Static Techniques for Vulnerability Detection

Kamran Zafar Asad Ali
Linköpings university, Sweden
Email: {kamza037,asaal503}@student.liu.se

Abstract

For the last 10 years the importance of building secure software is becoming more and more significant. Since then many techniques and tools were built by developers and researchers to make the software mechanisms secure, and to protect software programs from malicious users and attackers. Existing solution for detecting software vulnerabilities are not very much trustworthy as they all possesses some breaches. In this report we discuss some techniques and tools for vulnerability detection and pros and cons of some tools and technique.

1. Introduction

Software vulnerabilities provide a way to an attacker as vulnerabilities are the well-known and well understood flaws by the carelessness of developer of the software. For example buffer overflow and format string vulnerabilities are most common and well known class of vulnerabilities. In order to identify these vulnerabilities a comprehensive analysis is required to develop some standard solutions against vulnerabilities.

According to statistics from CERT coordination center at Carnegie Mellon University, CERT/CC, in year 2004 more than ten new security vulnerabilities were reported per day in commercial and open source software (see Figure 1) [1]. In addition, the 2004 E-Crime watch survey respondents say that e-crime cost their organizations approximately $666 million in 2003 [2].

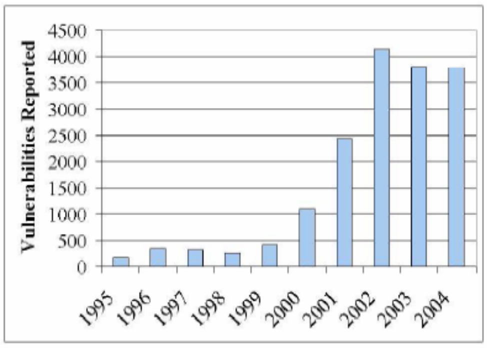

Figure 1: Software security vulnerabilities reported to CERT 1995–2004.

As vulnerability refers to a weakness in software now the question arises that what is a weakness of software? The main reason of vulnerability is due to carelessness of a software developer an attacker can take benefit from this carelessness and execute commands on the system or bypass some access control. There exist many software tools and techniques to discover and to remove vulnerabilities. We have studied 2 known vulnerabilities Buffer overflow and Format string; techniques and tools for their detection.

Vulnerability exists at least to some extent in every software, we cannot neglect it. What we can do is to detect and prevent and/or remove vulnerability present in soft ware.

There are two main approaches to detect or prevent vulnerabilities. Some tools are applied directly to the source code so they either solve or at least warn about presence of vulnerabilities in the source code. These tools are called static tools e.g. ITS4 and Splint. The other types of tools are dynamic tools that check the software at runtime against any known vulnerability e.g. ProPolice and CERD.

Huge amount of vulnerabilities exist in each domain (Operating System, Databases, Network Applications etc) of computer world. So it is not viable to coup with all these in this report and it is rather a big task. From all the domains we have seen so far most common vulnerabilities are Buffer overflow and Format String vulnerabilities. In rest of our report we will focus on these two vulnerabilities and the techniques and tools detecting these vulnerabilities.

1.1 Scope

We will discuss some techniques and tools implementing these techniques for detection of Buffer overflow and Format String vulnerabilities. We will also consider approaches like modified kernels and shell codes. We will give a description of static techniques for detection of the above said vulnerabilities. The static tools are applied to a program's source code.

1.2 Overview
The rest of this paper is organized as follows. **Part 2** illustrates a description of buffer overflow and format string vulnerabilities; how they subsist and how an attacker can take benefit from these vulnerabilities. In **Part 3** we look at static techniques and tools for vulnerability detection. How different tools can be used to handle these vulnerabilities and working of each tool. In **part 4** we present a comparison of tools and techniques, detecting the vulnerabilities. We present a comparison of tools based on the techniques used in tools to detect vulnerabilities. This part shows comparison of static technique and tools. **Part 5** concludes our work and suggestions.

2. Vulnerabilities & Attacks
In computer world the term buffer overflow has become much illustrious among computer security. Almost all Buffer overflows are due to the weakly build software programs.

Vulnerabilities in software can be viewed as mistakes made by the developers or a weakness due neglection of aspects like no checks on stack boundaries etc. Mistakes and weaknesses exist in the code as the developers were unaware of vulnerability creation at the time of development. In particular the misuse of unsafe and error-prone features of the C programming language, such as pointer arithmetic, lack of a native string type and lack of array bounds checking [4]. Figure2 [15] shows relationship between the attack and attacker's knowledge.

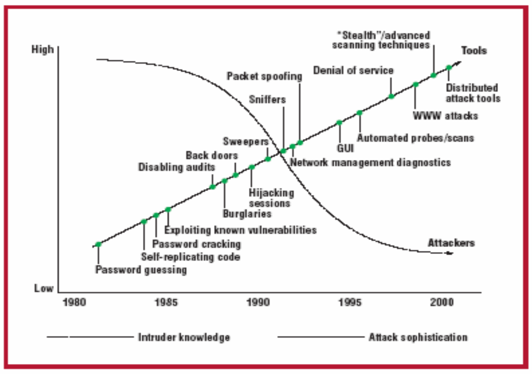

Figure 2 from [15] Attacks Vs Intruder knowledge

Locating the factors that cause the existence of vulnerability is very important. To accomplish this task a deep analysis is required in order to detect attacks against vulnerabilities. Vulnerabilities in software are the main source that make the software risky and provides an attacker a line of attack.

2.1 Buffer Overflow
In software buffer overflows are considered as prime source of vulnerabilities. For example, the CodeRed worm that caused an estimated global damage cost $2.1 billion in 2001 exploited a buffer overflow in Windows [3]. In addition report, on the basis of CERT (Computer Emergency Response Team) advisories, that "buffer overruns account for up to 50% of today's vulnerabilities, and this ratio seems to be increasing over time [3].

2.1.1 Buffer
A buffer is a temporary memory area normally with fixed size; used to hold some inputs or outputs. These inputs or outputs are used to communicate with the outside devices or with the processes inside the computer/operating system.

2.1.2 Buffer Overflow
Usually boundary checks are ignored for fixed size buffers. So when a process starts to store data further than the boundaries/capacity of buffer, the extra data then overwrites the adjacent memory locations and overflow the buffer. This condition is referred to the term buffer overflow. In this condition a process may produce incorrect results or crashes. An attacker can use this flaw of buffers to execute malicious code or make a program error prone. When buffer overflow occurs, a type-safe language or a language with explicit bound checking throws an exception. But in unsafe language like C/C++ buffer overflow exception is not thrown because they allow buffers to be overflowed [10].

An attacker can take benefit from this vulnerability by first finding some way of injecting data in buffers to overflow. Then the attacker execute arbitrary code e.g. **shell code** and gain access of administrative privileges.

A shellcode is a small piece of machine code written in assembly language. Shell codes can be used by an unauthorized person to launching shells with command lines of operating system. This will allow an attacker to type commands just like a regular authorized user or even as a system administrator. Shell code are mostly used to exploit buffer overflows and format string vulnerabilities [16].

2.2 Format String Vulnerability
Format string vulnerability allows an attacker to alter the control flow of an application by using string formatting library features to access other memory space. Vulnerability occur when user-supplied data is used directly as formatting string input for certain C/C++ functions for example fprintf, printf, sprintf, setproctitle, syslog, ...) [5]. A format string is a way of telling the C

compiler how it should format numbers when it prints them [6]. By a format string attack an attacker can execute arbitrary code and read the values of the stack.

2.2.1 Format function

A format function takes variables arguments as a format string e.g. %s, %d etc.

Example printf("Home Address : %s ", C - 90)
The output would be Home Address: C – 90

In this case as user supplied data is included in printf function as a format specification string, so this constitutes a format string bug. This leads to information disclosure and potentially the execution of arbitrary code. Normally, the format string is stored on the stack [7], so we can use the format string itself to supply arguments that the printf function will use when evaluating format specifiers.

3. Techniques & Tools

Basically the techniques for detecting and/or removing vulnerabilities lie in two categories, Static and Dynamic. First we will give the overview of both techniques i.e. static & dynamic and then we will give a description of how these techniques detect, prevent or remove vulnerabilities.

3.1 Static & Dynamic Techniques

In *static vulnerability detection* technique source code is analyzed in order to find vulnerabilities. The source code is checked against the known vulnerabilities and a tool implementing static technique detects the existing vulnerability. The two main drawbacks of this approach is that someone has to keep an updated database of programming flaws to test for, and since the tools only detect vulnerabilities the user has to know how to fix the problem once a warning has been issued [8]. In static code analysis source code is checked before compilation, against known vulnerabilities. The static code analysis technique address problems like array bound check, un-initialized variables, unreachable code, syntax problems, undeclared variables, parameter type mismatch, uncalled function and procedure, non-usage of function results and misuse of pointers [9].

Dynamic vulnerability detection technique [8] is a run time technique that detects and/or removes vulnerabilities and attacks by changing functionality of a system or run time environment in order to prevent vulnerabilities and attacks. In this technique typically a program is terminated in case if a vulnerability is detected, this technique also cope with bugs that are known already and this dynamic technique is useless if an attacker attack using some other way.

In the following section we have presented 5 static techniques and description of the tools which are using these techniques for detection of buffer overflow and format string vulnerabilities.

3.1.1 Pattern matching

This technique tries to find all occurrences of strcpys followed by all sprintfs in the source code. This technique is adopted by a tool **grep** which performs simple string matching. This technique checks all the calls to strcpy() to determine either they are safe or not. Lacking a proper C parser, a pattern matching tool is unable to tell apart comments from real code and is easily fooled by unexpected white space and macros [11].
Since grep is only performing simple string matching, its false positive rate can be quite high [10]. **grep** searches the input files for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output [17]. Another tool that implement this technique is Flawfinder[13] which detects vulnerability by using pattern matching technique. This tool analyzes the source code and scans it to figure out buffer overflow and format string vulnerabilities by using its database for C/C++ functions and produce a list of flaws sorted by risk level. The Flawfinder 0.19 vulnerability database contains 55 C security bugs [8].

3.1.2 Lexical analysis

Lexical analysis builds a token stream of the code in order to make a distinction between variables of a function and to identify arguments of a function. These tokens are then matched with existing vulnerabilities. Lexical analysis improves the accuracy of pattern matching, because a lexer can handle irregular whitespace and code formatting. Lexical analysis techniques are fast and simple, but their power is very limited since they do not take into account the syntax or semantics of the program.
Unfortunately, the benefits of lexical analysis are small and the number of false positives reported by these tools is still very high [8].

Tools like RATS[10] and ITS4[8,19] follow this technique. ITS4 scans the code, performs a lexical analysis and build a token stream of the source code. It has the ability to analyze other languages as well as C. This tool has a modular design which allows for integration in various development environments by replacing its front-end or back-end.

3.1.3 Annotations
Annotation is the information about a particular point in the document. In a source code annotations are the comments given by the developer which are used by static analyzers to analyze the code, for example;

Strcpy(char *s1, char *s2)

Annotation /* maxSet(s1) >= maxRead(s2) */

This annotation is used to ensure that s1 must be big enough to hold all characters from s2. This technique is used to check buffer overflow and format string vulnerabilities. This technique is implemented in a tool SPLINT [8], which performs a static analysis on syntactic level by using programmer provided semantic comments and uses a program's parse tree. This tool can be used to detect problems such as NULL pointer dereferences unused variables, memory leaks and buffer overruns [11].

3.1.4 Parsing
This technique parses the source code and builts an abstract syntax tree representation of the code. The abstract syntax tree allows us to analyze not only the syntax, but also the semantics of a program. This task is performed by compiler. To be able to correctly parse and analyze a wide range of programs, a static analysis tool needs a parser compatible with at least one of the major compilers. Integrating with a compiler frontend will ensure this compatibility. For this reason most of the advanced analysis tools on the UNIX platform utilize the GCC (GNU Compiler Collection) frontend, which is freely available under the GPL (General Public License) license. Lexical analysis tools can be confused by a variable with the same name as a vulnerable function, but AST (Abstract Syntax Tree) analysis will accurately distinguish the different kinds of identifiers. The pattern matching approach can be significantly improved by matching AST trees instead of sequences of tokens or characters. On the AST level macros and complicated expressions are expanded which can reveal vulnerabilities hidden from lexical analysis tools. [11]. This technique makes a fairly complete and easy to navigate representation of a program. This technique is used in one of the earliest C static source analysis tools, lint [20].

3.1.5 Type qualifiers
Jeffrey Foster [21] has described type qualifiers, a lightweight, type-based mechanism, to improve the quality of software. Type qualifiers are lightweight annotations for specifying program properties. According to the author In particular, type qualifier systems, when applied to type-safe languages, are sound, meaning that programs with valid qualifier annotations do not violate the semantics of the qualifiers. This assurance enables the programmer to use type qualifiers to eliminate whole classes of bugs from their program. In his system user-defined type qualifiers are added by annotating the source code and detecting type inconsistencies by type qualifier inference. This technique is implemented in a tool Cqual[21] for adding user-defined flow-insensitive and flow-sensitive type qualifiers to C to find format string and buffer overflow vulnerability.

This technique is implemented in a tool BOON [8] used to detect buffer overflow vulnerability. This tool first analyzes the strings variables. Then checks the variables according to the allocated size and number of bytes currently in use. This tool parses the code and reports any detected vulnerabilities.

3.1.6 Data-flow Analysis
Data-flow analysis is a traditional compiler technique for solving buffer overflow and format string problems and can be used as a basis of vulnerability detection systems [11].

3.2 Few other techniques
Samuel Z. Guyer [22] presented a technique, automatic compiler-based approach for detecting buffer overflow and format string vulnerability. Their system uses a configurable and scaleable whole-program dataflow analysis engine driven by high-level programmer-written annotations. According to them this system automatically detects all known errors in five medium to large C programs without producing any false positives. They cast vulnerability detection as a dataflow analysis problem which their compiler solves using a configurable dataflow analysis engine.

Nurit Dor [23] presented a technique that statically uncovers all string manipulation errors. They implemented this technique in a tool CSSV (**C** **S**tring **S**tatic **V**erifier) that statically uncovers all string manipulation errors. This technique performs a static analysis to detect all string runtime errors with just few
false alarms. They implemented this technique in different phases to detect vulnerabilities.

John Viega [10] presented a technique for statically scanning security-critical C source code for vulnerabilities. Their scanning technique stakes out a new middle ground between accuracy and efficiency. Their method is efficient enough to offer real-time feedback to developers during coding while producing few false negatives. This method is also simple enough to scan C++ code despite the complexities inherent in the language. Using ITS4 they found new remotely exploitable vulnerabilities in a widely distributed software package as well as in a major piece of e-commerce software.

David Larochelle [18] have presented a technique to mitigate buffer overflow vulnerabilities by detecting likely vulnerabilities through an analysis of the program source code. Their approach exploits information provided in semantic comments and uses lightweight and efficient static analyses. Their approach is implemented by extending the LCLint annotation-assisted static checking tool. Their tool is built upon LCLint. Their technique exploits semantic comments *(annotations)* that are added to source code and standard libraries.

In the above section we have presented some static detection techniques for buffer overflow and format string vulnerabilities. Next section of this report contains a comparison of different techniques.

4. Comparison & Evaluation

Static analysis techniques have several advantages over run-time techniques. Static techniques find errors by analyzing the source code and do not require running the program. They do not incur run-time overhead and they narrow down the vulnerabilities specific to the source program being analyzed, yielding a more secure program before it is deployed [10]. However, a pure static analysis can produce many false alarms due to the lack of vulnerabilities related information. As static techniques detect known vulnerabilities.

In table 1 we gave a comparison of tool and techniques bases upon the use of a technique in a tool.

We have discussed 10 techniques out of them 6 techniques are adopted by a tool and 4 techniques are not yet commercially implemented in a tool.

Tech. / **Tools**	Pattern Matching	Lexical Analysis	Annotatio ns	Parsing	Type Qualifiers	Data Flow analysis
Grep	*					
Flawfinder	*	*				
RATS		*				
ITS4		*		*		
SPLINT			*	*		
Lint				*		
Cqual					*	
Guyer [22]						*
Dor [23]			*			
Viega[10]			*			
Larochelle [18]			*			

Table 1: comparison of tools and techniques

The comparison given in table 1 presents the particular techniques used by a particular tool. From this result we concluded that most of the tools used the technique that performs analysis of a program source code and exploits information provided in semantic comments, implementing *annotation* technique.

Tools like SPLINT and the tools presented by Nurit Dor[24], John Viega[10], David [18] are using annotation technique. Splint is the only tool that can distinguish between safe and unsafe calls to strcat() and strcpy() [8]. Splint is using annotation technique which implicates that this technique has a good possibility to accurately detect security bugs with a low rate of false positives. Therefore this approach is used by many people given in section 3.2; they have used annotation technique in their tool along with some enhancement to make it more powerful.

The technique presented by Dor [23] in their tool is statically detecting string errors for buffer overflow and format string vulnerabilities. Their technique is also handling multilevel pointers and structures. According to them their technique is detecting all C security vulnerabilities in a precise manner where as other techniques like Lint is not performing this task successfully.

J. Viega [11] have presented a technique for static analysis of C/C++ source codes. They implemented their technique in ITS4. According to them the parsing model of ITS4 makes it poorly suited for highly accurate static analysis. But with their technique implemented in the same tool makes the tool efficient for static analysis of the program. With their technique ITS4 is scanning large programs efficiently and achieving adequate results.

5. Conclusion

It is very difficult that a tool completely detect all the security vulnerabilities without false alarms. As more and more software are developing day by day so vulnerabilities are also growing continuously. Although significant work is done to cope with buffer overflow and format string vulnerabilities a satisfactory solution is still needed. But by recent tools like ITS4, Flawfinder & SPLINT these vulnerabilities can be handled to some good extent. Security can be improved if vulnerability detection tools are used as a part of software development lifecycle.

We have discussed different static techniques and tools to detect buffer overflow and format string vulnerabilities, and compared available static tools based on the technique they use.

References

[1]CERT Coordination Center, “CERT/CC Statistics 98-2004” April 2007, http://www.cert.org/stats/cert_stats.html

[2] CSO magazine, U.S. Secret Service, and CERT Coordination Center. 2004 e-crime watch survey. http://www.csoonline.com/releases/052004129_release.html, April 2007.

[3] S. Chaki, S. Hissam; Software Engineering Institute, Carnegie Mellon University, USA.

[4] A.I SOTIROV.; Automatic vulnerability detection using static source code analysis; Final thesis Department of Computer Science in the Graduate School of University of Alabama. April 2007
Online: http://gcc.vulncheck.org/sotirov05automatic.pdf

[5] Web Application Security Consortium, April 2007 http://www.webappsec.org/projects/threat/classes/format_string_attack.shtml

[6] What is a Format String Vulnerability? April 2007 http://www.tech-faq.com/format-string-vulnerability.shtml

[7] Introduction to Format String Bugs; Chapter 4 Page(s): 55-82
http://media.wiley.com/product_data/excerpt/83/07645446/0764544683.pdf

[8] J. Wilander, M. Kamkar. A comparative study of publicly available tools for static intrusion prevention. In Proceedings of the 7th Nordic Workshop on Secure IT Systems, Karlstad, Sweden, November 2002.

[9] Code Analysis Steven Lavenhar, Cigital, Inc. U.S Dept. of Home land security, May 2007.
https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/code/214.html

[10] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, .ITS4: A static vulnerability scanner for C and C++ code,. in 16th Annual Computer Security Applications Conference. ACM, Dec. 2000

[12] J. Viega, J. Bloch, T. Kohno, and G. McGraw, "Token-Based Scanning of Source Code for Security Problems", ACM Transactions on Information and System Security, Vol. 5, No. 3, August 2002, Pages 238–261. (About development of ITS4.)

[13] David A. Wheeler. Flawfinder, April 2007 http://www.dwheeler.com/flawfinder/

[14] R. Lippmann, M. Zhivich, T. Leek,; Dynamic buffer overflow detection. In Proceedings of Workshop on the Evaluation of Software Defect Detection Tools, june 2005.

[15] McHugh, J.; Christie, A.; Allen, J.; IEEE Software Volume 17, Issue 5, Sept.-Oct. 2000 Page(s):42 - 51

[16] Introduction to Shellcoding - How to Exploit Buffer Overflows, 22 April 2007.
http://www.securiteam.com/securityreviews/6Q00L00BFI.html

[17] Introduction to grep tool, 30 April 2007.
http://www.gnu.org/software/grep/doc/grep_1.html#SEC1

[18] D. Larochelle, D. Evans.; Statically detecting likely buffer overflow vulnerabilities. Proceedings of the 10th USENIX Security Symposium, USENIX: Washington, DC, 2001; Page(s) 177-189.

[19] Secure Software. RATS – rough auditing tool for security, 01 May 2007.
http://www.securesoftware.com/resources/tools.html.

[20] Lint, a C Program Checker, *S. C. Johnson*
Bell Laboratories Murray Hill, New Jersey 07974

[21] S. Foster. Type Qualifiers: Lightweight Specifications to Improve Software Quality. PhD thesis, University of California, Berkeley, Dec. 2002.

[22] S. Z. Guyer, E. D. Berger, and C. Lin. Detecting errors with configurable whole-program dataflow analysis. In Proc. Conference on Programming Language Design and Implementation, Berlin, Germany, 2002.

[23] Dor, N., Rodeh, M., and Sagiv, M. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 155–167, June 2003.