IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-7, Issue-2, Feb 2019 E-ISSN: 2347-2693

Analyzing the Vulnerability in Open Source Software

Madanjit Singh[1]*, Munish Saini[2], Manevpreet Kaur[1]

[1]Department of Computer Science, Guru Nanak Dev University, Amritsar 143005, India
[2]Department of Computer Engineering and Technology, Guru Nanak Dev University, Amritsar 143005, India

*Corresponding Author: mdnjtsingh@gmail.com, Tel.: +91-9501715960

DOI: https://doi.org/10.26438/ijcse/v7i2.815 | Available online at: www.ijcseonline.org

Accepted: 8/Feb/2019, Published: 28/Feb/2019

***Abstract*—** Secure code is one of the key parameters which must be taken care while software is being developed. Inspecting the source code at the earlier stages is always a better approach. Inspection involves carefully examining the source code for any flaws which may cause problems in the later stage of the software life cycle. The Vulnerability is a kind of weakness or security flaws in code that can be exploited by an attacker to perform unauthorized actions. A vulnerable code will lead to severe threats to the security of software. In this paper, we have investigated the source code of a well-known open source software (OSS) projects written in C and C++ programming language and figure out the presence of vulnerability in the software. The results also indicate that the vulnerabilities in the source code have shown an increasing trend with the lines of code (LOC). It pointed to the fact that addition of new features or change request into the OSS project will cause an increase in the vulnerability as well. It gives significant implication to the developers or project managers of OSS projects to not deny the existence of security flaws in the software as the software evolves. The obtained results will also help the project managers and developers to measure the state of software.

Keywords— Open Source Software, Software Quality, Hits, Flawfinder, Vulnerability, Code Scanning tools.

I. INTRODUCTION

The writing of a secure code is more concerned than simply writing code to fulfil the purpose [1]. Every software developer must keep care of writing a secure code for the sake of security of particular software. Any loophole left behind in source code will lead to serious consequences in later stages or after implementation of the software. Most of these software vulnerabilities can be traced back to a few mistakes that programmers make over and over again [2]. In this case, software inspection plays a vital role in assuring that any security flaws should not present in the code.

Moreover, the cost of finding any security bug and fixing it in the earlier stage is much lesser than as compared with the cost of post-implementation process of software. ENISA (European Union Agency for Network and Information Security) defines vulnerability [3] as "The existence of a weakness, design, or implementation error that can lead to an unexpected, undesirable event compromising the security of the software, computer system, network, application, or protocol involved". One of the major concerns of our research is to find the vulnerability in the source code. The conducted study has a high significance in terms of software quality. The early detection of vulnerabilities or security flaws will help the project managers and developers to measure the state of software. The existence of high vulnerability or security flaws will degrade the quality of software and it may lead to the major failure to the software, if not checked and removed timely.

So, the purpose of this study is to examine the OSS projects source code. It explores answers to the following research questions:

1. Examining the existence of the vulnerability in the source code of OSS.
2. Analyzing the trend in vulnerability with respect to the addition of new features or change request into the OSS.

The rest of the paper is organized as follows. Section II presents the related work Section III gives details of the vulnerability and its types Section IV explain the analysis tool. Section V gives details on data collection and results analysis. We have classified the results according to the level of threats detected in the OSS. It also discusses and justifies the interpretations. The last section concludes the paper.

II. RELATED WORK

The research goals are mainly aimed on how these flaws will weaken the overall quality of the software. Although a lot of work has been done in categorizing vulnerabilities with many different taxonomies being made, each with its own relative

strengths and weaknesses, little work has been done in categorizing the available countermeasures to some important vulnerabilities [4], [5].

Yves Younan defines possible vulnerabilities in C and C++ applications that could lead to situations that allow for code injection and describes the techniques generally used by attackers to exploit them. A fairly large number of defence techniques have been described in the literature, but still, the satisfying results have not achieved yet. Few security flaws can get detected automatically by the compiler, but still, a majority of them are identified by the tedious auditing of the source code [6]. To improve this situation, Fabian Yamaguchi purposed a manual auditing mechanism named Chucky, a method used to expose the missing checks in the source code. This method proceeds by statically tainting source code and identifying anomalous or missing conditions linked to security critical objects. Similar to other methods for the discovery of security flaws, Chucky cannot overcome the inherent limitations of vulnerability identification.

In practice, checking tools such as Microsoft PREfast [7] or PScan [8] are used to statically find vulnerabilities in source code. These tools possess built-in information about correct API usage and common programming mistakes, which severely limits the kind of vulnerabilities these tools can detect. An alternative approach is taken by scanners such as Splint, which allow code annotations to be supplied by analysts. However, creating these annotations and rules regarding API usage is both times consuming and challenging as it requires an intimate understanding of both internal and external APIs of a target program.

The software development is always expected to be flaws free, but the existence of vulnerabilities will never be denied. In this paper, we have analyzed the source code of the various versions of MySQL-server to find the vulnerabilities and its trends, as the software evolves. The obtained results have shown that the number of vulnerabilities is always getting affected with the increase in features or size of the OSS project.

III. METHODOLOGY

VULNERABILITY: The vulnerability is weakness or security flaws in the code, which may results in exploited your product to the attacker or for unauthorized access. It is always good to find vulnerabilities from the source code at an early stage. Otherwise, if it remains undetected it may cause severe problems like degrading the quality of software, security [9], software management issues and affecting the evolution of software. The vulnerability may exist for both static as well as dynamic code. But, our research is only limited to static code analysis of C [10] and C++ [11] projects. There are various kinds of vulnerabilities that might exist in the source code of C/C++ programs. Few of them are as follows:

(i) Stack-based Buffer Overflows Vulnerability: This type of vulnerability occurs when large sized data are assigned or copied to fixed length buffers that are situated on the program without any consideration about bounds checking. This vulnerability is known to be of high risk, as their utilization would mostly permit alternate code execution or Denial of Service. It allows the attacker to take direct control over instruction pointer and may execute their arbitrary code or may alter the normal flow of the program. Consider a scenario below:

```
#include<stdio.h>
int main(int argc, char *argv[])
{
 char buff[20];    // Declare a String of size 20.
 printf("copying into buffer");
 strcpy(buff,argv[1]); /* Take command line argument and put into fixed sized buff string.*/
return 0;
}
```

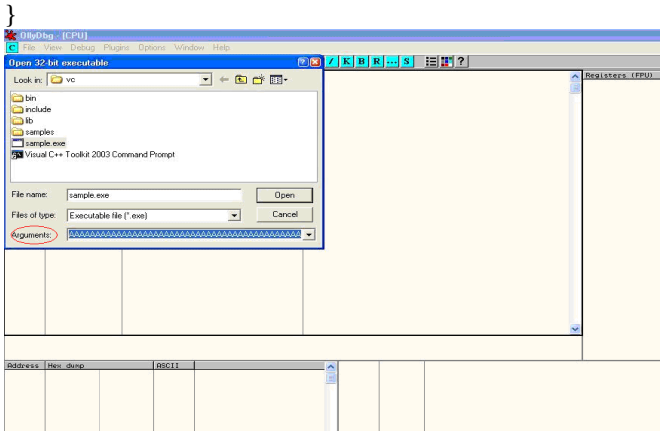

Figure 1. OllyDbg debugger screenshot [12,13]

Since the program is taking command line arguments, a large sequence of characters such as 'A', can be supplied in the argument field as shown in figure 1. While opening this executable file with the supplied arguments and continuing execution the following results are obtained.

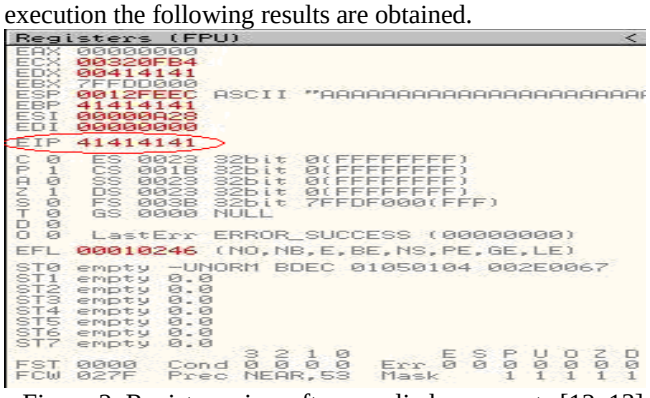

Figure 2. Registers view after supplied arguments [12, 13]

As shown in figure 2 the debugger register window, the Extended Instruction Pointer(EIP) that used to points to the next instruction to be executed, contains the value '41414141'. '41' is a hexadecimal representation of 'A' and so the string 'AAAA' gets translated to 41414141. This clearly signifies how the data which is being inputted by the user can be used to overwrite the instruction pointer with values inputted by the user and thus can control over the execution of the program. The functions like strcat(), strcpy(), gets() and many more that do not check the length of source strings and copy data blindly into fixed length buffers must be reviewed.

(ii) Heap-based buffer Overflows Vulnerability: Dynamic allocated data and global variables are stored using heap memory by providing them dynamic space. Each chunk of memory in heap comprises of boundary tags that contain memory management information. Just like the case with stack-based arrays, arrays on the heap can be over flown as well. Heap use to grow upwards in memory and stack grows in descending way. In any case, no return address (from where it is being called or from where it should proceed with its execution after the stack task) is put away onto store so attacker utilized distinctive strategy to pick up authority over the system. One technique for exploiting a buffer overflow situated on the heap is by overwriting heap-stored function pointers that are situated after the buffer that is being over flown. Function pointers are not constantly accessible however, so different methods for misusing heap-based overflows are by overwriting a heap-allocated object's virtual function pointer and directing it toward an attacker created virtual function table. At the point when the application endeavors to execute one of these virtual techniques, it will execute the code to which the attacker-controlled pointer refers.

Dynamic memory allocators: Overwriting the memory management information which is generally associated with a dynamically allocated block is more general way of attempting to exploit a heap-based overflow rather than using function pointers or virtual function pointers as they are not always available when an attacker encounters a heap-based buffer overflow The 'dlmalloc' (dynamic memory allocator) library is a run-time memory allocator that isolates the heap memory available to its into contiguous chunks, that change size as the various allocation and free routines are called. An invariant is that a free chunk never borders another free chunk when one of these routines has finished: if two free lumps had bordered, they would have been consolidated into a bigger free chunk. These free chunks are kept in a doubly linked list of free chunks, arranged by size. At the point when the memory allocator at a later time asks for a chunk of the same size from one of these free chunks, the first chunk in the list will be expelled from the list and will be accessible for use in the program (for example it will transform into an allotted chunk).

(iii) Dangling pointer references: A pointer to a memory area could allude to a memory area that has been de-allocated either unequivocally by the developer (for example by calling free) or by code produced by the compiler (for example a function epilogue, where the stack frame of the function is expelled from the stack). Dereferencing of this pointer is commonly unchecked in a C compiler, causing the dangling pointer reference to wind up an issue. In ordinary cases, this would make the program crash or show uncontrolled conduct as the value could have been changed at wherever in the program. In any case, double free vulnerabilities are a particular variant of the dangling pointer reference issue that could prompt misuse. A double free vulnerability occurs when previously free memory is de-allocated a second time. This could again enable an assailant to overwrite discretionary memory areas.

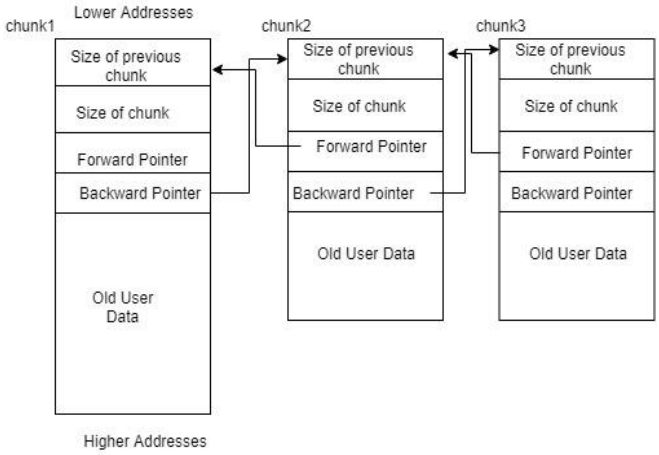

Figure 3. List of free chunks [14]

Figure 3 depicts what the list of free chunks of memory will look like when using the dlmalloc memory allocator. Chunk1 is bigger than the chunk2 and chunk3, meaning that chunk2 is the first chunk in the list of free chunks of its size. When a new chunk of the same size as chunk2 is freed it is placed at the beginning of this list of chunks of the same size by modifying the backward pointer of chunk1 and the forward pointer of chunk2. When a chunk is freed twice it will overwrite the forward and backward pointers and could allow an attacker to overwrite arbitrary memory locations at some later point in the program.

(iv) Format String Vulnerability: This vulnerability specifies how to test for format string attacks. These attacks can be used to crash a program or to execute harmful code [15]. The problem stems from the use of unfiltered user input as the format string parameter in certain C functions that perform formatting, such as printf(). The various C-Style languages provision formatting of output by means of functions like printf( ), fprintf( ) etc. The formatting is governed by a parameter to these functions termed as a format type specifier, typically %s, %c etc. The vulnerability arises

when format functions are called with inadequate parameters validation and user controlled data.

A simple example would be printf(argv[1]). In this case, the type specifier has not been explicitly declared, allowing a user to pass characters such as %s, %n, %x to the application by means of command line argument (argv[1]). This situation tends to become precarious since a user who can supply format specifiers can perform the following malicious actions: Enumerate Process Stack: This allows an adversary to view stack organization of the vulnerable process by supplying format strings, such as %x(print the hex value of an integer) or %p, which can lead to leakage of sensitive information. It can also be used to extract canary values when the application is protected with a stack protection mechanism. Coupled with a stack overflow, this information can be used to bypass the stack protector.

Control Execution Flow: This vulnerability can also facilitate arbitrary code execution since it allows writing 4 bytes of data to an address supplied by the adversary. The specifier %n comes handy for overwriting various function pointers in memory with the address of the malicious payload. When these overwritten function pointers get called, execution passes to the malicious code.

Denial of Service: If the adversary is not in a position to supply malicious code for execution, the vulnerable application can be crashed by supplying a sequence of %x followed by %n. Format string vulnerabilities manifest mainly in web servers, application servers, or web applications utilizing C/C++ based code or CGI scripts written in C. In most of these cases, an error reporting or logging function like syslog ( ) has been called insecurely.
When testing CGI scripts for the format string vulnerabilities, the input parameters can be manipulated to include %x or %n type specifiers. For example, the legitimate request is like:
http://hostname/cgi-bin/query.cgi?name=john&code=45765
can be altered to
http://hostname/cgi-bin/query.cgi?name=john%x.%x.%x&code=45765%x.%x

If a format string vulnerability exists in the routine processing this request, the tester will be able to see stack data being printed out to the browser.

If code is unavailable, the process of reviewing assembly fragments (also known as reverse engineering binaries) would yield substantial information about format string bugs. Take the instance of code below:

```
int main(int argc, char **argv)
{
printf("The string entered is\n");
printf("%s",argv[1]);
return 0;}
```

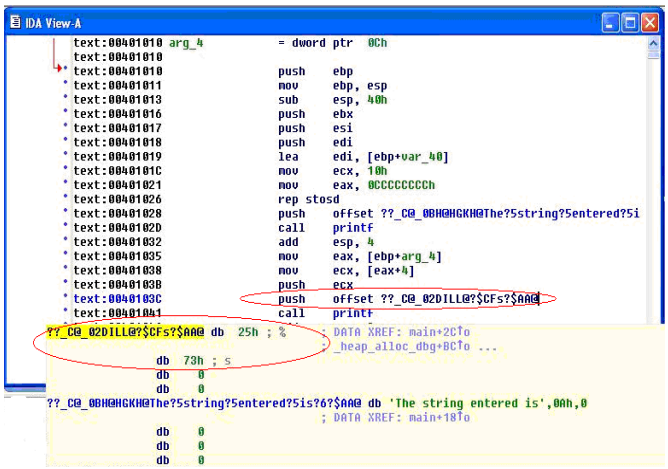

Figure 4. IDA view of Stack[12,16]

When the disassembly is examined using IDA Pro[16] in Figure 4, the address of a format type specifier being pushed on the stack is clearly visible before a call to printf is made. On the other hand, when the same code is compiled without "%s" as an argument, the variation in assembly is apparent. As seen below in figure 5, there is no offset being pushed on the stack before calling printf.

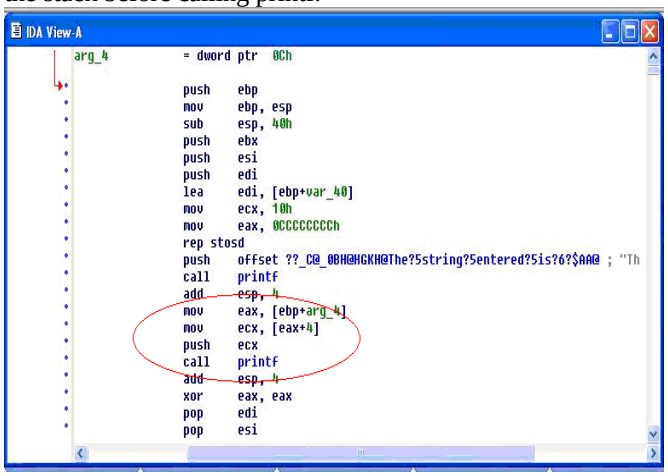

Figure 5. IDA view without using %s as an argument [12,16]

The functions that are primarily responsible for format string vulnerabilities are ones that treat format specifiers as optional. Therefore when manually reviewing code, emphasis can be given to functions such as: printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf. There can be several formatting functions that are specific to the development platform. These functions should also be reviewed for the absence of format strings once their argument usage has been understood.

(v) Integer Errors: Integer errors [17] are not exploitable vulnerabilities by themselves, but the exploitation of these errors could lead to a situation where the program becomes vulnerable to one of the previously described vulnerabilities.

Integer overflows and integer signedness errors are two kinds of integer errors that lead to exploitable vulnerabilities. An integer overflow occurs when an integer grows larger than its capacity or the maximum value it can hold. Integer signedness errors on the other side are minuter: A simply defined integer is assumed to be signed integer unless explicitly declared unsigned. When the programmer later passes this integer as an argument to a function expecting an unsigned value, an implicit cast will occur. This can lead to a situation where a negative argument passes a maximum size test but is used as a large unsigned value afterward, possibly causing a buffer or heap overflow if used in conjunction with a copy operation (e.g. memcpy3 expects an unsigned integer as size argument and when passed a negative signed integer, it will assume this is a large unsigned value).

IV. TOOLS FOR VULNERABILITY SCANNING

There are numerous static code analyzers [18] available over the web for e.g ITS4[19], Flawinder[20], VisualCodeGrepper [21] etc. The all have some significance to find security issues in the source code. We have chosen an open source tool named as 'Flawfinder'. The code for this tool is written in python and is executed over python 2.7 or higher. This tool is free to use and easily available over the web. The main reason for using this tool is its simplicity as well as its performance to find out the vulnerable code and then categorize them into various levels based on the severity of their risk. This tool works as a lexical analyzer and finds out the functions or methods that might be exploited for risk.

The vulnerability scanning tool Flawfinder works on some well-known problems, such as buffer overflow which is mostly due to use of functions like strcpy(), gets(), strcat(), scanf() and sprint() family. The format string problems, race conditions, potential shell metacharacter dangers, and many other problems are also gets checked by this tool. This tool works by using built-in database of C and C++ functions. This tool takes the source code text and matches the text with function names mentioned above. The tool ignores all the text which is inside the strings or in comments. Flawfinder also uses the gettext to pass the constant strings and thus reduce the number of false hits. This tool produces the result as a list of "hits"(potential security flaws), sorted by risk type. Hits mean here the number of times security flaw has been found, it works as counter here, which counts the number of flaws in sorted order. The level of the risk not only depends on the function but also on the values of the parameters in that function. Here examples of strings can be taken where constant strings are often less risky than fully variable string in many contexts. The only drawback of this tool is that it does not understand the semantics of the code, it just do simple text pattern matching. But still it is a very helpful in finding vulnerability and help developer to remove the vulnerabilities.

V. RESULTS AND DISCUSSION

The development repositories of various versions (see Table 1) of OSS project (MySQL) are obtained from GIT Hub [22]. A repository is downloaded by making the clone of the original repository onto the local machine by using GIT Bash. The vulnerability scanner tool (FlawFinder) is used to load all the source code files of the OSS project. Finally, the various security flaws (Hits or vulnerabilities) are obtained by analyzing the source code of the OSS project. We have considered the different versions of the MySQL project (see Table 1). The Flawfinder tool is utilized to inspect the source code of every one of these versions, to discover security flaws in the code. After getting the results, we have arranged them in order as indicated by their risk level as appeared in Table 1 (as 0, 1, 2, 3… … ,0+, 1+… ., 5+). The hits of type constant character, constant string, and the constant maximum length in the source are placed in level 1 of the hits list classification as they are not of high risk or rarely cause the issue in the execution process of any software. However, the issues like format string parameter, unchecked buffer overflow while concatenation to the destination, confound in the real and formal parameter's format(data types) are viewed as of great threat, so they are placed in a more elevated amount of risk category(i.e. level4 and level 5). Level 2 and 3 are viewed as moderate sort of risks. The security flaws of type: unchecked buffer overflow while copying a value from source to destination, or functions which do not check buffer overflow condition or no protection against internal buffer overflow are put in these classes. In our analysis, there are a lot of hits are experienced in every release of the product, which points to the presence of the vulnerability in the code of OSS (see Table 1). Level 1 of hits gradually increases over every version released; whereas level 2 increases at fast rate because of ignorance of coding guidelines. Level 4 and 5 are of higher risk, the increment in the hits value mirrors a risk to the software.

In Table 2(see below), we analyzed the different parameters (Total lines in code, Physical SLOC, Time Duration, Lines scanned every second and Hits/KSLOC) of the OSS. Hits/KSLOC figures the normal number of lines after which the hit of a specific type will happen. Lines analyzed connotes the total number of lines present in one version of the OSS which includes the real source code, preprocess directives, constant definitions and comments, though the values in the physical source lines of code represent the real lines of code excluding comments or other documentation. Time span speaks to the time taken by the scanner to examine the OSS in addition to lines scanned every second. From Table2 we collected values for the total number of hits and the total number of physical lines present in every version of the chosen OSS. These two parameters will additionally be utilized in the depiction of our research aim.

Table 1: Hits Classification into different levels

.	Hits@level										
version	***1***	***2***	***3***	***4***	***5***	***0+***	***1+***	***2+***	***3+***	***4+***	***5+***
mysql-server-8.0.0	3659	9686	360	1337	81	15118	15118	11646	1778	1418	81
mysql-server-8.0.1	3729	9571	359	1316	86	15061	15061	11332	1761	1402	86
mysql-server-8.0.2	3888	9950	369	1370	86	15663	15663	*11775*	1825	1456	86
mysql-server-8.0.3	3922	10117	304	1404	86	15914	15914	11992	1875	1490	86
mysql-server-8.0.4	4066	10704	417	1457	89	16733	16733	12667	1963	1546	89
mysql-server-8.0.11	3857	10697	402	1691	84	16731	16731	12874	2177	1775	84
mysql-server-8.0.12	3765	10608	401	1694	84	16552	16552	12787	2179	1778	84
mysql-server-8.0.13	3956	10730	439	1762	103	16990	16990	13034	2304	1865	103

Table 2**:** Various Statistics of the code

	Hits/KSLOC										
version	***0+***	***1+***	***2+***	***3+***	***4+***	***5+***	***Hits***	***Lines analyzed***	***Physical source lines of code(SLOC)***	***Time Duration (seconds)***	***Lines per Second***
8.0.0	7.00048	7.00048	5.30847	0.823314	0.656613	0.037537	15118	3083327	2159566	289.05	10667
8.0.1	5.88695	5.88695	4.42938	0.688329	0.548005	0.033615	15061	3529024	2558370	334.58	10548
8.0.2	5.96379	5.96379	4.48341	0.694881	0.554382	0.032745	15663	3639266	2626348	367.27	9909
8.0.3	5.95243	5.95243	4.48546	0.70132	0.557316	0.032167	15914	3718015	2673530	442.23	8407
8.0.4	5.54407	5.54407	4.1969	0.650392	0.51223	0.029488	16733	4339546	3018178	423.30	10252
8.0.11	5.98554	5.98554	4.6057	0.778825	0.635009	0.030051	16731	4092483	2795235	477.34	8574
8.0.12	5.98226	5.98226	4.6215	0.787538	0.642608	0.030359	16552	4054457	2766849	438.75	9241
8.0.13	5.81248	5.81248	4.45908	0.788225	0.638038	0.035237	16990	4308496	2923023	424.41	10152

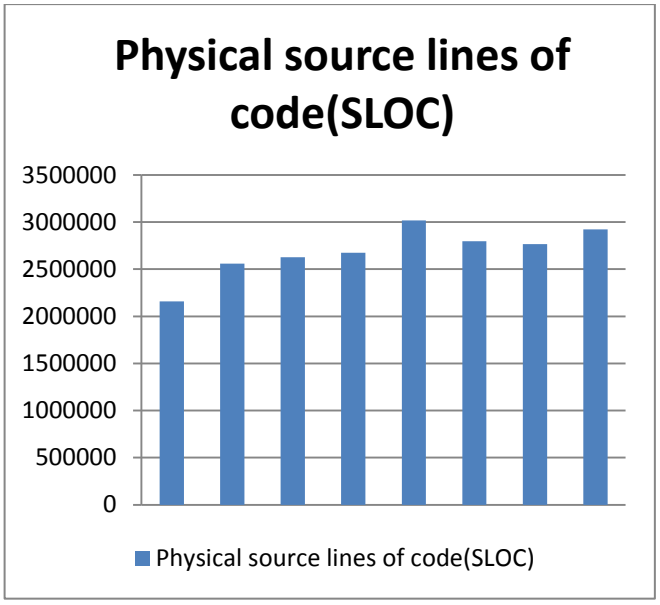

Fig 6.1: Physical source lines of code

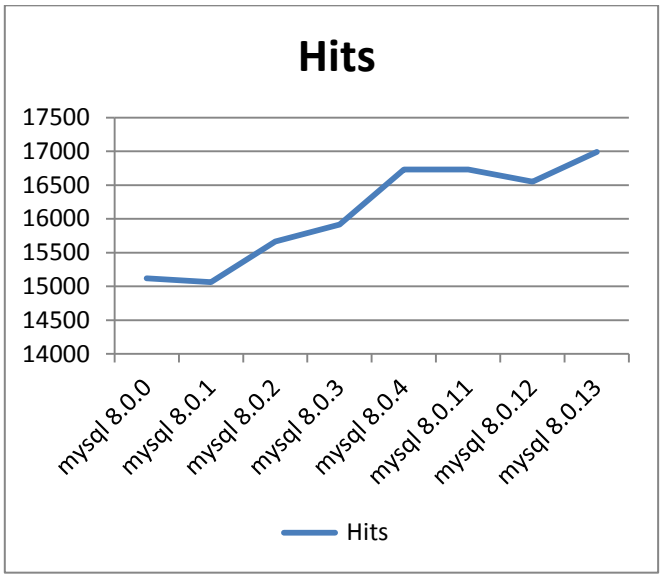

Fig 6.2: Total hits present in each version

Fig 6.1 represents the number of physical lines present in each version of the code. The total physical source lines of code vary from 21 million to 30 million. Figure 6.2 represents the graphical view of the number of hits in all the software releases. As we can see the size of the code gets increased with each release of the software because of the new features get added with each release so as the number of hits.

The Table3 below represents the regression equation, from which we can conclude the rate at which the hits of each level will get increased in the next release of the product. The regression analysis is utilized to discover the trend and equation of the regression for hits, (level insightful) in the ongoing releases of the OSS. In Table3 we demonstrate the outcomes after the regression analysis. From the regression equation of each level, we have observed that all the regression have a positive slope, which implies the increasing trends in all hit level.

Table 3: Trend analysis of hit level for all the version of MySQL

Level of Hits	Regression Equation	Trend
Level 1	y = 27.5x + 3731.5	Increasing
Level 2	y = 182.39x + 9437.1	Increasing
Level 3	y = 11.607x + 329.14	Increasing
Level 4	y = 70.012x + 1188.8	Increasing
Level 5	y = 1.6786x + 79.821	Increasing

We also process the relationship by utilizing Karl Pearson technique for correlation between the physical source lines of code and all the hits experienced in all previously mentioned version of the OSS. The estimation of R (Karl Pearson Coefficient unit) is 0.8608. The value is nearer to the 1, which connotes that there is a strong positive relationship. This outcome suggests that higher the source lines of code there are equivalent odds of getting higher number of hits.

VI. CONCLUSION AND FUTURE SCOPE

Quality software must be sound in both: its purpose and its security. Vulnerabilities present in the code have an adverse impact on the quality of the software so these should be removed. Here in this research work, we took a well-known OSS and check it against a series of vulnerabilities and found out that there are numbers of flaws present in the code. Our main aim in this paper is to found the vulnerability in the source code and afterward notice the relationship between the size of the software and the number of flaws. The results show that there is a strong relationship present between the size of the software and number of security flaws. So, the main job for developers of the software should not be just to adding new features to the existing product but also to improve the quality of the software by writing code which should be secure against vulnerabilities, which are mentioned in this paper. Our future work will be one stage ahead starting here, which is to apply a similar technique for vulnerability scanning on all sizes of the OSS and will endeavor to establish out that the relationship between the size of product and number of hits will stay same for all sort of sizes of OSS

or not, and furthermore we need to add some commitment to expel the vulnerable code without influencing the actual working of the code. As we have mentioned earlier a vulnerable code will always remain a threat for the developer. There is a lot of work had already done around there, yet the vast majority of the work had been performed on the web application or dynamic code. Our point will be to decrease workload of the dynamic scanner by making the code vulnerability free at the static end.

REFERENCES

[1] Younan, Y., W. Joosen, and F. Piessens. "Code Injection in C and C++: A Survey of Vulnerabilities and Countermeasures (Tech. Rep. No. CW 386)." *Leuven, Belgium: Departement Computerwetenschappen, Katholieke Universiteit Leuven* (2004)

[2] Piessens, Frank. "A taxonomy of causes of software vulnerabilities in internet software." Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering. 2002.

[3] "Glossary." *Risk Management & Information Security Management Systems - ENISA*, 20 Jan. 2016, www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/glossary#G52.

[4] Abbott, Robert P., et al. Security analysis and enhancements of computer operating systems. No. NBSIR-76-1041. NATIONAL BUREAU OF STANDARDS WASHINGTONDC INST FOR COMPUTER SCIENCES AND TECHNOLOGY, 1976.

[5] Aslam, Taimur. "A taxonomy of security faults in the unix operating system." Master's thesis, Purdue University 199.5 (1995).

[6] Yamaguchi, Fabian, et al. "Chucky: Exposing missing checks in source code for vulnerability discovery." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[7] Ball, Thomas, et al. "Thorough static analysis of device drivers." ACM SIGOPS Operating Systems Review 40.4 (2006): 73-85.

[8] DeKok, Alan. "PScan: A limited problem scanner for C source files." (2013).

[9] Evans, David, and David Larochelle. "Improving security using extensible lightweight static analysis." IEEE software 1 (2002): 42-51.

[10] Kernighan, Brian W., and M. Dennis. "Ritchie. The C Programming Language." (1988).

[11] Stroustrup, Bjarne. The C++ programming language. Pearson Education India, 2000.

[12] HeapOverflow:https://www.owasp.org/index.php/Testing_for_Heap_Overflow,StackOverflow:https://www.owasp.org/index.php/Testing_for_Stack_Overflow,FormatString:https://www.owasp.org/index.php/Testing_for_Format_String .

[13] Conover, Matt. "w00w00 on heap overflows." (1999).

[14] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture, 2001. Order Nr 245470.

[15] scut. Exploiting format string vulnerabilities. http://www.team-teso.net/articles/formatstring/, 2001

[16] IDA PRO, https://www.hex-rays.com/products/ida/overview.html

[17] Brumley, David, et al. "RICH: Automatically protecting against integer-based vulnerabilities." Department of Electrical and Computing Engineering (2007): 28.

[18] Zitser, Misha, Richard Lippmann, and Tim Leek. "Testing static analysis tools using exploitable buffer overflows from open source code." ACM SIGSOFT Software Engineering Notes. Vol. 29. No. 6. ACM, 2004.

[19] Viega, John, et al. "ITS4: A static vulnerability scanner for C and C++ code." Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference. IEEE, 2000.

[20] Flawfindetr: https://dwheeler.com/flawfinder/flawfinder.pdf and A book entitled as "Secure Programming HOWTO" by David A. Wheeler.

[21] Fatima, Anum, Shazia Bibi, and Rida Hanif. "Comparative study on static code analysis tools for C/C++." Applied Sciences and Technology (IBCAST), 2018 15th International Bhurban Conference on. IEEE, 2018.

[22] GIT HUB, https://github.com/mysql/mysql-server.

Authors Profile

Madanjit Singh completed Bachelor of Computer Applications from Punjab Technical University, Jalandhar, Punjab, India in 2012 and Master of Computer Applications from Guru Nanak Dev University, Amritsar, Punjab, India in year 2015. He is currently working as Assistant Professor in Department of Computer Science, Guru Nanak Dev University, Amritsar, Punjab, India since 2015. His main research work focuses on Software Security, Open Source Software and Natural Language Processing. He has 3 years of teaching experience.

Munish Saini works as an Assistant Professor in Department of Computer Engineering and Technology, Guru Nanak Dev University, Amritsar. He had completed his P.hd in 2018. His main research work focuses on Software Engineering, Open Source Software. He has 2 years of teaching experience and 4 years of Research Experience.

Manevpreet kaur completed her Master of Computer Applications from Guru Nanak Dev University, Amritsar, Punjab, India in year 2015. She is currently working as Assistant Professor in Department of Computer Science, Guru Nanak Dev University, Amritsar, Punjab, India since 2015. Her main research work focuses on Software Security and Open Source Software. He has 3 years of teaching experience.

