Software Security Assessment smbftpd-0.96

Using manual analysis and static analysis tools: Splint, Rats, and Flawfinder

Eman Alashwali
Information Security, Computer Science
University College London (UCL)
London, UK
eman.alashwali.10@ucl.ac.uk

***Abstract*—This paper aims to assess security for smbftpd-0.96 program that contains a format string vulnerability. The assessment will be achieved manually at first, then by using three different software static analysis tools which are: Splint, Rats, and Flawfinder. The paper will discuss each tool's assessment results individually in order to answer the following questions: Whether the tool discovered the format string vulnerability? Whether the tool found other vulnerabilities? And whether the tool was helpful?. At the end of the paper, there will be a rough comparison between the three tools to summarize what are the common and unique results between the tools.**

Keywords-manual analysis; static analysis; Splint; Rats; Flawfinder.

I. INTRODUCTION

As software applications usage increases, securing these software becomes more essential [1]. Building secure software is a real challenging mission. It is not uncommon to hear from software security experts that there is no 100% secure software. The number of discovered security flaws confirms such claims. In 2011 alone, the National Vulnerability Database (NVD) [2] published 4,151 vulnerabilities under the software flaws criteria. Software vulnerabilities are the main source for software security incidents [3] [1]. These vulnerabilities are caused by many different reasons, they can be a result of flaws, back doors made by the developer, or weaknesses in the programming language itself [1]. Vulnerabilities are bugs that that can be exploited by untrustworthy to launch attacks against the system.

One of the well known vulnerabilities in software security area is format string vulnerability. Format string vulnerability occurs when an untrustworthy has the ability to provide arbitrary characters in the format string to a format function [4]. To exemplify, the following code: printf(string) is an example of format string vulnerability, while the correct method of writing this code is: printf("%s",string) [4]. Reference to the NVD [2], format string vulnerability can result in dangerous consequences such as: violating confidentiality, integrity and availability, unauthorized information disclosure, service disruption and unauthorized access.

Despite the proposed solutions by researchers for the format string vulnerability since it was publicly recognized as a type of security attacks in 1999, it continued to appear years after that [4]. For example, the Common Vulnerabilities and Exposures (CVE) database hit 400 entries for format string vulnerabilities for the year 2007 [4].

To address this problem, various techniques have been developed to improve software security and discover software vulnerabilities. Manual analysis and Static analysis are some of the used techniques to spot and correct software flaws [5].

This paper aims to assess the security of SmbFTPD-0.96 program which is a FTP daemon built based on the FTP daemon of FreeBSD 5.4 [6], by using manual analysis first, and then by using the following static analysis tools: Splint, Rats and Flawfinder and given that it contains a format string vulnerability. This is to answer the following questions: Do the tools find the vulnerability? Do the tools find other vulnerabilities? Are the tools helpful?. Also, the paper is going to compare the three tools to answer the following two questions: What are common results? What are unique results? This paper will not cover all security warnings from the tools, but only the most important vulnerability and bugs.

The coming sections are divided as the following: Section 2 will discuss the manual analysis. Section 3 will discuss automated analysis using three different static analysis tools: 1) Splint, 2) Rats, and 3) Flawfinder. Section 4 will show the common and unique results. Finally, in section 5 the conclusion.

II. MANUAL ANALYSIS

As its name implies, manual analysis requires human(s) to manually test the code [7]. It is one of the oldest methods to analyze the code and it requires reading the source code thoroughly looking for vulnerabilities [3]

In our experiment, in order to perform manual analysis for smbftpd-0.96 to find vulnerabilities, this required us to have knowledge about the most common vulnerabilities such as buffer overflow, format string, etc. and how they occur. Then, looking at each line of code that is prone to any of these vulnerabilities. We achieved this by using "find" and "egrep" linux tools to search for some functions existence in all C files in smbftpd-0.96 and display their contexts. The command we used is:" find . -exec egrep --colour -n -i -H '*printf*|setproctitle|syslog' /home/e/Desktop/CW1/smbftpd-0.96/filename.c {} \; "

Where the value of “filename” stands for the C file that we want to analyze.

In our experiment, using “find” and “egerp” tools, it was possible to perform the search on both directory and file level. We preferred to analyze one file at a time since analyzing all C files at once using this method will produce a tremendous amount of lines and is a processor exhaustive for our experiment's intel centrino2 laptop with 4 GB RAM. The command must include the function name(s) that the known vulnerabilities are more likely to occur in. For example, we search for strcpy to locate buffer overflow vulnerabilities, and search for format functions such as printf, sprintf, etc. to look for format strings vulnerabilities. We observed that when we tried to include other format functions with the * symbol to include all possible functions such as: err*, verr*, warn*, vwarn*, the search time got extremely long since the letters “err” and “warn” occurred in many contexts such as comments and not only in the functions we are looking for. The command's output shows the file name, line number, and the context that contains the function we search for, and the function appears in color. After the command result's displayed, we read the displayed lines thoroughly to make sure they are not vulnerable. If required, we viewed the full code by opening the file in a text editor for further checking. The following figure shows the command output.

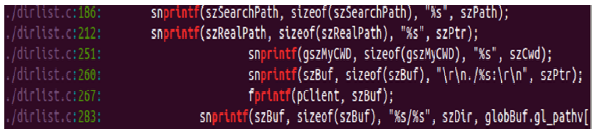

Figure 1. Part of the output for the following command: find . -exec egrep --colour -n -i -H '*printf*|setproctitle|syslog' /home/e/Desktop/CW1/smbftpd-0.96/”filename”.c {} \; for manual analysis

By using manual analysis to assess smbftpd-0.96, we were able to find the format string vulnerability in “dirlist.c” file in “SMBDirList” function in line 267 after searching about *printf*. It was possible to identify the format string vulnerability since we were looking in small number of files for a specific type of vulnerabilities that occur in limited number of functions. We considered line 267 as a format string vulnerability since it uses a format function as the following: fprintf(pClient, szBuf) which allows an attacker to exploit this vulnerability by listing specially crafted filenames, which will allow him to run arbitrary commands after login (including anonymous login), while as explained previously in the introduction, the correct code must be: fprintf(pClient, "%s", szBuf). We also tried to find buffer overflow vulnerabilities manually by searching for strcpy functions. We found this function used many times in more than file. Clearly, using this unsafe function represent a bug. It can be exploited if the source string does not have bound checking. For example, in file smbftpd.c, we found the following suspicious piece of code in line 337: strcpy(szReturnPath, szCurDir), but since szCurDir is a value that is not going to be entered by untrustworthy (any outside user), we considered this a bug but not a vulnerability. Similar bugs exists in other files such as ftpd.c, misc.c.

Manual analysis was tedious but, at the end, possible. However, to locate other vulnerabilities with this method, this was not possible (at least in a reasonable amount of time). Since this will require us to specify all the suspicious function names, and search for them in every single file. In addition to that, this method is very slow and as much as we include more functions in the command, the search process get slower. With manual analysis, the number of false positive results was very high as the command we used is merely a matching process that outputs the functions we search for whenever occurred without any checking for the correctness of the code by any means. The checking and deciding is completely left for the analyzer.

Manual analysis is known to be time and labor consuming process [3] [7]. Some might argue that it is the best method. However, its efficiency mainly relies on human knowledge and experience [3] [5]. It requires full understanding for the code, in addition to computer security experience [3]. It is a feasible method if the analyzer knows what he is looking for, i.e specific type of vulnerabilities and in specific file, while this might be extremely difficult for large programs and different types of vulnerabilities.

III. AUTOMATIC ANALYSIS USING STATIC ANALYSIS TOOLS

To overcome the difficulties in manual analysis, several solutions have been found to automate the analysis process, static analysis tools are examples of such solutions. Static analysis is the process of examining and making a judgment about the source code without executing it [3]. A number of advanced and effective static analysis tools have been developed in the recent years [1]. Splint, Rats, Flawfinder are examples of static analysis tools produced to test source codes. In order to test these tools, we are going to make a security assessment for the source codes of smbftpd-0.96 using Splint, Rats and Flawfinder.

A. Splint (Secure Programming Lint)

Splint is an open source static analysis tool that scans for security vulnerabilities in C code [8]. In addition of performing most of its predecessor (Lint tool) checks, it does more advanced checks by using annotations which are stylized comments that gives inference about variables, functions, parameters and types [8] [5]. Annotations are added to libraries and code to document the programmer intents [5]. Splint checks whether the code matches the specifications mentioned by annotations [5]. Splint can detect wide range of problems such as: Buffer overflow, dereferencing a possibly null pointer, type mismatches, memory management errors such as memory leaks, problematic control flow such as possibility for infinite loops [8]. Unlike other tools, it checks for coding style and errors that are not related to security [9]. Splint can perform the checking for a directory or a single file.

After running splint against our code for smbftpd-0.96, testing one file at a time. When we tried to test “dirlist.c”, using the command: “splint -Iinclude/ +posixlib -preproc -D__gnuc_va_list=va_list dirlist.c”, we got 109 warnings which is a large number and this is for one file consist of 319 lines of code only. The output gives the filename, line number, column number, description of the warning, and sometimes, a possible reason for the warning and suggestion for how to inhibit a warning. Splint could detect the format string vulnerability in “dirlist.c” in “SMBDirList” function in line 267. In addition to that, it detected another 108 warnings. One reason for this high number of warnings is that our code is not annotated and splint depend on annotations. Using code annotations can reduce the number of warnings. For example, the following warning:
“ dirlist.c:182:25: Null storage szPath passed as non-null param: strchr (szPath, ...) [..] “ can be resolved by adding a /*@null@*/ annotation to the function parameter declaration. In addition to annotations, splint manual has a long list of command flags to customize the output. For example, the argument “-nestcomment” will ignore warnings such as the following: “ dirlist.c:193:34: Comment starts inside comment A comment open sequence (/*) appears within a comment. [..]”. Also, the -weak flag will make weaker checking but the analyzer must be careful in using these arguments to avoid overlooking real vulnerabilities. We customized our command to be: “ splint -nestcomment -Iinclude/ +posixlib -preproc -retvalint -compdef -predboolint -usedef -mustfreefresh -type +ptrnegate -retvalother -D__gnuc_va_list=va_list dirlist.c | less”, and we got 27 warnings, one of them is the format string vulnerability in line 267, and the rest 26 are false positives.

Splint output was difficult to read and it was difficult to prioritize the severity warnings. Splint developers say: “Our design criteria eschew theoretical claims in favor of useful results.” [5]. Splint uses heuristics to analyze the code and in order to increase the class of properties to be checked, it has to sacrifice the results correctness and completeness, i.e, it will warn falsely and it will not detect all problems [5]. In order to get more sound reports, the code must be annotated. Splint reports warnings that resulted from inconsistencies between the source code, annotations and convention of the language [5]. The needed effort in annotating codes stands an obstacle for adopting annotations [5].

B. Rats (Rough Auditing Tool for Security)

Rats is an open source analysis tool for vulnerability scanning. It supports several programming languages such as: C, C++, PHP, Python and Perl [10]. It discovers common errors such as buffer overflow, time of check, time of use (TOCTOU) race conditions [10]. It outputs a list of possible problems, along with a short description and suggested solution. Also, it provides a rough assessment of the problem severity (1: High, 2: Medium, 3: Low) which might help code analysts to manage the problems' priorities. Rats As its name implies, makes rough checking. Therefore, it might not discover all vulnerabilities, and might report false positives. It performs basic analysis to avoid the conditions that are obviously not a risk [10].

Rats can analyze a directory or a single file. If the command given a directory name, Rats will check all C files inside this directory. For more convenient output, Rats provides several options such as -w which allows the analyzer to list warnings belongs to a certain severity level (1, 2 or 3). It also has a useful option -i which will display the functions that takes external input at the end of the report [10] the thing that makes the report more readable.

In our assessment we tried both levels, directory and file. We preferred to perform the assessment by testing one file at a time. For example, when we analyzed “dirlist.c”, the file that contains the format string vulnerability, we typed the following command: “rats dirlist.c”. Rats discovered the format string vulnerability and classified it as (High). Also, it discovered another 9 vulnerabilities under the same severity level. Using Rats in other files, we found that most of false positive warnings are repeated such as the following: “High: fixed size local buffer”, that is displayed for any fixed size array as a caution for buffer overflow when dealing with fixed size arrays. For example, in dirlist.c line 37, we have char szPerm[11] when we checked it in the code, we found that it will be used in a strcpy function, but the source string is a fixed sstring as the following: strcpy(szPerm, "lrwxrwxrwx"), which does not represent a risk since the source string is less than or equal to 10 charachters. Also, running Rats against different files, we observed that Rats reports any use of syslog, without checking whether is it used correctly or not.

Rats uses lexical analysis instead of parsing the code [5]. Therefore, it gives imprecise warnings as it warns whenever it finds a name of unsafe function written in another ways (for example: local user-defined functions) without checking whether the reported function is used in a safe different method or not [5].

C. Flawfinder

Flawfinder is another open source static analysis tool. It is limited to C and C++ programming languages only [11]. It can handle many types of security problems such as: format string, buffer overflow, race conditions, poor random number acquisition and potential shell meta character dangers [11]. Although, it is similar to RATS in its approach, but it has some features over other tools. Flwafinder can deal with the gettext (“ a common library for internationalized programs”), the thing that reduces the false positive hits in internationalized programs [11]. Second, it can give better risk priority levels as it determines the risk level by the values of the parameters of the function in addition to the function itself, for example, in many contexts, constant strings has lower risk level than variable ones [11]. Third, it provides the option to show the context where the flaw occurred which makes it easier to use and will save the analyzer’s time [11].

The output shows a list of the potential risks with the filename, line number along with the risk level. There are five levels of risks in flawfinder starting from 5: The most severe, till 1: the less. Flawfinder provides several flags to customize the output, some useful flags that we used in our experiment are: --minlevel that sets a minimum level of risks to be displayed in the output, and –context that shows the context that the risk occurred in. In addition to that, the output can be produced the in HTML format using –html flag which when exported to html file gives a more readable output than on the command line. For the above mentioned features, we believed that flawfinder is a good option for a third tool to be used in our experiment.

Flawfinder can analyze code in directory and file levels. In our experiment, again, we preferred to test files one by one. For example, to analyze the code for "dirlist.c" file, we used the following command:" flawfinder --context –minlevel=3 /home/e/Desktop/CW1/smbftpd-0.96/dirlist.c >results.html ". After running the command, flawfinder could find three potential vulnerabilities "hits". It could find the format string vulnerability in addition to another two vulnerabilities but they are false positives.

The tool was easy to use, the flags that the tool offers to adapt the output resulted in a short report that provided us with the real vulnerability in our code (the format string vulnerability) with the minimum false positive records. However, the results are not very accurate as it classified the format string vulnerability in level 4, while a false positive was in level 5.

IV. COMMON AND UNIQUE RESULTS

After trying three tools, it is convenient to summarize the common and unique results in these tools. All the three tools were open source tools, easy to get and install. They could discover the format string vulnerability in "dirlist.c" line 267. All of them gave some flags in order to customize the output as desired, for splint, it was necessary to use these flags (or add annotations) to get a reasonable amount of errors. The three tools differ in some points, here is a summary based on our experiment.

SUMMARY TABLE

	Splint	***Rats***	***Flawfinder***
Ability to detect vulnerabilities	High. Can find large scale of faults.	Medium. Could detect the main one.	Medium. Could detect the main one.
Ease of use	Low. Requires understanding the notations. The command needs customization in order to get reasonable output.	High. The output can still be reasonable without using any flags.	High. The output can still be reasonable without using any flags.
Report Accuracy	Low. Can be higher with proper code annotation	Medium. Medium rate of False positive.	Low. High rate of false positive.
Languages support	Low. (C / C++)	High.C/C++/ Perl/ Python)	Low. (C / C++)
Tool's Scope	General purpose.	General purpose.	General purpose.
Technique	Rule checking	Lexical analysis	Lexical analysis
Code treatment	Need pretreatment for the code.	No need for pretreatment	No need for pretreatment
Report format	Hard to understand, does not include the context, does not classify risks, and does not suggest solution.	Easy to read, risks classification in 3 levels, ordered by risk level, and suggest solution	Easy to read, risks classification in 5 levels, ordered by risk level, suggest solution

Table 1: Summary of the main differences between the three tools

V. CONCLUSION

Static analysis tools are great aid for programmers. They spot suspicious code that need to be checked which is the first step to detect vulnerabilities but they will not fix them. Therefore, they can never replace human manual checking and judgment as they can not detect design problems and can not replace secure implementation. Also, they have the false positive and false negative problem. There is a trade off between the ease of use and report simplicity and the complexity of the analysis they perform and the scope of the problems they cover. Nothing is perfect! And there is no perfect static analysis tool. Human knowledge and experience still needed.

VI. REFERENCES

[1] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference*, 17-19 Dec. 2010, pp. 521-524.

[2] National Institution of Standard and Technology (NIST). 2012. National Vulnerability Database. [Online]. Available: http://web.nvd.nist.gov

[3] A. Stirov. 2005, *Automatic vulnerability detection using static code analysis*. [Online]. Available: http://gcc.vulncheck.org/view/vulnsotirov05automatic.pdf

[4] K. Chen and D. Wagner, "Large-ScaleAnalysis of Format String Vulnerabilities in Debian," in *2007 workshop on Programming languages and analysis for security (PLAS '07)*, SanDiego,California, 2007, pp. 75-84.

[5] D.Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *Software, IEEE* , vol.19, no.1, pp.42-51, Jan/Feb 2002

[6] C. Wang. 2012, *Smbftpd*. [Online]. Available: http://www.twbsd.org/enu/smbftpd/index.php

[7] G. Saha. 2008. *Ubiquity.* [Online]. Available: http://ubiquity.acm.org.libproxy.ucl.ac.uk/article.cfm?id=1348484

[8] D. Larochelle D. Evans. 2003. Splint. [Online]. Available: http://www.splint.org

[9] T. La. 2002. SANS. [Online]. Available: http://www.sans.org/

[10] Fortify. 2011. *RATS Rough Auditing Tool for Security.* [Online]. Available: https://www.fortify.com/ssa-elements/threat-intelligence/rats.html

[11] David A. Wheeler. *Flawfinder.* [Online]. Available: http://www.dwheeler.com/flawfinder