International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Integrated Approach to Detect Vulnerabilities in Source Code

Monica Catherine S[1], Geogen George[2]

[1]Information Security and Cyber Forensics, SRM University, India
[2]Cyber Security Research Centre, SRM University, India

Abstract: ***Nowadays, security breaches are greatly increasing in number. This is one of the major threats that are being faced by most organisations which usually lead to a massive loss. The major cause for these breaches could potentially be the vulnerabilities in software products. Though there are many standard secure coding standards like CERT (Computer Emergency Response Team), software developers fail to utilize them and this leads to an unsecured end product. The difficulty in manual analysis of vulnerabilities in source code is what leads to the evolution of automated analysis tools. Static and dynamic analyses are the two complementary methods used to detect vulnerabilities in source code. Static analysis scans the source code without executing it but dynamic analysis tests the code by executing it. Each has its own unique pros and cons. The proposed approach helps the developers to correct the vulnerabilities in their code by an integrated approach of static and dynamic analysis for C and C++. This eliminates the pros and cons of the existing practices and helps developers in the most efficient way. It deals with common buffer overflow vulnerabilities, format string vulnerabilities and improper input validation. The whole scenario is implemented as a web application.***

Keywords: Secure coding, Static analysis, Dynamic analysis, Buffer overflow

1. Introduction

Cyber security attacks have increased exponentially in the last few years. In 2013, it was found that the number of security breaches rose up to 62% from 2012[1]. Buffer overflow errors are the leading threat in most cases and 21% of the Common Weakness Enumeration (CWE) threat categories cause it [2]. A report by USA today stated that 43% of companies in the US experienced a data breach in the year 2013[3]. The major cause for some of these breaches could potentially have been the vulnerabilities in software products.

There are many causes for vulnerabilities in software products, but the most common ones are flaws introduced by developers during program construction. Though there are many standard secure coding practices that are to be followed during code construction, software developers fail to follow them and thus this leads to major threats in the end product.

Therefore, there is a great demand for detecting vulnerabilities in software product. But detection of code during development phase itself will reduce the time, risk and cost of correcting it. So, careful analysis of code during development phase is a necessary action but it is more difficult to do manually.

Static and dynamic analyses are the two complementary methods that are used to detect vulnerable codes. Static analysis just scans the source code to check for the flaw and this eliminates the need of executing it. On the other hand, dynamic analysis tests the code by executing it along with the test cases. Many tools are available for the above said methods in the market but both have their own pros and cons.

Static analysis is fast and simple to use. It scans the code line by line and detects the flaws that can lead to vulnerability in the software. Since it scans the source code it can be used easily while constructing the code and the cost of fixing it will be low when it is detected earlier. On the negative side, it generates many false positives and false negatives. For example, Splint by Larochelle and Evans, is a lightweight static analyzer which generates a number of false positives and negatives.

On the other hand, in dynamic analysis false positives and negatives are reduced but it increases the duration of analysis. Also, it may miss some flaws without detecting them because some execution path might not have been executed during testing.

Since the two approaches have positive as well as negative aspects, an integrated approach which employs both of the ideas effectively is a better option. And that should adopt the strengths of the two and eliminates their weaknesses. In the proposed system, static and dynamic analysis is combined. First, source code is subjected to static analysis and the result is stored. Secondly, it is subjected to dynamic analysis where the code is executed along with the test cases and vulnerabilities are detected during runtime. Finally, warnings are displayed to the user with alternative solutions. It mainly focuses on common buffer overflow vulnerabilities, format string vulnerabilities and improper input validation. Also, it is implemented as a web application which extends the benefits by making it platform independent.

The paper comprises of five sections. Literature review is described in section II, design and methodology in section III, Results and findings in section IV, conclusion and future work in section V.

2. Literature Review

Piromsopa et al [5] defines buffer overflow and proves the necessary condition for preventing the buffer overflow attack. Here, buffer overflow is described as the condition where the data transferred to a particular buffer exceeds

the storage capacity of the buffer and some of the data to be copied overflows into the succeeding buffer, where the data was not supposed to go into. It therefore proves that the preservation of the integrity of addresses across domains as a necessary condition for preventing buffer-overflow attacks. Erik et al [6] talks about improper input validation and current best practices to minimize it. Special characters should not be allowed in inputs, character encoding should be done for dynamic inputs, inputs should be sanitized such that malicious characters should be removed before sending it to the database are some of the proper input validation steps.

Nishiyama et al [7] proposed a tool called "SecureC" that is aimed at protecting applications from general buffer overflow attacks. Here, it translates the given source code into a security enhanced code such that it avoids buffer overflow. One of the methods it uses is "shadow stack" where the dynamic memory is allocated externally to the normal stack frame and last page of the shadow stack is made read only. It is done by linking the source code with SecureC runtime library. Fig.1 is an example output of this tool; here buffer overflow will lead to segmentation fault.

```
1: void foo() {
2: char buf[10];
3: gets(buf);
4: }
```

```
1: void foo() {
2: struct SHADOW {
3: char buf[10];
4: } *_shadow_
5: =ALLOC(sizeof(SHADOW));
6: gets(_shadow_->buf);
7: FREE(_shadow_);
8: }
```

Figure 1: Source Code and Translated Code

Aishwarya et al [8] introduced a tool which is used to locate vulnerable files which are known to have been the root cause for buffer overflow in the application. The tool has two stages; firstly, a record of the stack trace is made through the entire normal execution of the application. In the second stage**,** a record is made of the stack trace with an injection of the buffer overflow attack through the entire execution of the application.

Finally, a comparison of the two traces is made and then a result is determined. If the stack traces are found to be similar, this means an attack will not be successful. However, if they are not similar, an attack may be successful.

Chuang et al [9] proposed a method for bounds checking which helps to increase the efficiency. It basically checks the memory locations that are prone to buffer overflow attacks and then the rest can be safely pruned away.

Kendra et al [10] tested the basic capabilities of some static and dynamic tools which detects buffer overflow. Using twenty two attributes they have created many test cases for testing the tools and have calculated detection rate, false alarm rate and execution time of the tool.

3. Design and Methodology

The proposed system is a web application which helps the developers to detect vulnerable code during the development phase. This is an integrated approach involving static and dynamic analysis which gets the file as input and then analyses the code and also generates warnings. The proposed system mainly focuses on buffer overflow vulnerabilities, format string vulnerabilities and improper input validation. HTML5 was used to create the webpage along with python CGI scripts. Code evaluation was also done in python language using python 2.7. Apache2 and MySQL were the servers and database used. Currently, it is designed for C and C++, but it will later be enhanced for other languages as well.

The web page is designed in such a way that the user can upload the source code as a file (either in .c/.cpp based on the language chosen or .txt format). The application also has a different tab for each of the different languages. The uploaded file is stored in a predefined location in the server. Later it is converted into a .c or .cpp file based on the language chosen.

The goal of this application is to detect all vulnerabilities in the code by an integrated approach consisting of static and dynamic analysis and which combines the positives of both. The overall process is given in Fig.2. It comprises of two modules:

- Static Analysis
- Dynamic Analysis

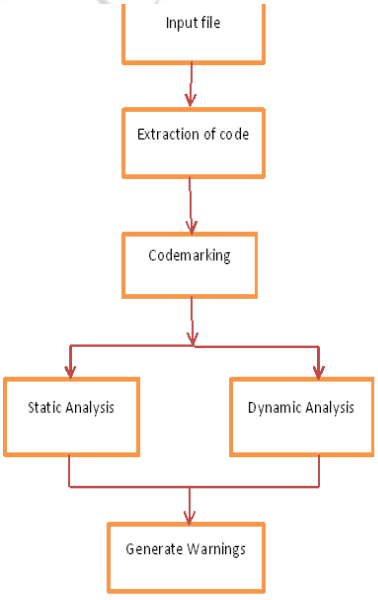

Figure 2: Overall flow of the process

A. Static Analysis

This module works on source code and tests the code without executing it. Initially, built-in functions that may lead to vulnerabilities are stored in a table of MySQL database. After the user uploads the file via the webpage, the file is forwarded to static analyzer for analysis. Here, it focuses on two steps before checking for vulnerable code.

First, all the unsafe functions are marked with line numbers by comparing them with already stored functions. Secondly, bound values for all variables used in the program are stored in a table. These steps can reduce the time and increase the efficiency of analysis because only the marked lines will be checked for vulnerabilities, which means it reduces the time taken compared to testing each and every line of the program. A separate module is written for each type of vulnerability. For example 'strcpy()' and 'strcat()' are vulnerable because they may lead to buffer overflow but they can be used in a secure way also. So in order to check whether the particular function used is vulnerable or not, a separatemodule is written. An example of this is Fig.3.

```
Int main(arg,argv[])
{
Char buffer[4];
If(arg>0)
Strncpy(buffer,argv[0],5);
}
```

Figure 3: C code with vulnerable function

In Fig.3, initially, 'strncpy' is marked vulnerable and then bound values of buffer are stored. While using 'strncpy' the third parameter value should be the size of the destination buffer minus one. At this point, the buffer is getting overflowed since the buffer size is less than third parameter value. A separate module is written to check buffer overflow condition and the result is returned to the main function. Thus the application throws a warning to correct it.

B. Dynamic analysis

This module involves the actual running of the code. The .c or .cpp file that is stored in server is forwarded for dynamic analysis. This file will be linked with the library that is written in C language which manages the dynamic memory allocation. Then the linked file is compiled with "gcc" and "g++" compiler for .c and .cpp respectively. The compiled code is executed and vulnerabilities are detected during runtime. Since vulnerable code is allowed to run, it is executed in a sandboxed environment using Sandboxie[11] an open source sandbox and thus it is isolated from the memory of the host system. A prerequisite of this module is the test case table. All possible test cases are stored in the database initially. Here, test cases are the randomized input that is to be given when the code runs and it is based on the data type of the variable that accepts it. During the execution of the code these stored test cases are applied and the buffer overflow condition is checked. Also, check value is inserted after memory allocated for each buffer and therefore buffer overflow is detected when this check value is changed. The goal of this module is to detect the vulnerabilities that are not covered by the static analyzer and it mainly focuses on buffer overflow vulnerability. An example is shown in Fig 4.

```
1. Int siz;
2. char str1[10],str2[]="welcome India";
3. char *ptr,name[15];
4. strcpy(str1,str2);
5. scanf("%d",&siz);
6. ptr=(char *)malloc(siz);
7. strcpy(ptr,str2);
```

Figure 4: Example for buffer overflow

In line 4 of Fig.4 the vulnerability can be analyzed by a static analyzer but in line 7 it fails to analyze because the memory is dynamically allocated. At this point the dynamic analyzer can help to detect it.

4. Results and Findings

The goal of this approach is to detect vulnerable code at the time of development. It takes a file as input and processes it. The web application acquired the file and then the file was sent for static and dynamic analysis.

In static analysis, the code was scanned by our analyzer and all variables, its type and allocated memory size were stored in database. The built-in functions (for example: strcpy()) were also stored with line numbers. Later, those marked line numbers were scanned for insecure functions. For each vulnerable built-in function a separate module was written to check for vulnerability. With Fig.4 as an example, an explanation for how strcpy() function was handled in the application is given.

When the code was analyzed statically, all variables declared including str1and str2 were stored in a table in a MySQL database along with their data type and allocated memory size. After that, the built in functions 'strcpy' in line 4 and 7 of Fig.4 were marked. Since both are insecure according to secure coding standards, they were marked as insecure and forwarded for evaluation. In line 4, the static analyzer detected the vulnerability because the size of str1 was less than the size of str2 and all the detected vulnerabilities were stored in a file. But in line 7 it was not evaluated as vulnerable since memory wasn't allocated statically. This was handled later in dynamic analysis. In dynamic analysis, the code was linked with the library that is created in C and where this buffer overflow condition is checked. When "malloc" was called a function in the library linked was invoked. A check value was inserted at the upper boundary of the allocated memory size for the pointer "ptr". When the code prompted for a user input for "siz" randomized inputs were supplied based on the data type and buffer overflow was checked when the control reached the end of the code. It was detected that for user inputs less than '16', buffer overflow can occur. The detected vulnerabilities were stored in a file. The stored vulnerabilities by both the analyzers were displayed to the

user as a response. Fig.5 explains how it responds to format string vulnerability.

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

Figure 5: Format String vulnerability

The number of format specifiers did not match the variables given, hence it threw a warning. Static analysis was simple and fast but it generated false positives and false negatives also. For example, when a buffer was allocated dynamically, the buffer size could not be predicted in static analysis hence it did not throw warnings though there was a chance for that buffer to be overflowed. Since the proposed method is an integrated approach of static and dynamic analysis, false positives and negatives are reduced and most of the flaws are covered. However, it cannot detect 100% of the flaws. Current implementation is in C and C++ but the same method with slight changes can be used for other languages like C#, Python etc.

5. Conclusion and Future Work

The integrated approach of static and dynamic analysis is more efficient and detects most of the vulnerable code in the program and thus helps the developer to correct the vulnerabilities in code which leads to a flawless secure end product. Therefore this reduces the security incidents that happen because of vulnerable source code. Currently, the proposed system has focused only on the buffer overflow vulnerability, format string vulnerability and improper input validation in C and C++, still the idea of integrating the two analysis approaches can be used for detecting vulnerabilities in Python, C#, java, etc.

References

[1] Symantec Corporation Internet Security Threat Report 2014 :: Volume 19
[2] Cisco 2014 Annual Security Report
[3] http://www.usatoday.com/story/tech/2014/09/24/data-breach-companies-60/16106197/
[4] http://cwe.mitre.org/top25/.
[5] Piromsopa, Krerk, and Richard J. Enbody. "Buffer-overflow protection: the theory." Electro/information Technology, 2006 IEEE International Conference on. IEEE, 2006.
[6] http://www.sans.org/reading-room/whitepapers/application/web-application-injection-vulnerabilities-web-app-039-s-security-nemesis-34247
[7] Nishiyama, Hiroyasu. "SecureC: Control-flow protections against general buffer overflow attack." Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International. Vol. 1. IEEE, 2005.
[8] Iyer, Aishwarya, and Lorie M. Liebrock. "Vulnerability scanning for buffer overflow." Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on. Vol. 2. IEEE, 2004.
[9] Chuang, Weihaw, et al. "Bounds checking with taint-based analysis." High Performance Embedded Architectures and Compilers. Springer Berlin Heidelberg, 2007. 71-86.
[10] Kratkiewicz, Kendra, and Richard Lippmann. "A taxonomy of buffer overflows for evaluating static and dynamic software testing tools." Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics. Vol. 500. 2006
[11] http://www.sandboxie.com/index.php?HowItWorks