to a larger integer size [6]. Fig. 5 displays the code segment from *CWE-843* that should have been detected as a rule violation under the *STR34-C* rule. The character variable *defaultMessage* is cast to the integer *buf.nameID* without first casting the *char* to an *unsigned char*. Custom code was written to identify the variable being accessed using the member access operator for variables declared within complex data structures such as *union* and *struct*.

This code was written since the Eclipse CDT library lacked this mechanism. The logic failed to consider a complex data structure being nested within another complex data structure. This case was not considered because none of the CERT examples provided code segments where this case occurred. This is a limitation of the tool that will be addresses in future developments.

```
#define NAME_TYPE 1

struct MessageBuffer
{
int msgType;
union {
char *name;
int nameID;

};

};

int main (int argc, char **argv) {
struct MessageBuffer buf;
char *defaultMessage = "Hello World";

buf.msgType = NAME_TYPE;
buf.name = defaultMessage;
printf("Pointer of buf.name is %p\n", buf.name);


buf.nameID = (int)(defaultMessage + 1);
printf("Pointer of buf.name is now %p\n", buf.name);
}

}
```

Fig. 5. *CWE-843* code segment from CWE website [10].

5.2 Efficiency

The tool's efficiency was measured by running the *build* command against test suites from [11] and test files that were generate from the CERT website examples to initially test this tool. Each project was built 3 times with and without the tool enabled to gather the average build time. After each build, the *clean* command was called to delete all the generate binaries. The Secure Coding Assistant for C efficiency result are shown in Table 6. The second to last column in Table 6 shows the increase in time to build the binaries for a project. The time it takes to build a project appears to be correlated with the number of files in a project, as well as the number of detected violations. There is an average 4.45% increase in *build* time with the tool enabled.

Table 6. Efficiency Test Results.

Project	Files	Alerts	Time Increase (s)	Increase (%)
CERT	20	50	1.21	5.74
Test 45	66	13	3.48	15.84
Test 46	64	18	4.75	21.65
Test 101	58	29	5.42	9.04
Test 106	247	113	14.71	12.44

6 Limitations, Conclusion and Future Work

The enhancement to the Secure Coding Assistant for C programming language has proven to be pragmatic, efficient and accurate. The future developments will focus on improving the efficiency of the tool by fine tuning the rule logic and by minimizing the false positive and false negative rates. There will also be a focus on adding additional features such as providing the user the ability to hide problem markers if they disagree with the tool and by providing support for the C++ language. Additionally, the rest for the CERT rules for C that were identified as automatable will be implemented.

There are many static analysis tools that provide secure code analysis that are available for developers. However, none of these tools implement the CERT secure coding rules for the C programming language. This paper provides C programmers with an educational development tool that enforce secure coding standards. This tool is open source and will continue to be maintained by the Department of Computer Science at CSUS. The tool is available on the project website at GitHub (http://benw408701.github.io/SecureCodingAssistant/).

This project was conducted when Victor Melnik was a student in MS Computer Science program at California State University, Sacramento. More implementation details can be found in his Master Project Report [20], that is an extended version of this paper.

7 Acknowledgements

Acknowledgements and attributions are given to Carnegie Mellon University and its Software Engineering Institute, as this publication incorporates portions of the "SEI CERT C Coding Standard" (c) 2017 Carnegie Mellon University, with special permission from its Software Engineering Institute". Any material of Carnegie Mellon University and/or its software engineering institute contained herein is furnished on an "as-is" basis. Carnegie Mellon University makes no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material, Carnegie Mellon University does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement. This publication has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute. CERT and CERT Coordination Center are registered trademarks of Carnegie Mellon University.

References

1. 2017 Annual Data Breach Year-End Review, 2018. https://www.idtheftcenter.org/images/breach/2017Breaches/2017AnnualDataBreachYearEndReview.pdf. Retrieved on Feb 27, 2019.
2. Bearak, S., 2017. Uber Data Breach Affects 57 Million: It is Time to Own Our Identities. https://www.identityforce.com/business-blog/ubers-data-breach-affects-57-million-its-time-to-own-our-identities. Retrieved on Feb 27, 2019.

3. Cimpanu, C., 2017. SEC Says Hackers Breached Its System, Might Have Stolen Data for Insider Trading. https://www.bleepingcomputer.com/news/security/sec-says-hackers-breached-its-system-might-have-used-stolen-data-for-insider-trading/. Retrieved on Feb 27, 2019.
4. Eclipse., 2018. Extensions and Extension Points. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fconcepts%2Fextension.htm. Retrieved on Feb 27, 2019.
5. Hicken, A., 2018. ERR34-C. Detect errors when converting a string to a number. https://wiki.sei.cmu.edu. Retrieved on Feb 27, 2019.
6. Hicken, A., & Seacord, R., 2018. STR34-C. Cast characters to unsigned char before converting to larger integer sizes. https://wiki.sei.cmu.edu. Retrieved on Feb 27, 2019.
7. Leary, J., 2018. Equifax Breach Impacts 147.9 Million: Steps to Keep Your Identity Protected. https://www.identityforce.com/business-blog/equifax-breach-impacts-143-million-steps-to-keep-your-identity-protected. Retrieved on Feb 27, 2019.
8. Li, C., White, B., Dai, J., & Zhang, C., 2017. "Enhancing Secure Coding Assistant With Error Correction and Contract Programming". Proceeding of National Cyber Summit 2017, Huntsville, AL, Jun 6-8, 2017.
9. Long, F., & Hicken, A., 2018. MSC30-C. Do not use the rand() function for generating pseudorandom numbers. https://wiki.sei.cmu.edu. Retrieved on Feb 27, 2019.
10. MITRE, 2018. CWE-843: Access of Resource Using Incompatible Type ('Type Confusion'). Common Weakness Enumeration.
11. NIST, 2017. Test Suites, 4.9. NIST Samate: https://samate.nist.gov/SARD/testsuite.php. Retrieved on Feb 27, 2019.
12. Ozkan, S., 2018. Browse Vulnerabilities by Date. https://www.cvedetails.com/browse-by-date.php. Retrieved on Feb 27, 2019.
13. Pretz, K., 2014. 10 Recommendation for Avoiding Software Security Design Flaws. http://theinstitute.ieee.org/special-reports/special-reports/10-recommendations-for-avoiding-software-security-design-flaws. Retrieved on Feb 27, 2019.
14. Razmyslov, S., 2018. INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors. https://wiki.sei.cmu.edu. Retrieved on Feb 27, 2019.
15. Seacord, R., & Flynn, L., 2018. FIO32-C. Do not perform operations on devices that are only appropriate for files. https://wiki.sei.cmu.edu. Retrieved on Feb 27, 2019.
16. Seacord, R., & Hicken, A., 2018. MEM31-C. Free dynamically allocated memory when no longer needed. https://wiki.sei.cmu.edu. Retrieved on Feb 27, 2019.
17. Svoboda, D., & Snavely, W., 2017. FIO45-C. Avoid TOCTOU race conditions while accessing files. https://wiki.sei.cmu.edu. Retrieved on Feb 27, 2019.
18. White, B., Dai, J., & Zhang, C., 2016. "Secure Coding Assistant: Enforcing Secure Coding Practices Using the Eclipse Development Environment". Proceeding of National Cyber Summit 2016, Huntsville, AL, Jun 8-9, 2016.
19. Benjamin White, Jun Dai, Cui Zhang, "An Early Detection Tool in Eclipse to Enforce Secure Coding Practices". International Journal of Information Privacy, Security and Integrity (IJIPSI), Inderscience, 2018.
20. Victor Vladimirovich Melnik, "Enhancing Secure Coding Assistant: Enforcing Secure Coding Rules for C Programming Language", Master Report at California State University, Sacramento, 2018.