A Source Code Review Method for Discovering Security Flaws Efficiently

Cheng Zhou[a], YongLi[b], Weiwei Li[c], Chen Wang[d]

State Grid Smart Grid Research Institute (Nanjing), Nanjing, 210000,China

[a]e-mail: zhoucheng@sgri.sgcc.com.cn, [b]e-mail: liyong@sgri.sgcc.com.cn,

[c]e-mail: liweiwei@sgri.sgcc.com.cn, [d]e-mail: wangchen@sgri.sgcc.com.cn

Keywords:sources codes, security flaws, code review, reference tree

Abstract. Currently code review or white-box security detecting is widely used to parse the source codes and discover security vulnerabilities. In this paper we illustrate a more accurate code security review method based on the reference tree with security properties which made of all manipulable entries in source codes. This method in this paper can greatly reduce false positives and provides a better solution for automated secure reviewing on source codes by only checking the exploitable security flaws.

Introduction

Software flaws are the source of a vast majority of vulnerabilities in today's information system. Software flaws are some defect in software which may allow a third party or program to gain unauthorized access to some resource, or change the control flow to perform unintended operations. Software flaws can cause economic damage (e.g. sensitive information stolen, unavailable services), even unacceptable disaster when they occurred in critical software systems running the critical infrastructure (e.g. nuclear, biological, and chemical laboratories, electrical power grids, water treatment and distribution systems, air traffic control and transportation signaling systems) on which human lives and livelihoods depend.

Currently code security review or white-box security detect is widely used to parse the codes and discover security flaws. The problem is current code security review solutions have many shortcomings (e.g. precision problem, producing large amount of false positives and false repair identifications; scalability problem, insufficient support for complicated applications, and incapable of handling large applications; applicability problem, requiring large amount of additional manual workload; etc.) which preventing programmers or testers to adopt them.

The automated code security review techniques utilize parse tools to perform all of the code inspection for avoiding the highly labor-intensive activity of security experts which is needed in the process of manual code review. But in automated code security reviews practices; there is a trade-off between precision and scalability. Currently there are two different solutions:

1) By means of sacrificing precision, some automated code review tools utilize string match or other simple methods to inspect codes very quickly, but it will produce too many false positives and false negatives. These solutions are based on lexical detect which will discover dangerous C library functions and system calls in the code without further parse, i.e., semantic parse of RATS[1] and ITS4[2]. It is the simplest approach but will produce a large number of false positives. For example, even a local variable that has the name strcat can result in a false alarm.

2) Some other automated code review tool based on model detecting[3] which will focus more on the precision than scalability, and their program parses are highly computing-heavily and often have trouble handling large applications.

Lazy abstraction[4] present a model detecting method which continuously builds and clarifies a single abstract model on demand, driven by the model checker, so that different parts of the model may exhibit different degrees of precision for verifying the desisted property. Several other techniques like MOPS[5] also present different model detecting solutions, but the limitation of model detecting is lots of annotations or predicates need to be inserted into programs during the programming for verification, and programmers are involved. And the overhead of program analysis is high and the capability of handling large applications is insufficient.

© 2015. The authors - Published by Atlantis Press

Overview

In principle, security flaws are introduced into software since most software (their functions) cannot keep operating correctly under unanticipated conditions. Therefore, attackers will exploit software flaws by

1) Making up the inputs of the software;
2) Compromising the interface of the software to other application-level entities;
3) Assaulting the execution environment of the software.

Although a security flaw means specific part of software which has security risk and may cause a deviation of secure behaviors, there are two types of security flaws, exploitable or un-exploitable according to whether attackers can make use of the flaw. Since there is not a data flow or control flow path existed between inputs, interfaces, or other environment entries with un-exploitable security flaws, it is impossible that an attacker can exploit them and cause any damage. By avoid the unnecessary overhead for parsing un-exploitable security flaws, this invention can obtain great balance between precision and scalability by only focusing on exploitable ones which can be manipulated via inputs, interfaces, or other environment entries by attackers. For example, one application may use a function with buffer over-flow risk, but if its parameters cannot be manipulated, which means no one can exploit this security flaw to do malicious operations.

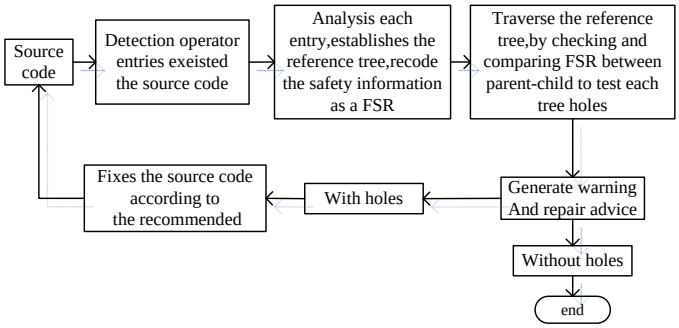

Figure 1. illustrates the process of discovering flaws in codes.

The first step in this process is to discover/locate all the manipulable entries in codes. These manipulable entries include while not limited, input entry for users, network input entries, I/O input entries, etc. One possible, while not limited way to discover them is to locate all the related APIs which will be used to obtain outside inputs, interact with interfaces and other environment entries.

The second step is to build a reference tree for each manipulable entry located in step 1. Every reference or implied reference which can traverse back to this entry should be included in the tree. And it is necessary to record or keep security properties in each node of the tree for further processing. The security property is defined as Formulated Security Restrictions (FSR). FSR can define various security properties, and is machine recognizable for automatically processing. FSR may include while not limited, regular expressions, ACLs, etc.

The third step is the tree traversal for discovering security flaws by utilizing security properties to detecting whether there is exploitation existed. The security flaw detecting method may be, while not limited, detecting the conflicts between Father's FSR and its Child's FSR, etc.

The forth step is to generate warnings according to flaws discovered in step 3, and provides repair recommends based on the parse of step 3.

At last users (developers) can revise their codes based on recommends and perform this process again to guarantee there is no new flaw introduced during the revision.

Procedure

Figure 2 illustrates the system architecture of this method. Four modules are included, Manipulable Entry Locator, Parse Engine, Holes Detector and Warning & Repair Generator.

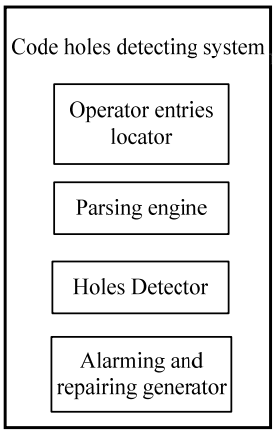

Figure 2. Source Code Hole Detecting System Architecture

The Manipulable Entry Locator can be implemented, while not limited, by searching all the relevant APIs for receiving outside inputs such as scanf(), Getchar(), or GetWindowsText, socket inputs, etc.(different program languages may have different APIs).

The Parse Engine can be implemented, while not limited, by utilizing compiler technologies. This engine can find all the relevant part which can traverse back to manipulable entries. And this engine can automatically assign some FSR during the syntax parsing. Finally, this Engine will produce code reference Trees which illustrated in Figure 3.

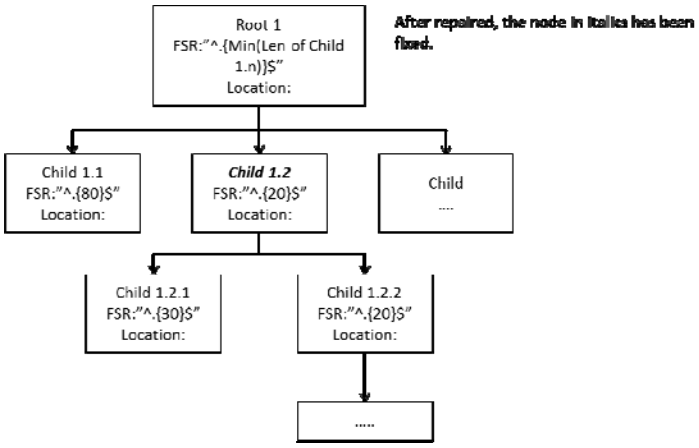

Figure 3. Source Code Hole Reference Tree

The Hole Detector will subsequently process the code reference trees. It will detect whether there is any security flaw in reference trees by parsing the FSR of points in reference trees.

The Warning & Repair Generator will provide security hole warning information and repair recommends for security flaws discovered by Hole Detector.

A. Scenario 1 - Code Injection Hole Disclosure

Code Injection is a term used when code is injected straight into a program/script from an outside source for execution at some point in time. Usually executable logic is inserted in non-executable text strings submitted to the application, e.g. SQL injection and Cross-site scripting (XSS or CSS). SQL Injection refers to the technique of inserting SQL meta-characters and commands into Web-based input fields in order to manipulate the execution of the back-end SQL queries. These are attacks directed primarily against another organization's Web server. Cross Site Scripting attacks work by embedding script tags in URLs and enticing unsuspecting users to click on them, ensuring that the malicious JavaScript gets executed on the victim's machine.

Because code injection is caused by embedding executable contents in the text strings which should be non-executable data. There are two ways to define the FSR, one is to define the property of string as no-executable, and another way is to indicate there should not include any executable “key word” or expression in the string (e.g. script tags such as <script>alert(“OK”)</script>, or SQL

meta-characters such as such as the single-quote (‘) or the double-dash (—)). Here we choose the second method, so we can build the FSR section as [FSR: restrictions].

Content restrictions can be expressed in regular expressions. For example, “^.$” can express all possible strings. This may represent original inputs from outside. When transferring outside inputs which regarded as data to construct SQL queries, the content restriction is not containing SQL specific meta-characters, and can be formulated as follows:

{FSR:

^(?!(/(\w*((\%27)|('))((\%6F)|o|(\%72)|r|(\%52))/tx).)*$}

$$\eqalignno{& \{{\rm FSR}:\cr& \wedge(?!(/ (\backslash {\rm w}^{\ast}((\backslash \hbox{\%}27)\vert (\backslash \hbox{'}))((\backslash \hbox{\%}6{\rm F})\vert{\rm o} \vert (\backslash \hbox{\%}4{\rm F}))((\backslash \hbox{\%}72)\vert {\rm r}\vert (\backslash \hbox{\%}52))/{\rm ix}).)\cr& {^{\ast}} \$\}}$$

- /w *denotes zero or more alphanumeric or underscore characters
- (\%27)|\denotes the ubiquitous single-quote or its hex equivalent
- (\|%6F)|o|(%4F))((\%72)|r|(\%52)denotes the word ‘or’ with various combinations of its upper and lower case hex equivalents.

•Ix denotes open IgnoreCase and IgnorePattern WhiteSpace options

When transferring outside inputs which regarded as data to construct URLs, the content restriction is not embedding script tags in URLs, and can be formulated as follows:

{FSR:

content^(?!(/((\%3C)|<)((\%2F)|v)*[a-z0-9\%]+((\%3E)|>)/ix).).*$}

$$\eqalignno{& \quad \{{\rm FSR}:\qquad {\rm content}\qquad \wedge(?!(/((\backslash \hbox{\%}3{\rm C})\vert <)((\backslash \hbox{\%}2{\rm F})\vert \backslash /)^{\ast}[{\rm a}-{\rm z}0-\cr& 9\backslash \hbox{\%}]+((\backslash \hbox{\%} 3{\rm E})\vert>)/{\rm ix}).). {^{\ast}}\$\}}$$

- ((\%3C)| <)denotes the detecting for opening angle bracket or hex equivalent
- ((\%2F)| v) *denotes the forward slash for a closing tag or its hex equivalent
- [a – z0 – 9\%] +denotes the detecting for alphanumeric string inside the tag, or hex representation of these
- ((\%3E)| >)denotes the detecting for closing angle bracket or hex equivalent

•Ix denotes open IgnoreCase and IgnorePattern WhiteSpace options

Figure 4 illustrated one reference tree which built by Parse Engine. All points will be numbered as n.n to indicate the relation of father and child points. The FSR and position info will also be recorded in the point. The security parse module will process this reference tree by detecting whether every child point's FSR consistent with its father's one. One solution is to traverse the tree, search from the leaf points, and compare their FSR with their father's. For example, in Figure 4, Child 1.2.2 is one child of Child 1.2, and its FSR “^(?!(/((\%3C)|<)((\%2F)|v)*[a-z0-9\%]+((\%3E)|>)/ix).).*$” which means should not contain any SQL spcific meta-characters, but the FSR of its father (Child 1.2) is “.^.$” which means it may contain all possible characters include SQL specific meta-characters. So here we find a security flaw, and Warning & Repair Generator will generate a warning and provide repair recommends such as add string validation on Child 1.2.

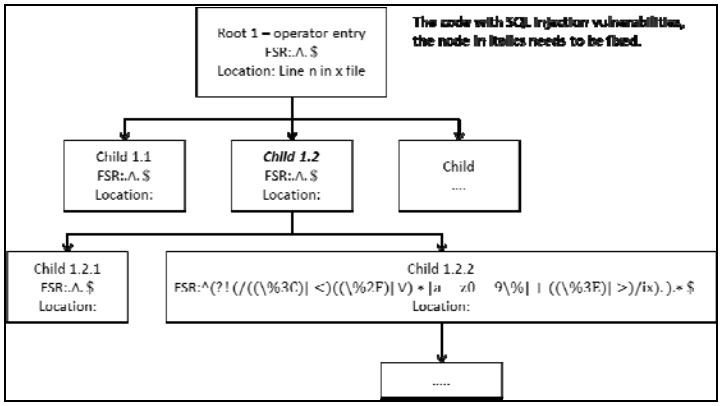

Figure 4. Source Code Hole Reference Tree - SQL Injection

B. Scenario 2 - Buffer-overflow Hole Disclosure

Generally speaking, a buffer overflow occurs when a piece of data B is written to a buffer A such that the size of B is greater than the legally allocated size of A; Another form of buffer overflow is integer overflow, or integer wrapping, which is a potential problem in a program based upon the fact that the value that can be held in a numeric data type is limited by the data type's size in bytes. We can formulate the FSR section as{FSR: ."^{length}$"}.

Figure 5 illustrates one reference tree which built by Parse Engine. All points will be numbered as n.n to record the relation of father and child points. The FSR and position info will also be recorded in the point. The security parse module will process this reference tree by detecting whether every child point's FSR conforms to its father's. One solution is to post-order traversal the tree, search from the leaf points, and compare their FSR with their father's.

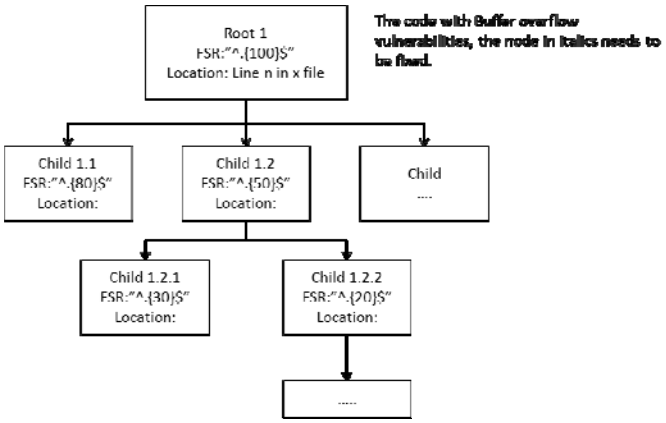

Figure 5. Source Code Hole Reference Tree - Buffer Over-flow

For example, in Figure 5, Child 1.2 has two children, Child 1.2.1 and Child 1.2.2, and their allowable max buffer length is 30 and 20. Because the allowable length of their father (Child 1.2) is 50, so overflow may occur when assign father's data to child. Analysis module will generate a warning and recommend repairing allowable length of father point (Child 1.2) with 20, illustrated in Figure 6.

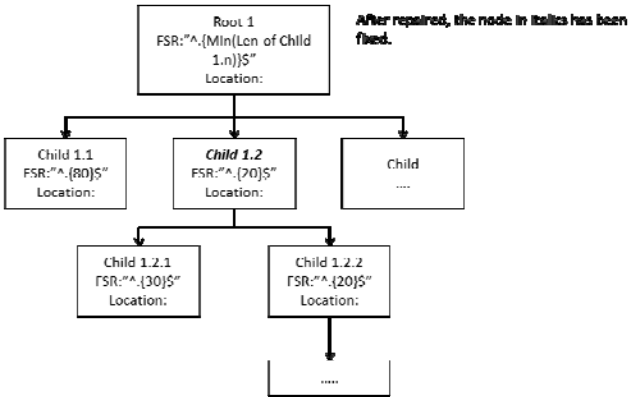

Figure 6. Built Tree from Revised Source Codes

C. Scenario 3 - Privilege Elevation Hole Disclosure

Elevation of privilege allows an attacker or codes to achieve a higher level of privilege than he shouldn't have. The privilege here may not only represents the privilege of users (such as administrator, power user or guests), but also represents the privilege of codes (such as kernel or user mode). So we may formulate this kind of holes with Access Control List (ACLs).

D. Scenario 4 - Information Disclosure or Data Tampering Hole Disclosure

Modification of data within the system to achieve a malicious goal or exposure of protected data to a user who is not authorized to access that data. Their FSR can be also described in ACL.

Related Work

Previous work related to code review has focusing on buffer overflow holes. D. Wagner, J. Foster, E. Brewer, and A. Aiken, provide a method for finding potential buffer overflow holes in security-critical C code[6], this method formulate the detection of buffer overflow as an integer constraint problem, use some simple graphtheoretic techniques to construct an efficient algorithm for solving the integer constraints, and finally security knowledge is used to formulate heuristics that capture the class of security-relevant bugs that tend to occur in real programs.

Vinod Ganapathy, Somesh Jha, David Chandler, David Melski and David Vitek, provide a method modeling C string manipulations as a linear program and using linear programming literature to determine buffer bounds for understanding and eliminating buffer overflow bugs from code.

Conclusion

The method described in this paper can greatly reduce false positives and provides an efficient solution for automated secure reviewing on codes by only detecting the exploitable security flaws.

This method can also reduce the parse overhead of codes reviewing, especially for large applications, by avoiding un-exploitable security flaws which cannot be manipulated by attacker via inputs, interfaces or other environment entries. Based on the FSR information on the reference tree, this method can also provide precise recommends for revising vulnerable codes.

Acknowledgements

We are grateful to a number of readers whose comments have substantially improved the paper. This work is supported by the Science Projects of State Grid Corporation of China under Grant EPRIXXKJ[2014]2242.

References

[1] Secure Software (Fortify acquired)."RATS-Rough Auditing Tool for Security". http://www.fortify.com/security-resources/rats.jsp, 2012.

[2] J. Viega, J. T. Bloch, T. Kohno, and G, McGraw. “ITS4: A static vulnerability scanner for C and C++ code”. ACM Transactions on Information and System Security, 5(2), 2002.

[3] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski and David Vitek, “Buffer Overrun Detection using Linear Programming and Static Analysis”, Proceeding of the 10th ACM conference on Computer and communications buffer overflow (CCD ’03), pages 345-354, Washington D.C., USA, 2003.

[4] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy Abstraction”, In Proc. POPL, pages 58-70. ACM, 2002.

[5] Ha Chen, David Wagner, MOPS: an infrastructure for examining security properties of software, Proceeding CCS ’02 Proceedings of the 9th ACM conference on Computer and communications security ACM New York, NY, USA ©2002.

[6] D. Wagner, J. Foster, E. Brewer, and A. Aiken, “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities”, Network and Distributed System Buffer overflow Symposium, San Diego, CA, February 2000.