AUTOMATION FRAMEWORK FOR SOFTWARE VULNERABILITY EXPLOITABILITY ASSESSMENT

Bagri
Information Technology
IGDTUW
Delhi, India

Gupta
Information Technology
IGDTUW
Delhi, India.

***Abstract*—Software has become an integral part of every industry and organization. Due to improvement in technology and lack of expertise in coding techniques, software vulnerabilities are increasing day-by-day in the software development sector. The time gap between the identification of the vulnerabilities and their automated exploit attack is decreasing. This gives rise to the need for detection and prevention of security risks and development of secure software. Earlier the security risk is identified and corrected the better it is. Developers needs a framework which can report the security flaws in their system and reduce the chances of exploitation of these flaws by some malicious user. Common Vector Scoring System (CVSS) is a De facto metrics system used to assess the exploitability of vulnerabilities. CVSS exploitability measures use subjective values based on the views of experts. It considers mainly two factors, Access Vector(AV) and Authentication (AU). CVSS does not specify on what basis the third-factor Access Complexity (AC) is measured, whether or not it considers software properties. Our objective is to come up with a framework that automates the process of identifying vulnerabilities using software structural properties. These properties could be attack entry points, vulnerability locations, presence of dangerous system calls, and reachability analysis. This framework has been tested on two open source softwares - Apache HTTP server and Mozilla Firefox.**

***Index Terms*—Structural Severity, Vulnerabilities, Reachability, Entry points**

I. INTRODUCTION

SOFTWARE vulnerability is a defect in the software construction that can be exploited by an attacker to get some privileges in the system. The least damaging software vulnerability is the one that can never be exploited as no software is ideally vulnerability proof. The earlier the software vulnerabilities are detected the better it is. It is necessary to audit the softwares for defects and remove them before attackers discover and exploit them. Open source softwares have an upper hand in this case as they let anyone to audit the source code and provide any enhancements or report the bugs in it. Attackers can break into any system, provided they have enough time, knowledge and resources. No security technology or procedure can guarantee the safety of a system from intrusion. Many probable attacks can be avoided from happening by adopting proper coding practices and ensuring less number of bugs. Most security consultants agree upon the standard security model called as CIA, or Confidentiality, Integrity, and Availability. Vulnerability assessment is nothing more than an internal audit of the system and network security; resulting in the evaluation of CIA security standards. Vulnerability assessment is a process that involves sequential steps to be followed. If vulnerability assessment of a house is done, each door and window of the house is checked if they are closed and secured. Similarly, software systems are scanned for entry points. First, we gather the relevant information about the target system and resources. This is known as reconnaissance phase. Second, we look for the possible vulnerabilities in the system. Third, comes the reporting phase where the severity of each vulnerability is leveled with a low(L), medium(M), or high(H) grade on the basis of initial results.

A. Problem Description

There exist a lot of vulnerabilities that are not reported publicly. We need a measure to detect the vulnerabilities that remain unreported so that

they can be handled before they are exploited. A framework that systematically checks the source code of the software and points to the location of the flaw in code so that timely steps can be taken to rectify it. There exist many challenges to assess the software vulnerabilities. The structural properties of the source code can reveal a lot about the potential security defects in the software.

1) Vulnerable Function Location : Any normal software contains n number function definition and functional calls. Functions are the basic building blocks of the any software. Here, we have considered vulnerabilities at functional level. One of the major challenges in vulnerability detection is to detect the location of the vulnerable function. It may happen that a function defined in one file may be used in some other file or a function is defined multiple times in different files. What metrics should be considered to mark a location in the source code as vulnerable?

2) Exploitibility of a vulnerability : Not all the detected vulnerabilities are exploitable. There exist many flaws in the source code but the probability of that defect to be exploited by any attacker depends on many factors. It depends upon a number of factors. To find the metrics that classifies the exploitibility of these vulnerabilities with low false positive rate and to derive these metrics objectively from the source code is a challenge.

3) Impact Level: Not all exploitable vulnerabilities have the same impact. A defect in the GUI of a software will have less impact then a defect in the main business functionality of a software. What factors can be used to decide the impact level of a vulnerability in the source code? Existence of dangerous system call can be used to decide the impact level as higher the privilege an attacker has the more damage it can cause. DSCs can be used by an attacker to increase the privilege it enjoys.

B. Research Objectives

The main objective of this research is to address the above mentioned problems with less human intervention. We mainly focus on automating the way to tackle these problems. Rather viewing and mapping the publicly reported vulnerabilities we try automate this process. The metrics used to detect the exploitability of a vulnerability can also be derived from the source code objectively. The metrics used for this purpose is attack surface metrics. For any attacker to attack a vulnerability in the software, it requires an entry point and if an entry point exists, it is important to have the knowledge of whether it is connected to the vulnerable function that is under consideration. To evaluate the impact of exploit, we consider the existence of dangerous system calls in the vulnerable function. Dangerous system call can be used by the attacker to escalate its privileges and hence can cause a greater impact. This research is based and performed on the software written in C language only.

The software selected as our case study to implement the stated solution are - Apache HTTP server and Mozilla Firefox web browser. The reason to select these software is the availability of source code, code diversity, size of the software, large publicly reported vulnerabilities database and bug tracking.

II. METHODOLOGY

Our aim is to build an automated framework that can help in assessment of vulnerabilities using software structural properties. The approach for this purpose is divided in major four steps. First, to detect vulnerable location. Second, to find entry points in source code. Third, to check the reachability of the vulnerable location from the entry point and lastly, checking the existence of any DSCs in the entry point function. The outcome of these steps is vulnerabilities classified as - reachable (R) from an attack entry point with dangerous system call(DSC), reachable(R) from attack entry point without dangerous system call (NDSC) or not reachable(NR). This study of Software Vulnerability Assessment focuses on reducing subjectivity in assessing vulnerability risk. We have tried to reduce manual effort as much as possible using python scripts and tools. Figure 1 shows an overview of our approach for assessing vulnerability exploitability risk.

A. Identify Vulnerable location

There can be many different ways to do this step - Using prediction model based on software metrics or by looking at the report in the vulnerability database such as NVD or using static analysis tool. According to Shin et al. [2013] using software metrics based predictions model resulted in a precision value of only 11%. Mapping of the vulnerable

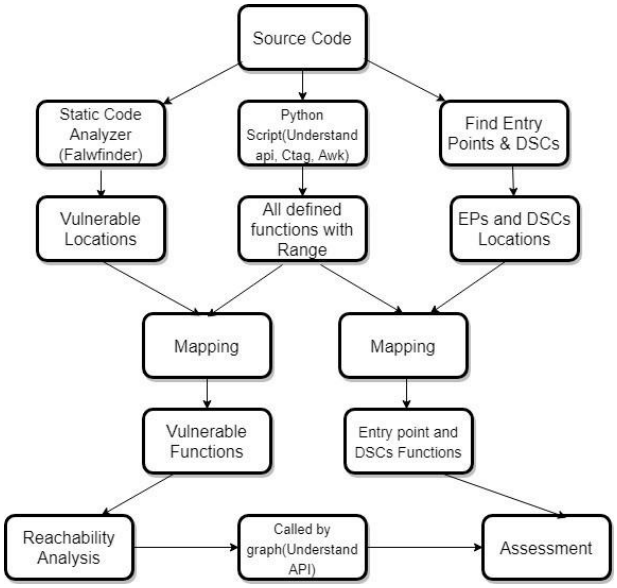

Fig. 1: Overview of the approach

location using vulnerability database as done by Younis et al. [2015] takes a lot of human efforts and time therefore it is not an efficient approach. Using static code analyzer was a convincing approach as it solves both the problems. Although, false positive rate of static analyzers is a problem.

Static code analysis is a technique to find possible bugs in the source code of a software without executing it. We are automating this process using FlawFinder [13] and a Python script. The reason for using static analysis tool is that it will list out all the reported and unreported vulnerabilities in the software. Since, many software vulnerabilities are not reported in the publicly available databases like NVD [10], EDB, etc. static code analysis will let the user find the unreported vulnerabilities as well. FlawFinder uses CWE [11] database to locate vulnerabilities in the code. It is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts. CWE has defined strategic classes of vulnerabilities. A class is a CWE entry that contains a set of other entries that share a common characteristic. Within classes there exist more specific base level weaknesses with sufficient description for detection and prevention.

Flawfinder [13] is an officially CWE compatible tool. Flawfinder works by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks (e.g., strcpy(), strcat(), gets()), format string problems ([v][f]printf(), [v]snprintf()), race conditions (such as access(), chown(), chgrp(), chmod(), tmpfile()), potential shell meta character dangers (most of the exec() family, system(), popen()), and poor random number acquisition (such as random()). Python script is used find the function defined in the source code inside which the vulnerable C/C++ functions as discussed above are called. Following steps are performed to identify the vulnerable functions.

- Obtain the source code of an open source software
- Using static analysis tool (FlawFinder), obtain all the software security weaknesses in CSV format.
- Data preprocessing using Python
- Find the calling functions of the vulnerable function using a python script.

The outcome of the above steps is a list of vulnerable locations along with their details of file path, line number, CWE ID, category and calling function.

B. Find Entry Points

Entry point function are the C/C++ library functions used to send input to the software environment. They are the resources used the attacker to get inside system. In this study we have only considered entry points as the main target of the malicious users. The entry points considered here are the one proposed by Manadhata and Wing [2011]. An entry point may be direct or indirect. Direct entry point directly calls the vulnerable function and indirect entry point calls another function which calls the vulnerable function.

We have used a python script to scan the source code for potential entry points for example - read, get, getline etc using a dictionary of the attack entry point functions. The dictionary contains all the well-known C/C++ input library function. Next, we find the calling functions using a python script as done previously in finding vulnerable locations. Following are the steps followed to find the entry points from the source code of the software.

- Using a dictionary of potential attack entry points (C/C++ library functions), find the list of all possible entry points in the source code.
- Find the calling functions of the entry point functions using a python script.

The result of this step is the list of entry point locations along with their details of file path, line number, and function name.

C. Dangerous system call

Dangerous system calls are the calls to the functions which may be used by the attacker for escalating its privileges of the compromised system. We have used dangerous system calls as an impact metrics of any vulnerability. These system calls have been identified and classified into four levels of threats. Level one allows full control of the system, while level two is used for denial-of-service attack. On the other hand, level three is used for disrupting the invoking process and level four is considered harmless. There are 22 system calls of the threat level one and 32 of the threat level two. This classification of the DSCs is proposed by Massimo et al. [2002] . We have used a python script to find all the dangerous system calls in the source code using a dictionary of the DSCs and then classified them into the above mentioned four classes.

- Using dictionary of potential DSCs, find the list of all possible DSCs used in the source code.

Outcome of this step is a list of DSC locations along with their details of file path, line number, level, and function name.

D. Reachability analysis

Once all the vulnerable functions and entry points are identified, the relationship between them is found using a called-by graph. This graph is similar to a dependency graph which captures all functions that call the vulnerable function directly or indirectly. This is done using Understand tool[13] Python API. For each function in the called-by verify whether it is an entry point or not. If yes then the vulnerable function is reachable else not-reachable. Also we simultaneously check the existence of any Dangerous System Calls in the entry point functions. Following steps are performed to do the reachability analysis.

- For each vulnerable function, generate a list of Called-By functions which we usually see in the dependency graphs.
- For every Called-By function, verify whether it is reachable through an entry point or not.
- For the entry point functions, verify whether it contains any Dangerous System Call or not.

The outcome of this step is the classifying vulnerabilities as reachable or not reachable from an attack entry point.

E. Assessment

It is considered that if a vulnerability is not reachable, then it has Low severity despite of the presence or absence of any DSCs. But if it is reachable, then it has Medium severity in the absence of a DSC and High exploitability risk in the presence of a DSC. After following all the above steps, we will have our desired dataset. Using this dataset, we can compare the outcome using publicly available vulnerability database (NVD) and generate a performance report. The vulnerabilities in the dataset will be classified as one of the following:

- Reachable with Dangerous System Calls (High)
- Reachable with No Dangerous System Calls (Medium)
- Not reachable (Low)

III. Evaluations and Results

This section presents the results of each step of the methodology. For assessing software vulnerability exploitability is based on the steps that have been discussed in the previous section. All the steps have been followed for both Apache HTTP server and Mozilla Firefox web server but we are only including the detailed results of Apache HTTP server.

A. Identify Vulnerable location

The vulnerability location can be found by looking at the report in the vulnerability database or by using a static code analyzer such as Splint or FlawFinder. Finding vulnerability location by looking at the reports in the vulnerability database was implemented by Younis et al [2015]. But finding vulnerabilities by mapping reports from vulnerability database poses several problems. Firstly, it involves a lot of manual work to identify each vulnerability, finding its CVE-ID and then looking in the database to locate that vulnerability. Secondly, there exist many software that do not maintain their vulnerability reports on the public platform for the users. Thirdly, considering only the publicly reported vulnerabilities may leave many unreported

TABLE I: Flawfinder Result Summary

CWE Category	Count of File
access	3
buffer	1260
crypto	4
format	29
integer	247
misc	17
race	6
random	3
shell	7
Grand Total	1576

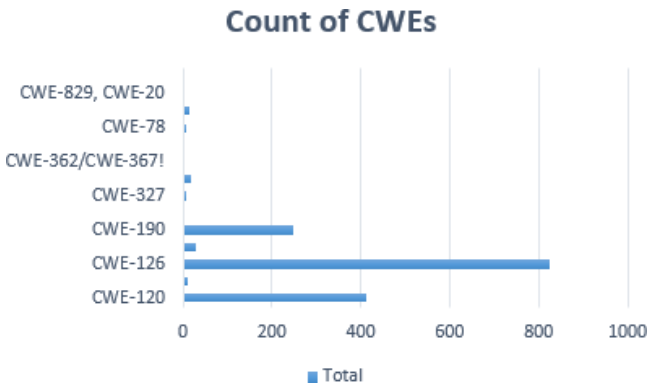

Fig. 2: Distribution of Vulnerabilities in Apache HTTP server according to CWE

vulnerabilities unnoticed. Our main purpose to undergo this study is to build a framework using which can be used during the development phase of any software to detect the week points in the code and take actions to remove those vulnerabilities according to their priority. We have used a static code analyzer Flawfinder to detect the vulnerable locations from the source code. Flawfinder uses CWE database to the find the vulnerable location the code. Flawfinder reported 1576 vulnerabilities in Apache HTTP server. Using a python script we find the functions defined in the source code inside which these vulnerabilities exists.We have made use of the Understand tool python API and Linux awk command to find the calling functions. Table 1 shows a summary of the results of static code analyzer with the category of the vulnerability and total number of the files in the source code having that kind of vulnerability. Figure 2 shows the distribution of vulnerabilities according to the CWE number. CWE-126 which belong to the buffer category is present in maximum number of files in the source code of Apache HTTP server.

B. Find attack entry points

Entry point is any library function used to get some input from external environment. To find these entry point function we used a python script that analyzes the source code and performs a keyword search for C/C++ input library functions such as read, gets, getline etc. After getting the attack entry points, we find the function defined in the code inside which that library function was actually called (i.e. the calling function). This is required later for mapping the entry points with the vulnerable functions for the reachability analysis part. For this we used a python script and Linux awk command. Table 2 shows the summary of this step with entry point

TABLE II: Summary of Entry Point Results

Entry point	Count of Files
// **Mapping with CWE ID in the table** read	707
send	187
socket	604
fgets	5
fopen	22
fputc	4
fread	4
freopen	1
fwrite	22
getc	55
gethostbyname	5
getline	61
gets	213
rename	49
scanf	25
setbuf	1
sscanf	23
Grand Total	1988

functions and the count of the files containing that attack entry points. Apache HTTP server contains in total 1988 files containing entry point functions and read has been used maximum number of times.

C. Reachability Analysis

Reachability analysis means the checking of the call relationships between the entry points and the vulnerability functions. We have made use of Understand tool python API to perform reachability analysis on all the mapped vulnerable functions. Figure 3 shows the graph image generated manually using the Understand tool GUI. In order to check whether the vulnerable function is reachable from

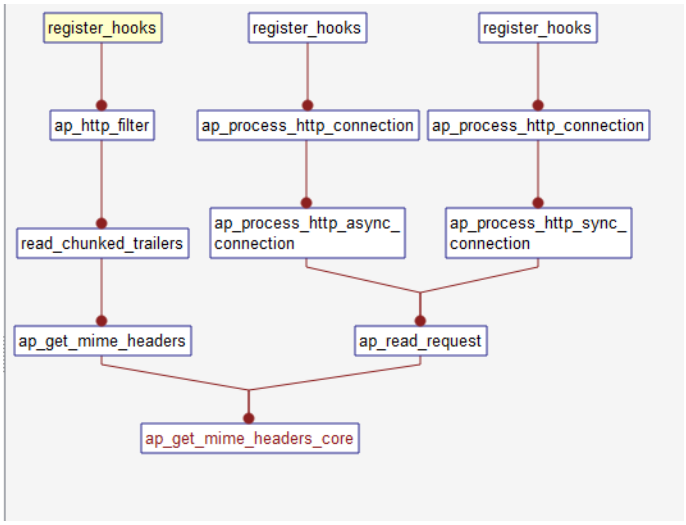

Fig. 3: Called-by graph generated by Understand tool for ap_get_mime_headers_core function

```
1  ap_get_mime_headers_core  server\protocol.c(927)
2  | table_do_fn_check_lengths  server\protocol.c(911)
3  | ap_get_mime_headers  server\protocol.c(1184)
4  | | read_chunked_trailers  modules\http\http_filters.c(250)
5  | | | ap_http_filter  modules\http\http_filters.c(297)
6  | ap_read_request  server\protocol.c(1248)
7  | | ap_process_http_async_connection  modules\http\http_core.c(135)
8  | | | ap_process_http_connection  modules\http\http_core.c(249)
9  | | ap_process_http_sync_connection  modules\http\http_core.c(183)
10 | | | ap_process_http_connection  modules\http\http_core.c(249)
11
```

Fig. 4: Textual Called-by graph generated using Understand Python API for ap_get_mime_headers_core function

TABLE III: Summary of threat level of DSCs

Threat Level	Count
Denial of Service	1223
Full System Control	238
NDSC	125
Grand Total	1586

an entry point, we had to verify whether any of the calling function (direct or indirect) has an entry point or not. The figure provided depicts just a small example. In figure 3 the called-by graph of function ap_get_mime_header_core contains an entry point function named register_hooks(). register_hooks is a function defined in the source code and it contains C/C++ library function such as read, gets etc. and it clear from the called-by graph that register_hooks is indirectly reachable from ap_get_mime_header_core. Hence, it is a vulnerable function reachable from an attack entry point. The called-by graph of this function is small but there are many functions with more than 10 functional invocations as well as levels. Hence, it is very time consuming to manually generate called-by graph of every vulnerable function and check the presence of entry points in it.

To generate these graphs automatically for all the vulnerable functions in a single run and simultaneously check the presence of entry points in it we used a python script which does all this efficiently in a single run and classifies each vulnerable function as reachable(R) or not-reachable(NR). This saves both time and efforts. Figure 4 is the textual format of the same graph in Figure 3 generated using python API of Understand tool. We made use of this textual format graph to check the reachability of the vulnerable functions.

D. Dangerous system calls

Existence of DSCs is used as an estimator of the impact of exploitation. DSCs are used by the attacker to enhance its privileges and create a larger impact. Entry points are the main target of the attacker so we check the presence of DSCs in entry point functions. A python script is employed to verify the presence of DSCs which uses a dictionary of predefined dangerous system calls used to escalate the privileges of the user such as getpid, chmod etc. The script also classifies the DSCs into one the four threat levels defined in table 5. If there is no DSC present in the file, then the output of threat level column is labeled as NDSC which means No Dangerous System Call. Table 3 shows the summary of the output of the python script.

E. Assessment of the vulnerabilities

The vulnerabilities are classified qualitatively on the basis of their exploitability risk as one of the following:

- High - Reachable with Dangerous System Calls
- Medium - Reachable with No Dangerous System Calls
- Low - Not reachable

The output also consists of metrics other than the specified ones. The callsCount metrics gives the count of number of function calls made by the vulnerable function directly or indirectly and the

FileName	CWE_ID	FuncName	callbyCount	Reachability	epCount	epDSC	epNoDSC	callsCount	ExploitRisk
modules/debugging/mod_dumpio.c	CWE-120	dumpit	4	YES	1	1	0	26	High
modules/mappers/mod_dir.c	CWE-190	configure_redirect	1	NO	0	0	0	3	Low
modules/mappers/mod_dir.c	CWE-126	fixup_dir	3	YES	1	1	0	28	High
modules/mappers/mod_imagemap.c	CWE-126	imap_url	4	YES	2	2	0	51	High
modules/mappers/mod_rewrite.c	CWE-807, CWE-20	lookup_variable	11	YES	3	3	0	126	High
modules/mappers/mod_negotiation.c	CWE-190	get_entry	15	YES	4	1	3	0	High
modules/mappers/mod_alias.c	CWE-190	add_redirect_internal	4	NO	0	0	0	46	Low
modules/mappers/mod_vhost_alias.c	CWE-126	mva_translate	2	YES	1	1	0	36	High
modules/mappers/mod_speling.c	CWE-126	check_speling	2	YES	1	1	0	68	High
modules/ssl/ssl_util_ssl.c	CWE-120	provide_pass	1	YES	1	0	1	1	Medium
modules/ssl/ssl_util_ssl.c	CWE-190	modssl_X509_getBC	4	YES	2	0	2	0	Medium

Fig. 5: Sample output with structural severity

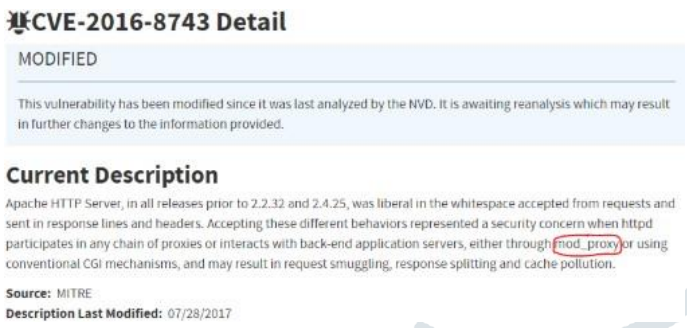

Fig. 6: NVD Description for CVE-2016-8743

callbyCount metrics gives the count of number of unique functions calling that function directly or indirectly. We will discuss it later in the results whether this metrics had any effect on the structural severity. The other metrics- epCount, epDSC, epNoDSC, and ExploitRisk tells the number of entry points, number of entry points having DSC, number of entry points having no DSC and the structural severity respectively. Figure 5 is a sample of the output we got after the final assessment. The complete result was quite large so we have included just the sample of it.

IV. Observations and Performance Evaluation

To compare our results of the automation framework which is a top-down alternative of the bottom-up approach proposed by Younis et al [2015] . Their results were based on the collection of publicly reported vulnerabilities from NVD [10] while our current result is based on the complete vulnerability information using CWE [12]. Note that the vulnerabilities that we collected using NVD are only those list of CVEs that are directly influenced by the user input. Since there are various types of vulnerabilities caused due to different reasons, it was essential to select a subset of those vulnerabilities which were desired for the comparison of our results.

Using this incomplete vulnerability information, we can identify that this vulnerability is located in the mod-proxy module in the source code of Apache HTTP server. We used this detail to map the vulnerabilities identified by our automated framework. Note that the complexity for this CVE-2016-8743 provided by the National Vulnerability Database is Low whereas in our results, we found a total of nine vulnerable functions in the same module, out of which only six have Low structural severity and the rest three have High structural severity.

It was observed that out of 50 vulnerabilities in Apache HTTP server 2.4.x, only 35 were directly influenced by the user input out of which we were able to match 32 CVEs based on their description of the vulnerable module. These 32 vulnerable modules covered 138 vulnerable functions present in it specifying the structural severity for each function. Apart from that, there were 487 other vulnerable functions identified by our approach in which 193 were of High structural severity.

To evaluate the performance, we have used the concept of confusion metrics. It is a good tool to analyze the performance of a binary classifier. Here, the files are classified as vulnerable and non-vulnerable. Total number of files in Apache HTTP source code were equal to 348 out of which 137 files were classified as vulnerable in our result and only 35 files were present in the reported vulnerabilities. The performance measures derived from confusion metrics are:

- **Recall** - percentage of vulnerable files detected in the software.
- **Precision** - the percentage of files which are vulnerable in the reported database as well as predicted to be vulnerable by the framework.
- **Accuracy** - the percentage of files that are correctly classified.

For Apache HTTP 2.4.x case study:

- Precision=(TP/(TP+FP)) = 17.4% [Note that we have used the static code analyzer to fetch the complete vulnerability details which include the unreported vulnerabilities as well. Since the comparison has been done with the publicly reported vulnerabilities only and there is not sufficient data for efficient comparison, hence it was expected to have high false positive rate and therefore, low precision value.]
- Recall=(TP/(TP+FN)) = 91.43%
- Accuracy=((TP+TN)/(TP+TN+FP+FN)) = 55.2%

TABLE IV: Confusion Matrix for Case Study: Apache HTTP 2.4.x

Actual vs Predicted	Vulnerable File	Non-Vulnerable File
Vulnerable File	32	4
Non-Vulnerable File	152	161

TABLE V: Confusion Matrix for Case Study: Mozilla Firefox 53.0.3

Actual vs Predicted	Vulnerable File	Non-Vulnerable File
Vulnerable File	39	0
Non-Vulnerable File	123	3403

We have also implemented the same approach in another open source software - Mozilla Firefox 53.0.3.

- Precision = (TP/(TP+FP)) = 24.1%
- Recall = (TP/(TP+FN)) = 100% [Mozilla Firefox has private vulnerability advisory and has provided more incomplete description than the publicly available reported vulnerabilities in NVD. There were total 39 CVEs in their advisory and total 162 distinct vulnerable files were found in our results. Due to the incomplete description of vulnerabilities, we have assumed that the actual vulnerable files have been predicted as vulnerable]
- Accuracy = ((TP+TN)/(TP+TN+FP+FN)) = 96.5%

V. Conclusion and Future Work

The goal to build an automation framework for all the steps of approach proposed by Younis et al [2015] to assess the structural severity of vulnerabilities has been achieved in this thesis. This will help in analyzing the results for other C software which do or do not have publicly available description of vulnerabilities. Moreover, this framework can be extended to calculate other structural metrics such as NodeRank proposed by Bhattacharya et al [2012] as an estimator of the vulnerability impact and penetration depth of a vulnerable location i.e. the minimum number of invocations required from an entry point with DSC to call the vulnerable location. Reachability is the major contributor for measuring the risk of exploitation of an vulnerability. However, this is not always true that a vulnerability that is reachable is exploitable. There are number of factors upon which the exploitability of a reachable vulnerability may depend for example the number function invokes the attacker has to make to reach the vulnerable location(depth) or authentication mechanism used in the software. These metrics may be added to the final result and evaluate their impact on the exploitability prediction.

Another improvement that can be made in this study is the impact factor. Here, we have only considered DSCs as the measure of the vulnerability exploitability impact. Many other factors such as business factor(e.g., monetary loss) may be considered.

There exist many vulnerabilities that are exploitable without the presence of a entry point. Work can be done to be incorporate these kind of vulnerabilities in the study.

References

[1] Red Hat Documentation
[2] Awad Younis, Yashwant K. Malaiya, Indrajit Ray, Assessing vulnerability exploitability risk using software properties, in Software Quality Journal, volume 24, Issue 1, March [2015].
[3] James Walden, Jeff Stuckman, Riccardo Scandariato, Predicting Vulnerable Software Components via Text Mining, in IEEE Transactions on Software Engineering, Volume: 40, Issue: 10, [2014].
[4] Manadhata, Wing. An attack surface metric. The IEEE Transactions on Software Engineering, 37(3), 371386.[2011].
[5] Bhattacharya, P., Iliofotou, M., Neamtiu, I., & Faloutsos, M. (2012). Graph-based analysis and prediction for software evolution. In: Proceedings of the 34th international conference on software engineering (ICSE 12) (pp. 419429). ISBN: 978-1-4673-1067-3.
[6] Willy Jimenez, Amel Mammar, Ana Cavalli, Software Vulnerabilities, Prevention and Detection Methods: A Review, Proceedings SEC-MDA 2009: Security in Model Driven Architecture pp.1 - 11, [2010]
[7] Sam Ransbotham, An Empirical Analysis of Exploitation Attempts based on Vulnerabilities in Open Source Software, in workshop on economies of information security, June [2010]
[8] Yonghee SHIN, Andrew MENEELY, Laurie WILLIAMS and Jason OSBORNE ,Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities [2010]
[9] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. 2013. Using complexity metrics to improve software security. Comput. Fraud Secur. 2013, 5 (2013), 817.[2013].
[10] Massimo, B., Gabrielli, E., & Mancini, L. [2002]. Remus: A security-enhanced operating system. ACM Transactions on Information and System Security (TISSEC), 5(1), 3661.
[11] National Vulnerability Database (2018). http://www.nvd.nist.gov/. Accessed January 2018.
[12] Scientic Toolworks Understand. (2017). http://www.scitools.com/.
[13] Common Weakness Enumeration. Accessed in January 2018. http://cwe.mitre.org/
[14] FlawFinder https://www.dwheeler.com/flawfinder/.
[15] Common Vulnerability Enumeration (CVE). Accessed in March 2018. https://cve.mitre.org/.