Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 4, April 2014, pg.1251 – 1258

RESEARCH ARTICLE

Integrating Static Analysis Tools for Improving Operating System Security

Ashish Joshi[1], Kanak Tewari[2], Vivek Kumar[3], Dibyahash Bordoloi[4]

[1]Assistant Professor, Department of CS, THDC-IHET Tehri- Uttarakhand

[2,4]Department of CS/IT, Graphic Era University, Dehradun, India – 248002

[3]Assistant Professor, Department of CS, THDC-IHET Tehri- Uttarakhand

Emails: [1] a.joshiuki@hotmail.com, [2] kanak.tewari@gmail.com, [4] dibyahash@gmail.com

Abstract: Static analysis approach is widely used for detecting vulnerabilities within the code before the execution. C/C++ programming languages consist of highest number of vulnerabilities of which buffer overflow is the highest rated. Of all static analysis tools available none has enabled to detect all the vulnerabilities. Hence, we have proposed an integrated approach using two open-source static analysis tools: Flawfinder and Cppcheck for developing a new static analysis tool.

Keywords: operating system security, static analysis, malicious code, vulnerability, buffer overflow

I. INTRODUCTION

Operating system security has always been a challenge for the users, developers and mainly administrator's. With the advent of new technologies, it has become even harder to secure systems from the malicious applications as the attackers too have advanced enough to trace the vulnerability within the operating system and exploit it. Out of all, mostly the software installed within the system causes the system crashing. Attackers inject their malicious code so nicely within the software that even the antivirus toolkit installed, fails to detect them.

It is known that buffer overflow [1] is the mostly found a vulnerability, others include format string, command line injection, SQL injection etc. Mainly in the C/C++ [2] programming languages and so many critical applications are developed using them. Applications developed in C/C++ languages are generally vulnerable to exploits due to limitation in these languages that allow an attacker to exploit vulnerability. For example, a buffer overflow is caused whenever a buffer is filled out of its limits, and it is the mostly found vulnerability [3] in C/C++ languages. Therefore, it becomes easier for a malicious user to inject code that introduces changes within the system behaviour.

Open-source codes in C/C++ are most prone to such exploitation. Illustrating, the gets function in C receives text from the input and places the first character as per the location specified and upcoming data consecutively in memory. Thus, reading continues till newline or EOF character is reached, the point at which the buffer is terminated using a null character. Thus, the programmer can't specify the size of the buffer i.e. passed to gets. As a result, when the buffer gets n bytes; the attacker attempting to write n + m bytes, will always succeed for if the data is excluding newlines.

Various kinds of techniques and tools are available in the market to detect the vulnerabilities in the code. Even there is a classification among the methods. Mainly there are two categories; firstly it's the static analysis [4], which is performed on the code without executing it i.e. during compile time. Our theme is concerned with this technique. Another is Dynamic analysis [5] which concerns about dynamically detecting the program i.e. during execution. Both techniques have their pros and cons. Although various tools are available, none of them is completely able to detect vulnerability presented. In our paper, we are using two static code analysis tools, i.e. Flawfinder and Cppcheck whose aim is to detect potential security vulnerabilities within the code available. Firstly, we have evaluated these tools and then proposed an integrated approach using them for the development of a new tool which is far more powerful and detects more vulnerability in their comparison.

II. LITERATURE SURVEY

Much work has been done to create methodologies that succeed in detecting vulnerabilities in the code. Researchers have given a lot effort in developing static analysis tools to detect vulnerabilities within the code. There are number of tools available in the market which can be used by the developer while testing their software code. For example: BOON, Cqual, Csur, SPLINT (successor of LINT) etc are present there. [6] Some of them are open-source and some are commercial. However, initially grep was used through the command line whose primary goal was to identify locations at which the program might fall prey to the common security problems. But grep has its limitations too; like it was inflexible, as it for handling it, large amount of expert knowledge was required to detect vulnerable calls within the code. So, later on techniques like Static code analysis and dynamic analysis were developed who aimed at retrieving defects within the code before and after its execution.

However, there has always been a competition between the static [7] and dynamic analysis [8] judging which is better as one of them aims at detecting vulnerabilities before executing the software or code and another while executing it. Hence, various researchers aimed at developing tools combining both the approaches and were successful too [9].Although, it is known that some of the weak variables in the program can only be detected during the run-time but mostly the lacks are detected best during the static analysis as the attacker most of the time attempts to insert his/her malicious code [10] in the source code. Also, in the dynamic analysis number of the test cases need to be applied to find the exact vulnerability. Thus, the need to develop such methodologies is growing that can secure the code from the attacker.

III. EVALUATION OF TOOLS

In our report, we have used Flawfinder and Cppcheck static code analyzers which are firstly evaluated as per their working. We have performed preliminary tests on the performance of Flawfinder and Cppcheck. We measured performance on an Intel, corei5 system with 3GB of RAM running Ubuntu 12.04.

3.1. OVERVIEW: FLAWFINDER

Flawfinder is used to detect the potential security flaws i.e. "Hits" in the source code. It is written in python so updates can be performed. It analyzes only C/C++ code. It produces a list of ''hits'' that are potential security flaws which are sorted by risk levels. It displays risk level inside the square brackets which varies from 0, i.e. very little risk, to 5 i.e. greatest risk. The level of risk not only depends on the function, but too on the values of the parameters of the function. For example, the constant strings are more often less risky than fully variable strings in many contexts, and in such contexts the hits risk level will be more modest. Flawfinder knows about the gettext i.e. a common library for the internationalized programs.

It uses an internal database called as ‘‘rule set’’. It identifies functions which are the main reasons of security flaws. This rule set comprises a large number of different potential problems that includes both general issues which can impact any C/C++ program, also specific UNIX-like and Windows functions which are problematic. We have tested various open-source projects with Flawfinder and have analyzed the whole code without excluding any part of the code. The summarized results presented in terms of hits, PSLOC (Physical source line of code) and Time.

Sample code like:

```
…….
int main() {
  char buffer[1];
  int var = 0;

  scan("%s", &buffer);
  printf("var = 0x%x\n", var);
  return 0;

}

…….
```

The above code will result in buffer overflow vulnerability.

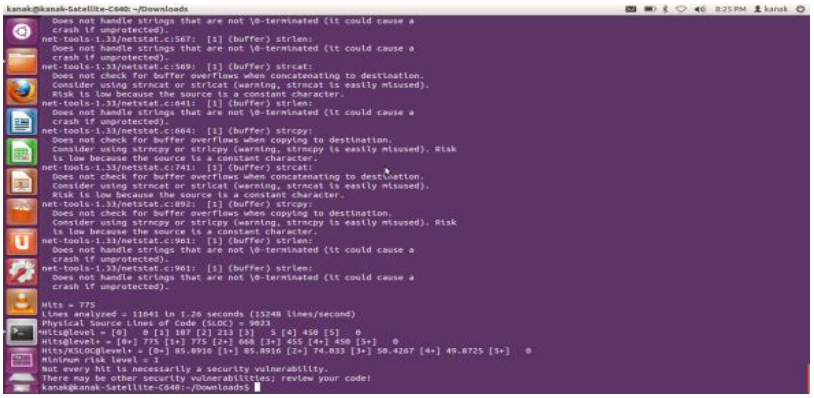

Fig 1. Screenshot for net-tools-1.33 with Flawfinder

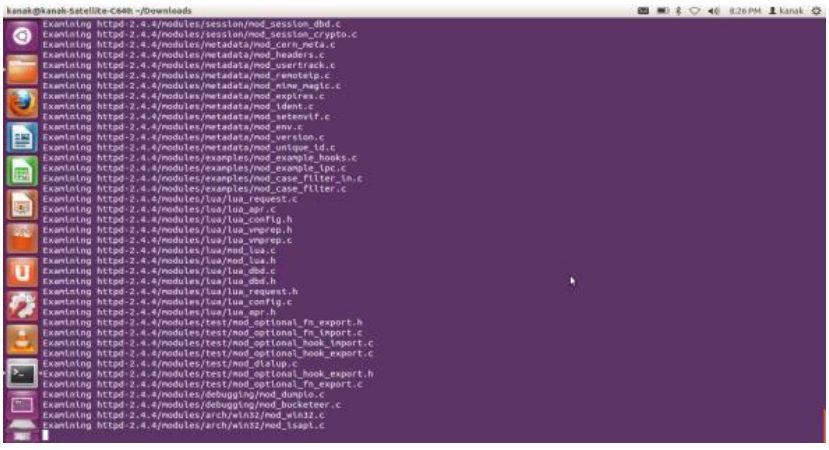

Fig 2.Flawfinder analyzing httpd-2.4.4 directory

Table 1. Code compiled for open-source packages using Flawfinder

Package	Hits (Total)	PSLOC	Time Taken (Sec)
Net-tools-1.33	775	11641	0.93
Httpd-2.4.4	1749	150011	4.82
Ssh-3.2.5	2018	161395	4.65
Wu-ftpd-2.4.4	512	9749	0.78

Table 2.Vulnerability Types and respective function causing vulnerability in various levels in Flawfinder

Levels / Vulnerability Types & Functions	[1]	[2]	[3]	[4]	[5]
Buffer	Strncpy,strcat, strcpy,Strlen, sscanf,getc, read	Char,sprintf, strcpy,Strcat, memcpy, TCHAR, open,bcopy	Getopt_long, Getenv,getwd, getopt,realpath	Strcpy,sscanf, Sprint,strcat, Fscanf,_tcscat,	gets
Format	-	-	-	Fprintf,printf, Sprintf,vfprintf, snprintf,vsnprintf, syslog,_sntprintf	-
Misc	-	Fopen,open	Chroot, InitializeCriticalSection, EnterCriticalSection, LoadLibrary	-	-
Shell	-	-	-	Execvp,execv, Popen,execlp, Winexec,execl, shellExecute	-
Race	-	vfork	-	access	Chown,chmod readlink
Port	snprintf	-	-	-	-
Access	unmask	-	-	-	-
Integer	-	atoi,atol	-	-	-
Tempfile	-	tempfile	-	-	-
Crypto	-	-	-	crypt	-
Random	-	-	Srand,srandom,random	-	-

Table 3.Hits in each level of Flawfinder within the package

Hits / Packages	[1]	[2]	[3]	[4]	[5]
Ssh-3.2.5	90	932	69	104	12
Net-tools-1.33	107	213	5	450	0
Wu-ftpd-2.4.2-beta-18	141	206	35	129	1
Httpd-2.4.4	719	898	27	99	6

3.2. OVERVIEW: CPP CHECK

Cppcheck is a static code analysis tool for C/C++ code. Like other C/C++ compilers and analysis tools, it does not detect syntax errors. It mainly detects the types of bugs that the compilers mostly fail to detect. The main goal is no false positives unlike Flawfinder where we often received a false positive. By default, Cppcheck checks all pre-processor configurations. One of the key features of Cppcheck is to look for memory leaks and resource leaks. It can detect many common misunderstandings which are by default in the code. Although Cppcheck understands the standard allocation and deallocation functions but it does not know what library functions actually do.

Sample code like:

```
……

int main()

{

char a[10];



a[10] = 0;

return 0;

}

…….
```

It will result in resource leak under Cppcheck testing. Cppcheck comes with command line options or severities like:

Error: used when bugs are found

Warning: provide suggestions about defensive programming to prevent bugs

Style: stylistic issues related to code cleanup (unused functions, redundant code, constness)

Performance: Suggestions for making the code faster. These suggestions are only based on common knowledge. It is not certain you'll get any measurable difference in speed by fixing these messages.

Portability: Portability warnings, 64-bit portability code might work different on different compilers.

Information: Informational messages about checking problems.

The results for testing Cppcheck on open-source packages are as:

Table 4. Code compiled for open-source packages using Cppcheck

Packages	Files	Time
Net-tools-1.33	45	8sec
Httpd-2.4.4	254	2m21sec
Ssh-3.2.5	328	2m38sec
Wu-ftpd-2.4.4	32	11sec

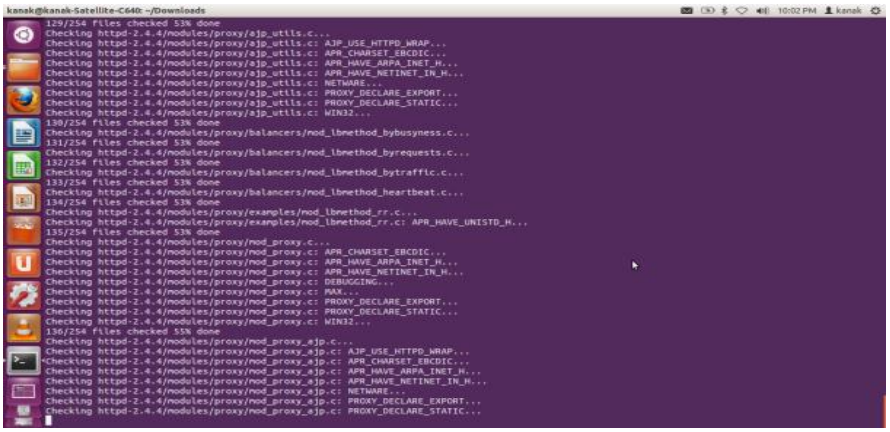

Fig3. Screenshot for httpd-2.4.4 done with Cppcheck

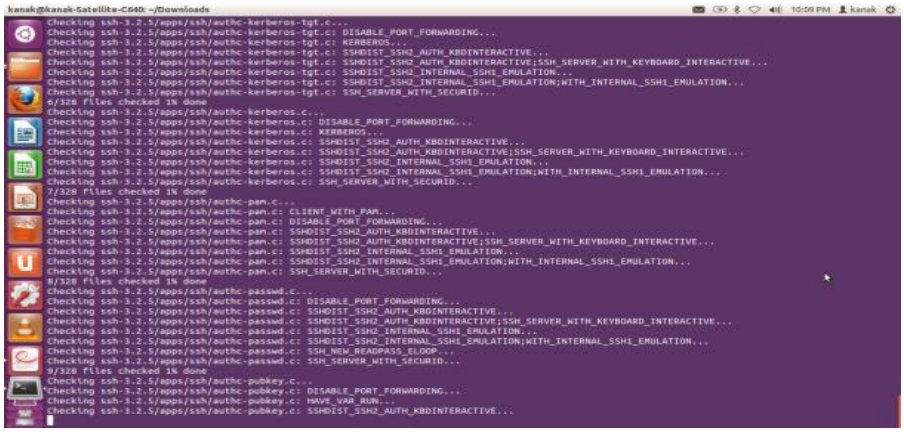

Fig 4. Cppcheck checking ssh-3.2.5 directory

IV. PROPOSED INTEGRATED APPROACH

As seen in the previous section, we have compared both the tools and found some limitations associated with them. Here we propose an approach integrating both of these tools techniques to develop a much better static analysis tool that can detect known as well as unknown vulnerabilities and indicate the locations where improvements are needed. It can be illustrated as follows.

a) Flawfinder is based on text-pattern matching thus enhancing it with the better algorithm like neural-network approach where the instrument can detect vulnerabilities based on previous experience. So, it can evaluate the code even more precisely.

b) As Cppcheck detect the errors not found by the compiler, the tool itself can embed the compiler options in it as well as having the ability to dynamically detect the unknown programming errors.

c) The "rule set" database provided with Flawfinder can be updated with latest library functions, code using functions that apart from finding commonly misused functions can retrieve malicious functions inserted by that of the attacker.

d) As both of these tools are for checking vulnerabilities in C/C++ code only, new conversion methods can be embedded within the developed tool so that it can work in other languages like Java too. Linguistic variable can be used regarding that to set language like 3 for C++, 2 for C etc. As Flawfinder is written in java so compiling it is not a problem, further scripts can be added to it via the conversion function.

Thus, integrating both of these tools can lead to the development of a new static analysis tool whose working is far much better compare to both of them and can detect vulnerabilities not found even by both of them. Diagrammatically, it can be shown below as:

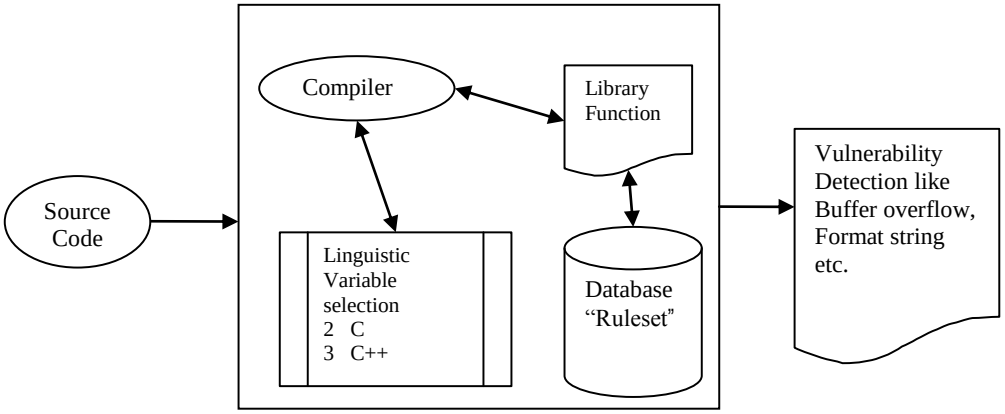

Fig 5.Integrated Approach using Flawfinder and Cppcheck static analysis tools

V. CONCLUSION

Static analysis approach aids a lot in detecting known as well as unknown vulnerabilities which can be exploited by an attacker. In this paper, we have integrated the approach used by two tools and also updating some parts of it and adding certain methodologies. Thus, it can be seen that much more can be done in the static analysis approach for the operating system security.

REFERENCES

[1] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*,Copyright 1999 IEEE, DARPA grant F30602-96-1-0331

[2] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan, " Defending against Buffer-Overflow Vulnerabilities", 2011 IEEE computer society

[3] Wenhua Wang, Yu Lei, Donggang Liu, David Kung, Christoph Csallner, Dazhi Zhang, Raghu Kacker, Rick Kuhn, "A Combinatorial Approach to Detecting Buffer Overflow Vulnerabilities", 2011 IEEE computer Society

[4] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, Xinying Wang,Anh Tuan Nguyen, Tien N. Nguyen "Detecting Recurring and Similar Software Vulnerabilities", Copyright 2010 ACM

[5] Ashish Aggarwal, Pankaj Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities", 2006 IEEE computer society

[6] Michael Howard, James A. Whittaker, "Secure Coding in C and C++", 2006 IEEE COMPUTER SOCIETY

[7] Jesse C. Rabek Roger I. Khazan Scott M. Lewandowski Robert K. Cunningham, "Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code", Copyright 2003 ACM

[8] Michael Zhivich, Tim Leek, Richard Lippmann, "Dynamic Buffer Overflow Detection",

[9] David Evans and David Larochelle, "Improving Security Using Extensible Lightweight Static Analysis", IEEE computer Society 2002

[10] Bas Cornelissen,Andy Zaidman,Arie van Deursen, Member, Leon Moonen, Member,Rainer Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis", IEEE Computer Society 2009

