Software Qual J (2016) 24:159–202
DOI 10.1007/s11219-015-9274-6

Assessing vulnerability exploitability risk using software properties

Awad Younis[1] · Yashwant K. Malaiya[1] · Indrajit Ray[1]

Published online: 27 March 2015

Abstract Attacks on computer systems are now attracting increased attention. While the current trends in software vulnerability discovery indicate that the number of newly discovered vulnerabilities continues to be significant, the time between the public disclosure of vulnerabilities and the release of an automated exploit is shrinking. Thus, assessing the vulnerability exploitability risk is critical because this allows decision-makers to prioritize among vulnerabilities, allocate resources to patch and protect systems from these vulnerabilities, and choose between alternatives. Common vulnerability scoring system (CVSS) metrics have become the de facto standard for assessing the severity of vulnerabilities. However, the CVSS exploitability measures assign subjective values based on the views of experts. Two of the factors in CVSS, Access Vector and Authentication, are the same for almost all vulnerabilities. CVSS does not specify how the third factor, Access Complexity, is measured, and hence it is unknown whether it considers software properties as a factor. In this work, we introduce a novel measure, Structural Severity, which is based on software properties, namely attack entry points, vulnerability location, the presence of the dangerous system calls, and reachability analysis. These properties represent metrics that can be objectively derived from attack surface analysis, vulnerability analysis, and exploitation analysis. To illustrate the proposed approach, 25 reported vulnerabilities of Apache HTTP server and 86 reported vulnerabilities of Linux Kernel have been examined at the source code level. The results show that the proposed approach, which uses more detailed information, can objectively measure the risk of vulnerability exploitability and results can be different from the CVSS base scores.

✉ Awad Younis
younis@cs.colostate.edu

Yashwant K. Malaiya
malaiya@cs.colostate.edu

Indrajit Ray
indrajit@cs.colostate.edu

[1] Computer Science Department, Colorado State University, 1873 Campus Delivery, Fort Collins, CO 80523-1873, USA

Keywords Risk assessment · Software vulnerability · Software security metrics · Attack surface · CVSS metrics · Source code analysis

1 Introduction

Security of the computer systems and networks depend on the security of software running on them. Many of the attacks on computer systems and networks are due to the fact that attackers try to potentially compromise these systems by exploiting vulnerabilities present in the software. Recent trends have shown that newly discovered vulnerability still continues to be significant (+8000 vulnerabilities in 2012–2013) (National Vulnerability Database 2013) and so does the number of attacks (+37 million attacks in 2012–2013) (Ponemon Institute (2013)). Studies have also shown that the time gap between the vulnerability public disclosure and the release of an automated exploit is getting smaller (Frei et al. 2008). Therefore, assessing the risk of exploitation associated with software vulnerabilities is needed to aid decision-makers to prioritize among vulnerabilities, allocate resources, and choose between alternatives.

A security metric is a quantifiable measurement that indicates the level of security for an attribute of a system (Jansen 2009). Security metrics give a way to prioritize threats and vulnerabilities by considering the risks they pose to information assets based on quantitative or qualitative measures. The metrics proposed include: vulnerability density, attack surface, flaw severity and severity-to-complexity, security scoring vector for web applications, and CVSS metrics. Each of them is based on specific perspectives and assumptions and measures different attributes of software security. They are intended to objectively help decision-makers in resource allocation, program planning, risk assessment, and product and service selection.

CVSS is the *de facto* standard that is currently used to measure the severity of individual vulnerabilities (Mell et al. 2007). CVSS base score measures severity based on the exploitability (the ease of exploiting a vulnerability) and impact (the effect of exploitation). The base score is rounded to one decimal place, and it is set to zero if the impact is equal to zero regardless of the formula. Exploitability is assessed based on three metrics: Access Vector (AV), Authentication (AU), and Access Complexity (AC) as shown by the following:

$$\text{Exploitability} = 20 \times \text{AV} \times \text{AU} \times \text{AC}$$

The AV metric reflects how the vulnerability is exploited in terms of local (L), adjacent network (A), or network (N). The AC metric measures the complexity of the attack required to exploit the vulnerability (once an attacker has gained access to the target system) in terms of high (H), medium (M), or low (L). The AU metric counts the number of times an attacker must authenticate to reach a target (in order to exploit a vulnerability) in terms of multiple (M), single (S), or none (N).

However, CVSS exploitability metrics have the following limitations. First, they assign static subjective numbers to the metrics based on expert knowledge regardless of the type of vulnerability. For instance, they assign AV 0.395 if the vulnerability requires local access, 0.646 if the vulnerability requires adjacent network access, and 1 if the vulnerability requires global network access. Second, two of its factors (AV and AU) have the same value for almost all vulnerabilities (Allodi and Massacci 2013). Third, there is no formal procedure for evaluating the third factor (AC) (Mell et al. 2007). Consequently, it is unclear whether CVSS considers the software structure and properties as a factor.

On the other hand, the impact subscore measures how vulnerability will directly affect an IT asset as the degree of losses in Confidentiality (I_C), Integrity (I_I), and Availability (I_A). The impact subscores are all assessed in terms of None (N), Partial (P), or Complete (C) *by security experts* and assigned one of the mentioned qualitative letter grades. Thus, there is a need for an approach that can take into account detailed information about the access complexity and the impact factors for less subjective exploitability measures.

1.1 Problem description and research motivation

The risk of vulnerability exploitability is dependent upon two factors: Exploitability and Impact, as expressed in:

$$\text{Exploitation risk} = f(\text{Exploitability, Impact})$$

The first factor, Exploitability, is the likelihood that a potential vulnerability can be successfully exploited. This factor concerns the question "Is the vulnerability exploitable?" The other factor, Impact, measures the losses that occur given a successful exploitation. This factor is related to the question "Which is the most exploitable vulnerability?"

There are challenges in assessing the exploitation risk that are needed to be addressed. In this work, we look into the major challenges in assessing the exploitability factor, and identify key questions that need to be addressed. Then, we discuss the challenges in assessing the impact factor and also identify a key question that is required to be tackled. To guide our analysis of the problem and to ensure that our solution is guided by recognized criteria, we used two guides from National Institute of Standards and Technology (NIST) namely: the Risk Management Guide for Information Technology Systems (Jansen 2009) and Directions in Security Metrics Research Report (Stoneburner et al. 2002).

Table 1 Limitations of exploitability estimators

Exploitability estimator	Limitations
1. Existence of an exploit	The data required to measure this attribute are not always available
2. Existence of a patch	The existence of a patch does not tell whether it has been applied or not because the study (Arbaugh et al. 2000) shows patches' application depends on the administrators' behavior
3. Number of vulnerabilities and number of attack entry points	These estimators are not informative as they do not specify which vulnerability is exploitable and rather they estimate the exploitability of the whole system
4. Exploit Kit in the black market	This approach requires a vulnerability intelligence provider, as the information about the attacks and tools are dynamic in nature. Moreover, if the vulnerability right now is not used by a tool or it is not a target of an attack, it does not mean that it is going to be so continually
5. Proof of concept	It is difficult and time consuming to generate a reliable exploit

1.1.1 *Exploitability factor*

There are three main challenges when it comes to assessing the exploitability factor. They are explained as follows.

(a) Exploitability Estimators:

The main challenge is determining the estimator (attribute) to be used in assessing exploitability factor. There are a number of estimators such as: existence of exploits, existence of patches, number of vulnerabilities, number of attack entry points, exploit kits in the black market, and proof of concept. Which one of those makes a good estimator of exploitability? To answer this question, let us look at every one of those estimators. As is summarized in Table 1, each of the estimators assesses exploitability from a specific perspective and has its own limitations. Thus, a good estimator should have the following characteristics: it should be measurable, it should not be expensive to obtain, and it should be obtained objectively.

(b) Evaluation of the Measures:

The next challenge is evaluating the exploitability estimators. Obtaining the measures can be achieved by using security expert opinions, one of the estimators mentioned above, or the software structure. Relying on expert opinions leads to subjectivity and can potentially hinder the accuracy of the assessment. The challenge is: How can we reduce subjectivity and minimize human involvement in exploitation assessment? On the other hand, the data regarding reported vulnerabilities or exploits are not always available especially for newly released software. Thus, the question is: How can the risk of exploitation be assessed in the absence of historical data? The alternative could be using the software code. However, in that case, the challenges would be: What type of features can be used as an indicator of vulnerability exploitability? How can these features be objectively derived? These questions are addressed in this paper.

(c) Level of Assessment:

A further challenge is deciding on the *level* of assessment that the exportability should be assessed at. Should we assess exploitability for each individual vulnerability or the whole software? Assessing the vulnerability exploitability at the software level is not informative as it assumes that all vulnerabilities have the same risk of exploitation. This is unrealistic as different vulnerabilities have different risk of exploitation. Thus, assessing the exploitation risk at the individual vulnerability level needs to be done first. Thereafter, we can combine risk value of each individual vulnerability and get the total risk of exploitation for the whole system.

1.1.2 *Impact factor*

Estimating the impact factor is challenging because it is context dependent. For instance, a shutdown of a mission critical server may be more "severe" than a print server. Manadhata and Wing (2011) introduced two types of impacts: Technical impact (e.g., privilege elevation) and Business impact (e.g., monetary loss). While the latter depends on the mission and the priority of the given context, the former can be estimated at a function level. For the technical impact the key question is: What estimators should be used to determine the technical impact? The answer could be any one of these estimators: function privilege, existence of dangerous system calls, presence of authentication, or presence of exploit mitigation. These estimators are explained as follows:

- The higher the privilege of a function (e.g., root), the more damage is going to be.
- Having a dangerous system call (e.g., open) in an entry point can help the attackers escalate their privileges and hence can cause more damage to the compromised system.
- An authentication procedure along the attack path to the vulnerable location makes exploiting a vulnerability harder and hence reducing the impact.
- Exploit mitigation reduces the impact of a vulnerability exploitation.

1.2 Research objectives and contribution

The main objective of this research is to propose an approach that can address the challenging questions discussed in the problem description. We mainly focus on reducing subjectivity in assessing vulnerability exploitation risk. To that end, we base our analysis on software properties. These properties represent measures that can objectively be derived from the source code using the attack surface analysis, the vulnerability analysis, and the exploitation analysis, irrespective of the level of security knowledge one has. Our proposed measure evaluates vulnerability exploitability based on the presence of a function call connecting attack surface entry points to the vulnerability location within the software under consideration. If such a call exists, we estimate how likely an entry point is going to be used in an attack based on the presence of the dangerous system calls. The dangerous system calls paradigm has been considered because they allow attackers to escalate a function privilege and hence cause more damage.

To determine the effectiveness of the proposed approach, the Apache HTTP server and Linux Kernel were selected as a case study. The two software systems have been selected because of: their rich history of publicly available vulnerabilities, availability of an integrated repository (which enables us to map vulnerabilities to their location in the source code), availability of the source code (which enables us to collect the measures of the proposed metrics), and their diversity in size and functionalities.

The paper is organized as follows. Section 2 presents the background. In Sect. 3, the key steps of our approach are introduced. In Sect. 4, the evaluation and the results of the proposed approach are shown. In the following section, some observations and threats to validity are presented. Section 6 presents the related work. Finally, concluding comments are given along with the issues that need further research.

2 Background

In this section, we describe the concepts and terminology related to the vulnerability exploitation risk. This section provides a brief introduction to software vulnerabilities, the attack surface metric used in the proposed approach, the system dependence graph used for reachability analysis, and the exploit databases used to validate the analysis.

2.1 Software vulnerabilities

A software vulnerability is defined as a defect in software systems which presents a considerable security risk (Pfleeger and Pfleeger 2006). A subset of the security-related defects, vulnerabilities, are to be discovered and become known eventually (Pfleeger and Pfleeger 2006). The vulnerabilities are thus a subset of the defects that are security related. The finders of the vulnerabilities disclose them to the public using some of the common reporting mechanisms available in the field. The databases for the vulnerabilities are maintained by several organizations such as National Vulnerability Database (NVD) (National Vulnerability Database 2013), Open Source Vulnerability Database (OSVDB) (OSVDB 2014), BugTraq (SecurityFocus 2015), as well as the vendors of the software. Vulnerabilities are assigned a unique identifier using MITRE Common Vulnerability and Exposure (CVE) service.

2.2 Attack surface metric

A system's attack surface is the subset of the system's resources that are used by an attacker to attack the system (Manadhata and Wing 2011). This includes the functions that serve as the entry points and exit points, the set of channels (e.g., sockets), and the set of untrusted data items (Manadhata and Wing 2011). Entry and exit points can be used by an attacker to either send or receive data from the system. Channels are the means that are used by an attacker to connect to the system. Untrusted data items are the data that an attacker can either send or receive from the system. In this study, we consider the entry points and the potential paths leading to the functions containing the vulnerabilities.

2.3 System dependence graph

A *system dependence graph* (SDG) introduced by (Horwitz et al. 1990) is an extension of the *Program dependence graph* (PDG) (Ferrante et al. 1987; Horwitz et al. 1990; Kuck et al. 1972). PDG is a directed graph representation of a program, where vertices represent program points (e.g., assignment statements, call sites, variables, and control predicates) that occur in the program and edges represent different kinds of control or data dependencies. An SDG consists of interconnected PDGs (one per procedure in the program) and extends the control and data dependencies with interprocedural dependencies. An interprocedural control-dependence edge connects procedure call sites to the entry points of the called procedure, and an interprocedural data-dependence edge represents the flow of data between actual parameters and formal parameters (and return values). A system dependence graph can be used for purposes such as code optimization, reverse engineering, program testing, program slicing, software quality assurance, and software safety analysis. This work employs SDG to determine the call sequence among functions in a source code, which may lead to a potential exploitation.

2.4 Exploit database (EDB)

EDB records exploits and vulnerable software (The Exploits Database 2013). It is used by penetration testers, vulnerability researchers, and security professionals. It reports vulnerabilities for which there is at least a proof-of-concept exploit. EDB is considered as a regulated market for the exploits. EDB contains around 24075 exploits as the time of writing this paper. Most of its data are derived from Metasploit Framework, a tool for creating and executing exploit code against a target machine. It provides a search utility that uses a CVE number to find vulnerabilities that have an exploit.

3 Approach

Figure 1 shows an overview of our approach for assessing vulnerability exploitability risk. It is based on combining the attack surface analysis, the vulnerability analysis, and the exploitation analysis.

We start by identifying the attack entry points and a vulnerability location from the source code. Next, we apply the reachability analysis to the identified attack entry points and the vulnerability location. The outcome of this step is to determine whether a vulnerability is reachable from any of the identified attack entry points (R) or not reachable (NR). After that, we verify whether any of the identified attack entry points has dangerous system calls. The result of this step is either the attack entry point has a dangerous system call (DSC) or it does not have a dangerous system call (NDSC). Then, we assess vulnerability exploitability based on the results of the steps three and four. The outcome of this step is a vulnerability is reachable with DSC, reachable with NDSC, or not reachable. Based on the result of this step, the risk of exploitation of a given vulnerability is assigned as shown in Table 2. The following subsections will explain these steps in detail.

3.1 Identify attack entry points

We define the attack entry points using the system's attack surface entry point framework proposed by Manadhata and Wing (2011). Entry points are the functions that an attacker uses to send data to the system. In this paper, we used only the entry points because they are the main target of malicious attacks. A function is a direct entry point if it receives data directly from the environment; read function defined in *unistd.h* in C library is an example (Manadhata and Wing 2011). Figure 2 shows how the entry points are identified. The required steps are as follows:

(a) Obtain the source code
(b) Identify all functions that receive data from the user environment (C/C++ Library functions)
(c) Verify whether these functions are used by any of the user functions

- Identify all functions called by the main function using cflow.
- Use python script to verify whether any of these functions has one of the C/C++ input functions.
- If you find any, then consider that function as an EP

(d) Get the list of all entry points

3.2 Find vulnerability 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 (Evans and Larochelle 2002). Figure 3 shows the way vulnerabilities are mapped to their locations.

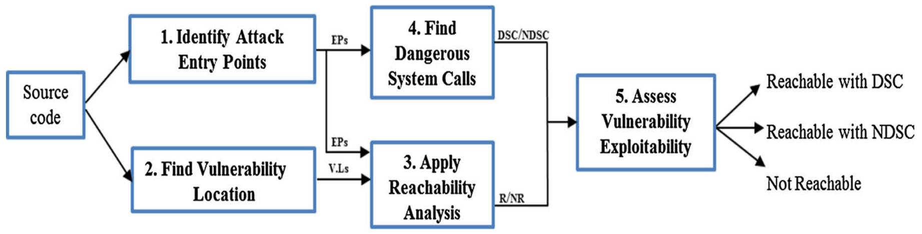

Source code
1. Identify Attack Entry Points
2. Find Vulnerability Location
EPs
EPs
V.Ls
4. Find Dangerous System Calls
3. Apply Reachability Analysis
DSC/NDSC
R/NR
5. Assess Vulnerability Exploitability
Reachable with DSC
Reachable with NDSC
Not Reachable

Fig. 1 Overview of the proposed approach

Table 2 Structural Severity measure

Metric 1 Reachability	Metric 2 Dangerous system call (DSC)	Measure Structural severity
Not Reachable	No DSC exist	Low
Not Reachable	DSC exist	Low
Reachable	No DSC exist	Medium
Reachable	DSC exist	High

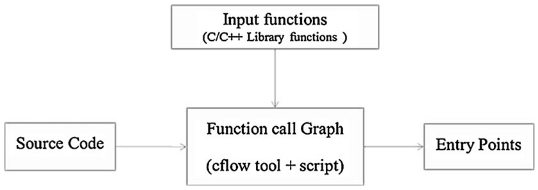


Fig. 2 Identification of attack entry points

(a) Mapping vulnerabilities using databases:

- From the vulnerability database, identify the vulnerability.
- From the Bug Repository and Version Archive:
  - Identify the vulnerable version (e.g., Apache 1.3.0)
  - Identify files by mapping Bug ID to CVE number
  - Identify the vulnerable function
- For the selected vulnerable version:
  - Search inside all folders in the main folder and find all. c files and store them in a list
  - From the list, select the. c files that contain the vulnerabilities
  - For every selected file, find the vulnerable function(s)

(b) Mapping vulnerabilities using a static code analyzer

Finding a vulnerability location can also be determined by using static code analyzers when the report does not finalize such information. The static code analyzers are tools that

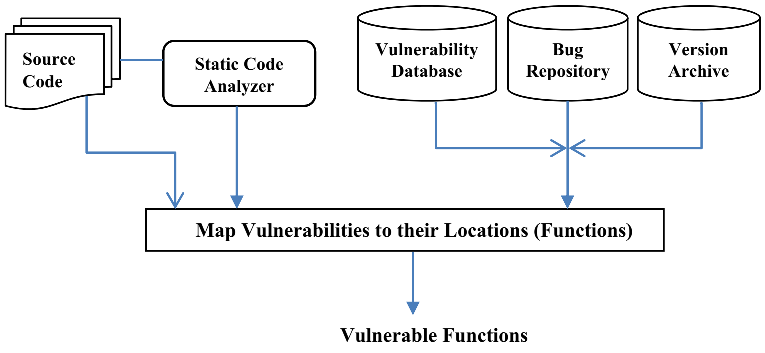


Fig. 3 Vulnerability location identification

are used to find common bugs or vulnerabilities in the code base without the need to execute the code. Secure Programming Lint (Splint) is an example. It is a tool that uses static analysis to detect vulnerabilities in programs (Evans and Larochelle 2002). However, in this paper, the vulnerability report was used to find the location of the vulnerability. The use of static tools was left as a future work.

3.3 Apply reachability analysis

Reachability means the analysis of the call relationships between the entry points and the vulnerability locations (vulnerable functions). We employed a *system dependence graph* (SDG) proposed by Horwitz et al. (1990) to determine the calls from an entry point function to a vulnerability location (vulnerable function). There are many tools available for automatically generating SDG from the source code. We have selected Understand. Understand is a static analysis tool for maintaining, measuring, and analyzing critical or large source code. We have used the Understand tool (Scientific Toolworks, Understand 2014), to generate this graph from the source code. This tool has been chosen because it is user-friendly and it has a good set of APIs that allows interaction with programming languages such as Python, Perl, and C++. Figure 4 shows how reachability analysis is used.

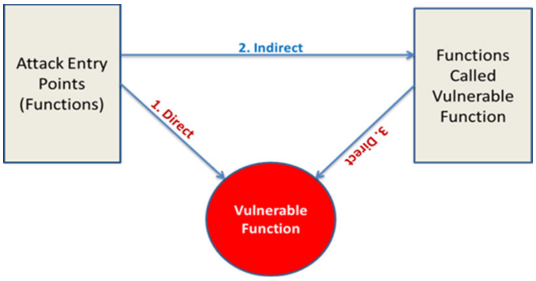
After identifying the entry points and the vulnerable functions, the reachability analysis was conducted as follows:

- Generate a (called by) graph that captures all functions that call the vulnerable function directly or indirectly
- Verify whether any of these functions is
    - Directly calling the vulnerable function and an entry point
    - Indirectly calling the vulnerable function and an entry point
- If so, the vulnerable function is recognized as reachable
- If not, the vulnerable function is recognized as not reachable.

3.4 Find dangerous system calls

System calls are the entry points to privileged kernel operations (Silberschatz et al. 2009). Calling a system call from a user function in a program can violate the least privilege principle. Massimo et al. (2002) define DSCs as specific system calls that can be used to take complete control of the system and cause a denial of service or other malicious acts.

Fig. 4 Reachability analysis


Attack Entry Points (Functions)
2. Indirect
Functions Called Vulnerable Function
1. Direct
3. Direct
Vulnerable Function

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. Table 3 shows some of the DSCs as classified by Massimo et al. (2002). There are 22 system calls of the threat level one and 32 of the threat level two.

We used the existence of DSCs as an estimate of the impact of exploitation. This is because having DSCs in a user function helps the attackers escalate their privileges and hence cause more damage to the compromised system. As the attack entry points are the main entries for the attacker to the system, we verify whether these points have DSCs. We do that by looking at the list of the identified attack entry points and verify whether each entry point contains DSCs by using a python script. If any DSCs are found, we annotate that function as an entry point with DSCs.

3.5 Assess vulnerability exploitability

An individual vulnerability exploitability risk is assessed using the measures obtained from step 1–4. Thus, a vulnerability can be classified as one of the following:

- Reachable with Dangerous System Calls.
- Reachable with No Dangerous System Calls.
- Not reachable.

4 Evaluation and Results

This section presents the results of the evaluation of the proposed measure and assesses its performance. To implement our approach, 111 vulnerabilities of Apache HTTP server and Linux Kernel have been chosen as shown in Table 4. These vulnerabilities have been collected from the National Vulnerability Database (National Vulnerability Database 2013). They have been selected based on the availability of information about their locations and their exploits. It should be noted that we have considered all Apache HTTP server vulnerabilities that have an exploit. Out of the 73 Linux Kernel vulnerabilities that have an exploit, we only examined 64 vulnerabilities. This is because seven out of the nine vulnerabilities depends on a configuration property and not a user input. In addition, we could not find information about the location of the other two vulnerabilities. Further, as

Table 3 Dangerous system calls

Threat level	Dangerous system calls
1. Full control of the system	chmod, fchmod, chown, fchown, lchown, execve, mount, rename, open, link
2. Denial of service	umount, mkdir, rmdir, umount2, ioctl, nfsservctl, truncate, ftruncate, quotactl, dup, dup2, flock, fork, kill, iopl, reboot, ioperm, clone
3. Used for subverting the invoking process	read, write, close, chdir, lseek, dup, fcntl, umask, chroot, select, fsync, fchdir, llseek, newselect, readv, writev, poll, pread, pwrite, sendfile, putpmsg, utime
4. Harmless	getpid, getppid, getuid, getgid, geteuid, getegid, acct, getpgrp, sgetmask, getrlimit, getrusage, getgroups, getpriority, sched getscheduler, sched getparam, sched get

the main focus of this research is to evaluate the capability of the proposed measure, we have only considered a few of the vulnerabilities that have no exploit. We also tried to select the vulnerabilities that are at least 3 or 4 years old, so that their lack of exploit is not due to their recent discovery.

Our approach in assessing software vulnerability exploitability is based on the sequential steps that have been discussed in Sect. 3. As applying the steps of our approach are the same for Apache HTTP server and Linux Kernel, we are only including the detailed explanation for Apache HTTP server here. However, the final results of the investigation are going to be presented in two separate tables.

4.1 Define attack entry points

Identifying the attack surface entry points requires looking at the code base and finding all entry points which could be a part of the attack surface. After finding such points, we then classify each one of them into an attack class. The code bases of the chosen version were obtained from (Archive.apache.org 2014).

The entry points along the method dimension were defined using the cflow tool and a python script. The cflow tool analyzes a code base written in C programming language and produces a graph charting dependencies among various functions (GNU Cflow 2013). Using the python script, we identified the entry points for the whole system and selected the ones that are related to the chosen vulnerabilities location. These entry points are shown in Table 5.

4.2 Find vulnerability location

The vulnerability location was determined by mapping information from NVD, Apache HTTP server bug repository (Red Hat Bugzilla Main Page 2014) and Apache SVN archive databases (Apache-SVN 2014). Vulnerabilities are either located in one of the entry points or located in a function that is called by the entry points directly or indirectly. By performing the following steps, we identified the location of the vulnerabilities.

(a) From the NVD, we selected the vulnerability. The following report shows one of the selected vulnerabilities. As can be seen, the vulnerable file and Apache version can be easily determined from the following report description.
(b) CVE-2012-0053 *protocol.c* in the Apache HTTP server 2.2.x through 2.2.21 does not properly restrict header information during construction of Bad Request (aka 400) error documents, which allows remote attackers to obtain the values of HTTP-only cookies via vectors involving a (1) long or (2) malformed header in conjunction with a crafted web script.
(c) Using Bugzilla, we mapped the CVE number to the Bug ID. Bugzilla is a bug- or issue-tracking system. Figure 5 shows the mapping step.

Table 4 Apache HTTP server and Linux Kernel vulnerabilities dataset

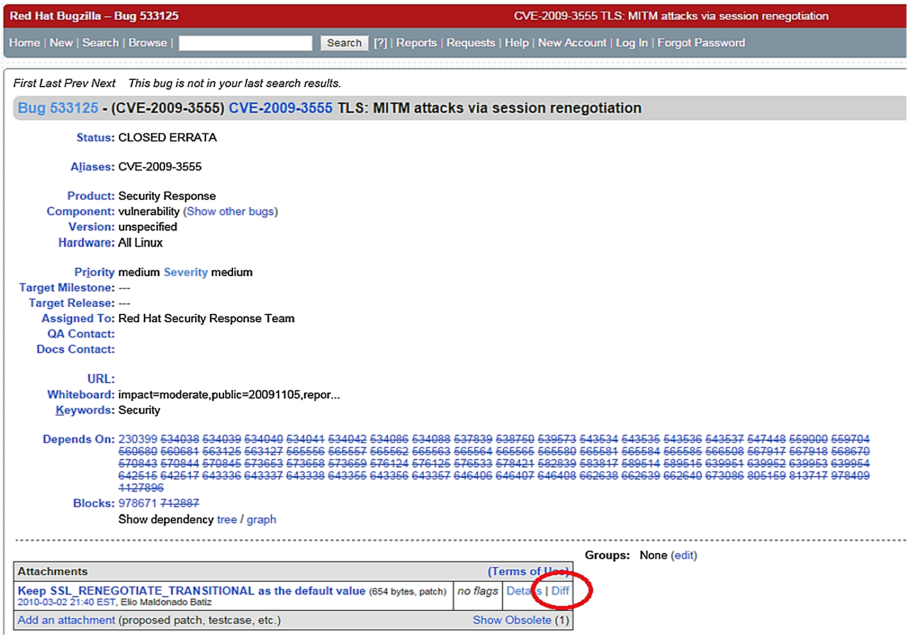
Software	Exploit exist	No exploit exist	Total each
Apache HTTP server	11	14	25
Linux Kernel	64	22	86
Total all	75	36	111

Table 5 Entry points and dangerous system calls

File name	Entry points		Dangerous system calls	
	C/C++ input functions	Entry point function	Denial of service	Full control
protocol.c	read	ap_get_mime_headers	–	–
	read	ap_read_request	–	–
	get	ap_set_sub_req_protocol	–	–
http_filter.c	scanf	ap_http_filter	–	–
http_core.c	scanf	register_hooks	–	–
mod_usertrack.c	get	spot_cookie	–	–
mod_proxy.c	scanf	register_hooks	–	link
mod_rewrite.c	scanf	register_hooks, hookuri2file	–	–
mod_proxy_ajp.c	read	ap_proxy_ajp_request	–	–
mod_deflate.c	read packets	deflate_out_filter	dup	–
mod_proxy_http.c	read, get, gethostbyname	Stream_reqbody_cl,	–	–
		Ap_proxy_http_reques,	dup	–
		proxy_http_handler	dup	–
mod_proxy_ftp.c	read packets	Proxy_send_dir_filter	dup	–
mod_proxy_balancer.c	get	balance_handler	dup	–
mod_status.c	get	Ststus_handler	–	–
core.c	get	include_config	–	–
main.c	get	main,	exit	open
mod_autoindex	read	index_directory	dup	open
proxy_http.c	read socket	ap_proxy_http_handler	dup	–
proxy_util.c	read buffer	ap_proxy_send_fb	–	–
mod_include.c	getc	get_tag	–	–
mod_imap.c	getline	imap_handler	–	open
mod_negotiation.c	get	do_negotiation	dup	open

(d) From Bugzilla, we accessed the Apache SVN to determine the vulnerable code. Apache SVN is a software used to maintain the current and historical versions of files such as source code, web pages, and documentation. We used *diff* link, which is shown circled at the bottom of Fig. 5, to access the source code and identify the vulnerable code (function). Basically, the *diff* link shows the difference between before the code has been changed and after the code has been modified. The *diff* page is shown in Fig. 6 where the vulnerable function *ap_get_mime_headers_core* has been circled with red color. The *diff* link shows the code for a bug fix, which corresponds to a vulnerability fix. The (—) symbol indicates the part of the code that has been removed, whereas (+++) symbol shows the code that has been added.

Once the vulnerable function name is identified, we look at the source code and identify the. *c* file name that contains the function and annotate its location. We followed the same



Red Hat Bugzilla – Bug 533125 CVE-2009-3555 TLS: MITM attacks via session renegotiation

Home | New | Search | Browse | Search [?] | Reports | Requests | Help | New Account | Log In | Forgot Password

First Last Prev Next This bug is not in your last search results.

Bug 533125 - (CVE-2009-3555) CVE-2009-3555 TLS: MITM attacks via session renegotiation

Status: CLOSED ERRATA

Aliases: CVE-2009-3555

Product: Security Response
Component: vulnerability (Show other bugs)
Version: unspecified
Hardware: All Linux

Priority medium Severity medium
Target Milestone: ---
Target Release: ---
Assigned To: Red Hat Security Response Team
QA Contact:
Docs Contact:

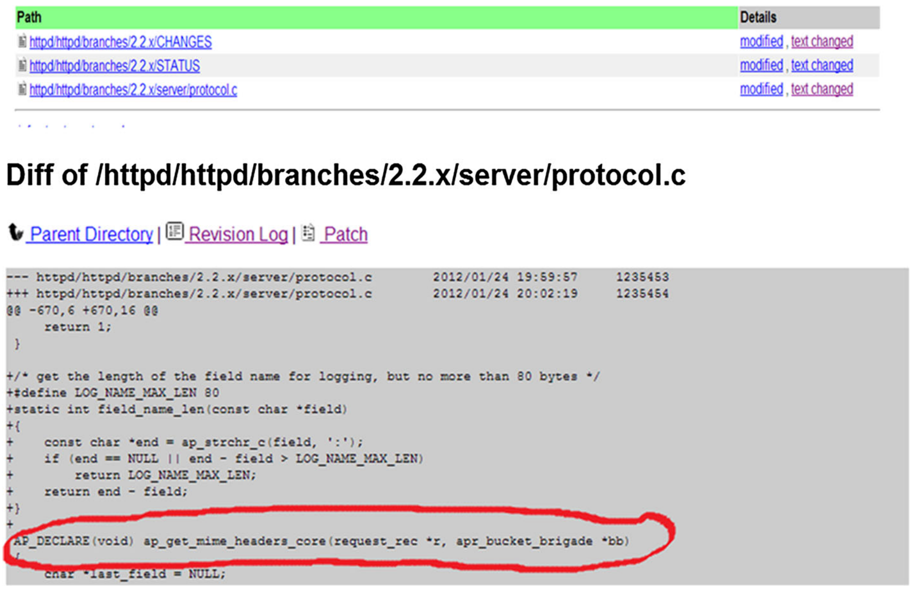
URL:
Whiteboard: impact=moderate,public=20091105,repor...
Keywords: Security

Depends On: 230399 ~~534038 534039 534040 534041 534042 534086 534088 537839 538750 539573 543534 543535 543536 543537 547448 559000 559704 560680 560681 563125 563127 565556 565557 565562 565563 565564 565565 565580 565581 565584 565585 566508 567917 567918 568670 570843 570844 570845 573653 573658 573659 576124 576125 576533 578421 582839 583817 589514 589515 639951 639952 639953 639954 642515 642517 643336 643337 643338 643355 643356 643357 646406 646407 646408 662638 662639 662640 673086 805159 813717 978409 1127896~~
Blocks: 978671 ~~712887~~
Show dependency tree / graph

Groups: None (edit)

Attachments		(Terms of Use)
Keep SSL_RENEGOTIATE_TRANSITIONAL as the default value (654 bytes, patch) 2010-03-02 21:40 EST, Elio Maldonado Batiz	no flags	Details | Diff
Add an attachment (proposed patch, testcase, etc.)		Show Obsolete (1)

Fig. 5 Mapping from CVE number to Bug ID



Path	Details
httpd/httpd/branches/2.2.x/CHANGES	modified , text changed
httpd/httpd/branches/2.2.x/STATUS	modified , text changed
httpd/httpd/branches/2.2.x/server/protocol.c	modified , text changed

Diff of /httpd/httpd/branches/2.2.x/server/protocol.c

Parent Directory | Revision Log | Patch

```
--- httpd/httpd/branches/2.2.x/server/protocol.c        2012/01/24 19:59:57      1235453
+++ httpd/httpd/branches/2.2.x/server/protocol.c        2012/01/24 20:02:19      1235454
@@ -670,6 +670,16 @@
     return 1;
 }

+/* get the length of the field name for logging, but no more than 80 bytes */
+#define LOG_NAME_MAX_LEN 80
+static int field_name_len(const char *field)
+{
+    const char *end = ap_strchr_c(field, ':');
+    if (end == NULL || end - field > LOG_NAME_MAX_LEN)
+        return LOG_NAME_MAX_LEN;
+    return end - field;
+}
+
 AP_DECLARE(void) ap_get_mime_headers_core(request_rec *r, apr_bucket_brigade *bb)
 {
     char *last_field = NULL;
```

Fig. 6 Identifying vulnerable code (Function)

steps to identify the location of the remaining vulnerabilities. Table 6 shows the locations of the chosen vulnerabilities. It has been noticed that some of the vulnerabilities have been located in more than one function.

4.3 Apply reachability analysis

Once the vulnerable functions and the entry points were identified, reachability analysis can be achieved using the Understand tool. This tool automatically generates a graph that contains a chain of all functions that can call the selected function. This graph is known as *Called By graph*. It shows what calls the selected function. Each line connecting an entity is read as "x is called by y". It should be noted that this graph is read from the bottom up. When x is called by y, x is in the bottom and y is in the top. We have added to the generated graph the entry point (EP) label, and vulnerability CVE number to make the graph easily viewable. The reachability analysis for all selected vulnerabilities can make the length of the paper more than it should be; therefore, only five vulnerabilities were selected. Those vulnerabilities are representative of the common trends among the studied vulnerabilities.

(a) CVE-2012-0053

Looking at Table 6, it can be verified that this vulnerability is located in the function *ap_get_mime_headers_core*, which in turn resides in the file *protocol.c*. To determine whether this vulnerability is reachable from an entry point, we performed the following:

Table 6 Vulnerabilities locations (functions)

Vulnerability	File name	Vulnerable function name
CVE-2012-0053	protocol.c	ap_get_mime_headers_core
CVE-2011-4415	util.c	ap_pregsub
CVE-2011-4317	mod_proxy.c	proxy_handler
	mod_rewrite.c	hook_fixup
CVE-2011-3192	byterange_filter.c	parse_byterange
CVE-2010-0434	protocol.c	clone_headers_no_body
CVE-2010-0408	mod_proxy_ajp.c	ap_proxy_ajp_request
CVE-2009-1891	mod_deflate.c	deflate_out_filter
CVE-2009-1890	mod_proxy_http.c	Stream_reqbody_cl
CVE-2008-2939	mod_proxy_ftp.c	Proxy_send_dir_filter
CVE-2007-6420	mod_proxy_balancer.c	balance_handler
CVE-2006-5752	mod_status.c	Ststus_handler
CVE-2009-1195	config.c	process_resource_config_nofnmatch, ap_process_config_tree
CVE-2007-5000	mod_imagemap.c	menu_header
CVE-2007-4465	mod_autoindex	index_directory
CVE-2010-0010	Proxy_util.c	ap_proxy_send_fb
CVE-2004-0940	mod_include.c	get_tag
CVE-2007-6388	mod_status.c	status_handler
CVE-2005-3352	mod_imap.c	read_quoted
CVE-2004-0488	ssl_util.c	ssl_util_uuencode_binary
CVE-2008-0455	mod_negotiation.c	make_variant_list
CVE-1999-0107	http_request.c	process_request_internal
CVE-2004-0493	protocol.c	ap_get_mime_headers
CVE-2006-3747	mod_rewrite.c	apply_rewrite_rule
CVE-2013-1896	mod_dav.c	dav_method_merge
CVE-2006-3918	http_protocol.c	get_canned_error_string

- Generate a called by graph that captures all functions that call the vulnerable function directly or indirectly. Figure 7 shows the generated graph with the function name at the top of the rectangle and.*c* file name in the bottom.
- Verify whether any of these functions is an entry point. Using the entry points in Table 5, the attacker can have an access to the vulnerability by two ways:
  - **Directly**: The *protocol.c* component has two entry points namely: *ap_get_mime_headers* and *ap_read_request* that directly can call the vulnerable function. These two entry points are located in the *protocol.c* file where the vulnerable function is located. None of these entry points have a dangerous system call.
  - **Indirectly**: The component *core.c* has an entry point function "*register_hooks*" that indirectly can call the vulnerable function in the component *protocol*.c from multiple paths. This entry point does not have a dangerous system call.

As a result, it can be concluded that there is a call relationship between the entry points and the vulnerable functions. The vulnerable function is also indirectly reachable from an entry point located in a different component. Neither of these entry points has DSCs.

(b) CVE-2004-0488

Looking at Table 6, it can be shown that this vulnerability is located in the function *ssl_utill_uuencode_binary*, which lives in the file *ssl_util.c*. To determine whether this vulnerability is reachable from an entry point, we performed the following:

- Generate a called by graph that captures all functions that call this function directly or indirectly. Figure 8 shows the generated graph.
- Verify whether any of these functions is an entry point. Using the entry points in Table 5, the attacker has no way of manipulating this vulnerability because there are no entry points found in any of these functions including the vulnerable function.

As a result, it can be concluded that this vulnerable function is not reachable from an entry point.

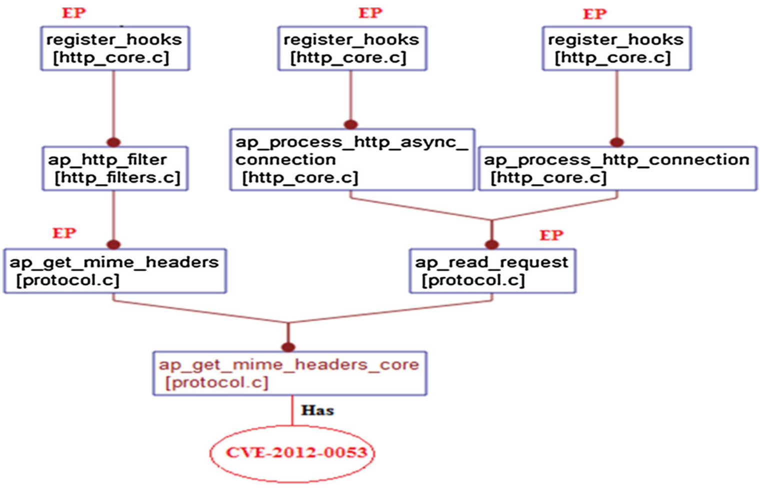

EP
register_hooks [http_core.c]
EP
register_hooks [http_core.c]
EP
register_hooks [http_core.c]
ap_http_filter [http_filters.c]
ap_process_http_async_connection [http_core.c]
ap_process_http_connection [http_core.c]
EP
ap_get_mime_headers [protocol.c]
EP
ap_read_request [protocol.c]
ap_get_mime_headers_core [protocol.c]
Has
CVE-2012-0053

Fig. 7 Direct and indirect call sequences from the EP to the vulnerable function

(c) CVE-2007-4465

From Table 6, it can be verified that this vulnerability is located in the function *index_directory*, which resides in the *mod_autoindex.c*. To determine whether this vulnerability is reachable from an entry point, we performed the following:

- Generate a called by graph that captures all functions that call this function directly or indirectly. Figure 9 shows the generated graph.
- Verify whether any of these functions is an entry point. Using the entry points in Table 5, it has been found that the vulnerable function itself is also an entry point. Besides, *register_hooks* is an entry point too. Thus, the attacker has an access to this vulnerability by invoking the vulnerable function directly. This entry point has a DSC.

As a result, it can be concluded that this vulnerability is reachable by directly calling the vulnerable function. Besides, this function has a DSC.

(d) CVE-2010-0434

From Table 6, it can be confirmed that this vulnerability is located in the function *clone_headers_no_body*, which resides in the file *protocol.c*. To determine whether this vulnerability is reachable from an entry point, we performed the following:

- Generate a called by graph that captures all functions that call the vulnerable function directly or indirectly. Figure 10 shows the generated graph.
- Verify whether any of these functions is an entry point. Using the entry points in Table 5, it has been found that the attacker can reach the vulnerable function in two ways:
    - **Directly:** The component *protocol.c* has an entry point: *ap_set_sub_req_protocol* that can directly call the vulnerable function. This entry point has no DSCs.
    - **Indirectly:** The component *mod_rewrite.c* has two entry points namely: *register_hooks* and *hookuri2file*. Thus, the attacker has an access to this vulnerability from multiple paths as shown in Fig. 11. This entry point has NDSCs.

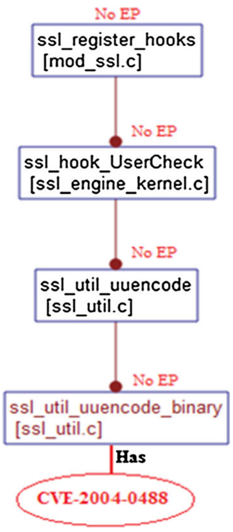


Fig. 8 No call sequence that reaches the vulnerable function in sll_util.c

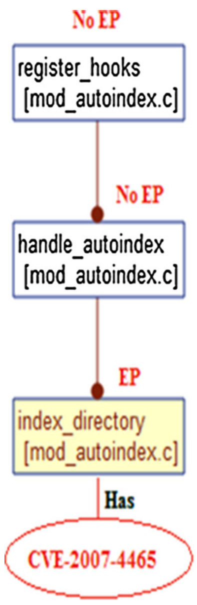

No EP
register_hooks [mod_autoindex.c]
No EP
handle_autoindex [mod_autoindex.c]
EP
index_directory [mod_autoindex.c]
Has
CVE-2007-4465

Fig. 9 Direct call sequence from the EP to the vulnerable function in mod_autoindex.c

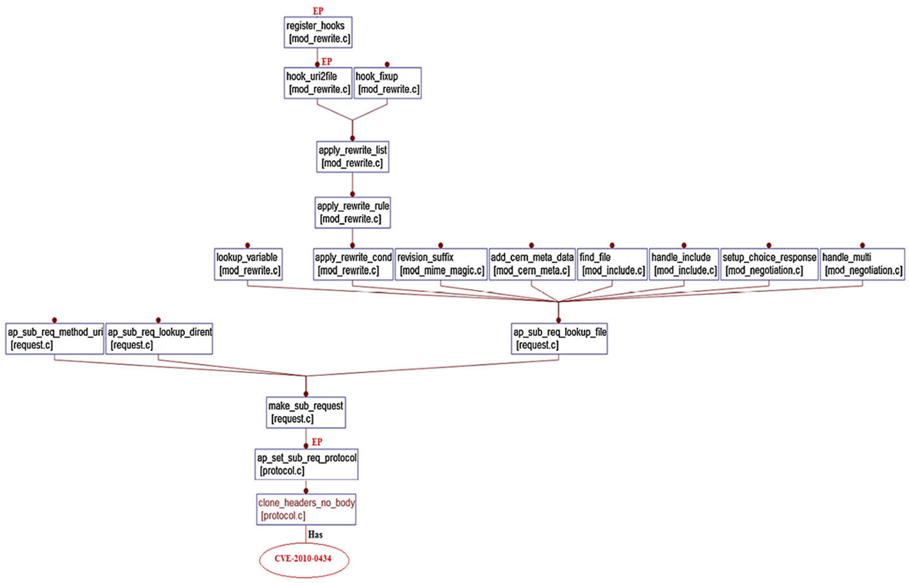

EP
register_hooks [mod_rewrite.c]
EP
hook_uri2file [mod_rewrite.c]
hook_fixup [mod_rewrite.c]
apply_rewrite_list [mod_rewrite.c]
apply_rewrite_rule [mod_rewrite.c]
lookup_variable [mod_rewrite.c]
apply_rewrite_cond [mod_rewrite.c]
revision_suffix [mod_mime_magic.c]
add_cern_meta_data [mod_cern_meta.c]
find_file [mod_include.c]
handle_include [mod_include.c]
setup_choice_response [mod_negotiation.c]
handle_multi [mod_negotiation.c]
ap_sub_req_method_uri [request.c]
ap_sub_req_lookup_dirent [request.c]
ap_sub_req_lookup_file [request.c]
make_sub_request [request.c]
EP
ap_set_sub_req_protocol [protocol.c]
clone_headers_no_body [protocol.c]
Has
CVE-2010-0434

Fig. 10 Direct and indirect call sequences from the EP to the vulnerable function in prorotocol.c

As a result, it can be concluded that this vulnerability is reachable by directly and indirectly calling the vulnerable function from multiple paths. Besides, these functions have NDSCs.

(e) CVE-2009-1195

From Table 6, it can be decided that this vulnerability is located into two functions namely: *ap_process_cnfig_tree* and *process_resource_config_nofnmatch*, which both live in the file *config.c*. To determine whether this vulnerability is reachable from an entry point, we performed the following:

- Generate a called by graph that captures all functions that call the vulnerable function directly or indirectly. Figure 11 shows the generated graph.
- Verify whether any of these functions is an entry point. Using the entry points in Table 5, the attacker has an access to this vulnerability by invoking the vulnerable function in two ways:
  - **Indirectly**: The *main.c* component has an entry point namely: *main* that can indirectly call the vulnerable function *process_resource_config_nofmatch* throughout the path: *main, ap_read_config,* and *ap_process_resource_config* as shown on the left side of the graph. The entry points have a DSC.
  - **Directly:** There are two ways the vulnerable function can be directly called. First, the *main.c* component has an entry points namely: ap*_process_resource_config* that can directly call the vulnerable function. Second, throughout the same entry point *main,* the vulnerable function *ap_process_resource_config_tree* can be directly called as shown on the right side of the graph. The entry points have a DSC.

As a result, it can be concluded that there is a call relationship between the entry points in the component *main.c* and the two vulnerable functions in the component *config.c*. The vulnerable function is also indirectly reachable from an entry point located in a different component. The entry point has DSCs.

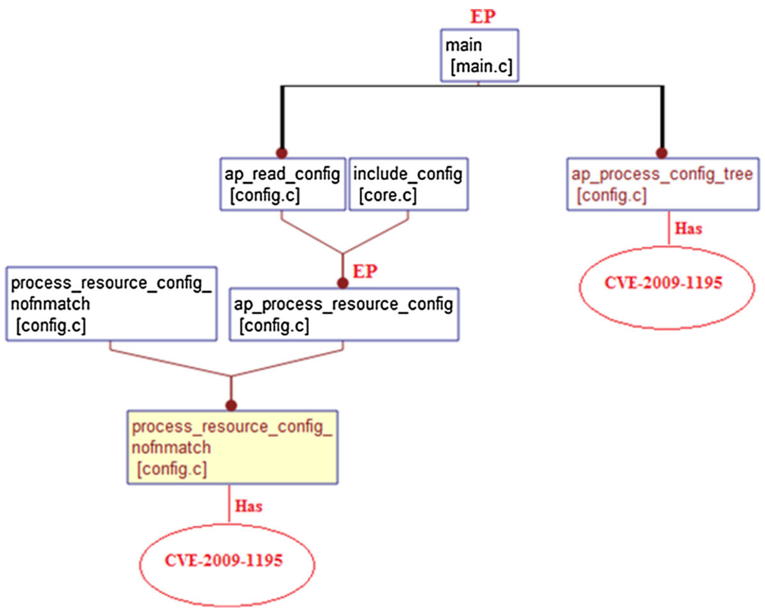

EP
main [main.c]
ap_read_config [config.c]
include_config [core.c]
ap_process_config_tree [config.c]
Has
CVE-2009-1195
EP
process_resource_config_nofnmatch [config.c]
ap_process_resource_config [config.c]
process_resource_config_nofnmatch [config.c]
Has
CVE-2009-1195

Fig. 11 Direct and indirect call sequences from the EP to the vulnerable function in config.c

Table 7 Obtained metrics of the proposed measure

Vulnerability	Function name	Path from entry points	Dangerous system call	Reachability
CVE-2004-0488	ssl_util_uuencode_binary	no path	NDSC	NR
CVE-2004-0940	get_tag	get_tag	NDSC	R
CVE-2005-3352	read_quoted	imap_handler	DSC	R
CVE-2006-5752	Ststus_handler	ststus_handler	NDSC	R
CVE-2007-6420	balance_handler	balance_handler	DSC	R
CVE-2007-5000	menu_header	no path	NDSC	NR
CVE-2007-4465	index_directory	index_directory	DSC	R
CVE-2007-6388	status_handler	status_handler	NDSC	R
CVE-2008-2939	proxy_send_dir_filter	proxy_send_dir_filter	DSC	R
CVE-2008-0455	make_variant_list	do_negotiation → store_variant_list	DSC	R
CVE-2009-1891	deflate_out_filter	deflate_out_filter	NDSC	R
CVE-2009-1195	process_resource_config_nofnmatch,	main → ap_read_config → ap_pro cess_resources_config	DSC	R
		include_config → ap_process_resources_config	NDSC	
	ap_process_config_tree	Main	DSC	
CVE-2009-1890	stream_reqbody_cl	stream_reqbody_cl	NDSC	R
		ap_proxy_http_request → proxy_http_handler	DSC	
CVE-2010-0010	ap_proxy_send_fb	ap_proxy_send_fb	NDSC	R
		ap_proxy_http_handler (Proxy_http.c)	DSC	
CVE-2010-0434	clone_headers_no_body	ap_set_sub_req_protocol	NDSC	R
CVE-2010-0408	ap_proxy_ajp_request	ap_proxy_ajp_request	NDSC	R
CVE-2011-4415	ap_pregsub	register_hooks → spot_cookie	NDSC	R
CVE-2011-3192	parse_byterange	register_hooks → ap_ byterange_filter	NDSC	R
CVE-2011-4317	proxy_handler	register_hooks	DSC	R
	hook_fixup	register_hooks	NDSC	
CVE-2012-0053	ap_get_mime_headers_core	register_hooks → ap_http_filter → p_get_mime_headers	NDSC	R
CVE-1999-0107	process_request_internal	Standalone_main → makechild → childmain → ap_process_request	NDSC	R

Table 7 continued

Vulnerability	Function name	Path from entry points	Dangerous system call	Reachability
CVE-2004-0493	ap_get_mime_headers	register_hooks → ap_http_filter	NDSC	R
CVE-2006-3747	apply_rewrite_rule	register_hooks → hook_urlfile → apply_rewrite_list	NDSC	R
CVE-2013-1896	dav_method_merge	register_hooks → av_handelr	DSC	R
CVE-2006-3918	get_canned_error_string	ap_read_request → ap_send_error_response	NDSC	R

R reachable, *NR* not reachable, *DSC* dangerous system call, *NDSC* no dangerous system call

4.4 Find dangerous system calls

For the identified attack entry points, we have checked whether an entry point has dangerous system calls. These dangerous system calls are shown in Table 5.

4.5 Assessing vulnerability exploitability

After tracing the call sequence from the entry points to the vulnerability location and identifying the DSCs, the individual vulnerability Structural Severity can be evaluated using the results included in Table 7. Looking at Table 7, a vulnerability is either:

1. Reachable with dangerous system calls.
2. Reachable with no dangerous system calls.
3. Not reachable.

Table 7 gives the Structural Severity measures of the Apache HTTP server vulnerabilities using the proposed method. Every vulnerability has its own location and the availability of the path from the entry point next to it. If a path from an entry point to a vulnerable function is found, the vulnerability considered to be reachable; otherwise, it is considered not reachable. Besides, the existence of the DSCs for every entry point has been specified.

4.6 Assigning a vulnerability Structural Severity value

The Structural Severity is measured using the reachability and DSCs metrics. It is then assigned one of the three values: high, medium, and low as described in Sect. 3. Based on these three values, the chosen vulnerabilities were assessed and compared to the CVSS overall severity scores. In following subsections, we will show the Structural Severity values for the selected vulnerabilities of Apache HTTP server and Linux Kernel datasets, respectively. It should be noted that we have used the exploit database (EDB), Open Source Vulnerability Database, and the Meta exploit database (Metasploit Database 2014) for gathering the exploits for the chosen vulnerabilities. Besides, the authors are not aware of a database that reports the impact of vulnerabilities exploitation and hence comparing those factors is not visible. The abbreviations used in the tables are explained as follows:

- DSC: Dangerous System Call.
- NDSC: No Dangerous System Call.
- AC: Access Complexity metric. Its values are Low (L), Medium (M), and High (H).
- AV: Access Vector metric. Its values are Network (N), Local (L), and Adjacent Network (A).
- AU: Authentication metric. Its values are none (N), single (S), and multiple (M).
- The CVSS impact metrics use C for Confidentiality, I for Integrity, A for Availability. These metrics takes the following values: None (N), Partial (P), and Complete (C).
- SS: Total subscore

4.6.1 Apache HTTP server

Table 8 shows the Structural Severity and the CVSS Metrics values for the Apache HTTP server dataset. The table also shows the availability of an exploit for every vulnerability. The following has been observed from Table 8:

Table 8 Obtained structural severity metrics compared to CVSS metrics of apache HTTP server dataset

No	Vulnerability	Structural severity			CVSS									Exploit existence
		Reachability	DSC	Severity	Exploitability				Impact				Severity	
					AC	AV	AU	SS	C	I	A	SS		
1	CVE-2004-0488	NR	NDSC	L	L	N	NR	10	P	P	P	6.4	H	NEE
2	CVE-2007-5000	NR	NDSC	L	M	N	NR	8.6	N	P	N	2.9	M	NEE
3	CVE-2008-0455	R	DSC	H	M	N	NR	8.6	N	P	N	2.9	M	EE
4	CVE-2009-1890	R	DSC	H	M	N	NR	8.6	N	N	C	6.9	H	EE
5	CVE-2010-0010	R	DSC	H	M	N	NR	8.6	P	P	P	6.4	M	EE
6	CVE-2013-1896	R	DSC	H	M	N	NR	8.6	N	N	P	2.9	M	EE
7	CVE-2004-0940	R	NDSC	M	M	L	NR	3.4	C	C	C	10	M	EE
8	CVE-2011-3192	R	NDSC	M	L	N	NR	10	N	N	C	6.9	H	EE
9	CVE-2012-0053	R	NDSC	M	M	N	NR	8.6	P	N	N	2.9	M	EE
10	CVE-1999-0107	R	NDSC	M	L	N	NR	10	N	N	P	2.9	M	EE
11	CVE-2004-0493	R	NDSC	M	L	N	NR	10	N	P	P	4.9	M	EE
12	CVE-2006-3747	R	NDSC	M	H	N	NR	4.9	C	C	C	10	H	EE
13	CVE-2006-3918	R	NDSC	M	M	N	NR	8.6	N	P	N	2.9	M	EE

Table 8 continued

No	Vulnerability	Structural severity			CVSS									Exploit existence
		Reachability	DSC	Severity	Exploitability				Impact				Severity	
					AC	AV	AU	SS	C	I	A	SS		
14	CVE-2006-5752	R	NDSC	M	M	N	NR	8.6	N	P	N	2.9	M	NEE
15	CVE-2007-6388	R	NDSC	M	M	N	NR	8.6	N	P	N	2.9	M	NEE
16	CVE-2009-1891	R	NDSC	M	M	N	NR	8.6	N	N	C	6.9	H	NEE
17	CVE-2010-0434	R	NDSC	M	M	N	NR	8.6	P	N	N	2.9	M	NEE
18	CVE-2010-0408	R	NDSC	M	L	N	NR	10	N	N	P	2.9	M	NEE
19	CVE-2011-4415	R	NDSC	M	H	L	NR	1.9	N	N	P	2.9	L	NEE
20	CVE-2005-3352	R	DSC	H	M	N	NR	8.6	N	P	N	2.9	M	NEE
21	CVE-2007-6420	R	DSC	H	M	N	NR	8.6	N	P	N	2.9	M	NEE
22	CVE-2007-4465	R	DSC	H	M	N	NR	8.6	N	P	N	2.9	M	NEE
23	CVE-2008-2939	R	DSC	H	M	N	NR	8.6	N	P	N	2.9	M	NEE
24	CVE-2009-1195	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	NEE
25	CVE-2011-4317	R	DSC	H	M	N	NR	8.6	N	P	N	2.9	M	NEE

EE exploit exist, *NEE* no exploit exist, *SS* subscore

- Two out of the 25 vulnerabilities are not reachable and have no exploits.
    - Thirteen out of the 25 vulnerabilities are reachable with no existing exploits. More than half of these vulnerabilities have DSCs and hence have been assigned a high Structural Severity value.
    - The remaining 11 vulnerabilities are reachable and have exploits. Four out of them have DSCs and thus have been assigned a high Structural Severity value.
- The majority of the vulnerabilities have been assigned high exploitability value by both metrics.
- More than half of the vulnerabilities have been found to have a low impact value by both metrics (CVSS impact SS, and Structural Severity DSC). This could be attributed to the type of software.
- More than half of the vulnerabilities have been assigned a medium severity value by the two metrics.
- It should be noted that all the vulnerabilities require no Authentication, whereas only three vulnerabilities require a local network access vector (L). The majority of the Access Complexity metric values are low and medium, and these values have no significant effect on the overall CVSS exploitability subscore.
- The exploitability total subscore is significantly changed only when the Access Complexity metric value is high. Nonetheless, there are only two vulnerabilities that have a high Access Complexity metric value.
- Out of the 11 vulnerabilities that have an exploit, only two vulnerabilities have been assigned a low CVSS exploitability subscore.

It should be notated that the vulnerabilities in Table 8 have been grouped into five groups based on their similarity with regard to their reachability, DSCs, and availability of exploit. This demonstrates their relationship. The following explains these groups.

(a) CVE-2004-0488 and CVE-2007-5000:

According to the CVSS metrics, the CVE-2004-0488 has a low Access Complexity value and a high overall severity, whereas CVE-2007-5000 has a low Access Complexity and a medium overall severity. Considering the network accessibility factor is useful but not sufficient. In order for a vulnerability to be exploited, it first has to be reachable regardless of the access conditions. However, based on the software structure analysis, we did not find any call relationship between the vulnerable function and any of the entry point functions. Additionally, no exploit was found for these vulnerabilities in any of the exploit databases. Thus, based on the proposed metrics, they have been assigned a low Structural Severity. Having just a potential vulnerability does not mean that it is groining to be exploited.

(b) CVE-2008-0455, CVE-2009-1890, CVE-2010-0010, and CVE-2013-1896:

As stated by the CVSS metrics, the CVE-2008-0455, CVE-2010-0010, and CVE-2013-1896 have a medium Access Complexity value and a medium overall severity, whereas CVE-2009-1890 has a medium Access Complexity and a high overall severity. Based on software structure analysis, we found the two vulnerabilities to be reachable and to have a DSC. Additionally, there exists an exploit for these vulnerabilities. However, based on the proposed metrics, they have been assigned a high Structural Severity.

(c) CVE-2012-0053, CVE-2004-0940, and CVE-2011-3192:

Based on the CVSS metrics, both CVE-2012-0053 and 2004-0940 have a medium Access Complexity value and a medium overall severity. On the other hand, CVE-2011-3192 has a low Access Complexity value and a high overall severity. Based on software structure analysis, we found these three vulnerabilities to be reachable and to have no DSC. Additionally, there exist an exploit for these vulnerabilities. Thus, based on the proposed metrics, the vulnerabilities have been assigned a medium Structural Severity.

(d) CVE-2006-5752, CVE-2007-6388, CVE-2009-1891, CVE-2010-0434, CVE-2010-0408, CVE-2011-4415:

Consistent with the CVSS metrics, the following vulnerabilities: CVE-2006-5752, CVE-2007-6388, and CVE-2010-0434 have a medium Access Complexity and a medium overall severity. Besides, CVE-2009-1891 has a medium Access Complexity and a high overall severity. Additionally, CVE-2010-0408 has a low Access Complexity and a medium overall severity. Moreover, CVE-2011-4415 has a medium Access Complexity and a high overall severity. However, based on software structure analysis, we found that all of the vulnerabilities are reachable and have no DSC. Furthermore, no exploit was found for these vulnerabilities. Based on the proposed metrics, these vulnerabilities have been assigned a medium Structural Severity.

(e) CVE-2005-3352, CVE-2007-6420, CVE-2007-4465, CVE-2008-2939, CVE-2009-1195, CVE-2010-0010, and CVE-2011-4317:

According to the CVSS metrics, all the above vulnerabilities have a medium Access Complexity value except for CVE-2009-1195 which has a low Access Complexity sub-score. Besides, all of them had a medium overall severity. However, based on software structure analysis, we found that all of them are reachable and have a DSC. Additionally, there was no exploit found for these vulnerabilities. Based on the proposed metrics, these vulnerabilities have been assigned a high Structural Severity.

4.6.2 Linux Kernel

Table 11 in the Appendix shows the Structural Severity and the CVSS Metrics values for Linux Kernel. The table also shows the availability of an exploit for every vulnerability. The following can be observed from Table 11 in the Appendix:

- Three out of the 86 vulnerabilities are not reachable and have no exploits. Two of them have DSCs.
- Nineteen out of the 86 vulnerabilities are reachable with no exploit exist for them. More than half of these vulnerabilities have DSCs and hence have been assigned a high Structural Severity value.
- All 64 vulnerabilities that have an exploit have been found to be reachable. More than half of these vulnerabilities have a DSC and thus have been assigned a high Structural Severity value.
- The majority of the vulnerabilities have been assigned a high exploitability value by the Structural Severity Reachability metric. On the other hand, most of the vulnerabilities have been assigned a low exploitability value by the CVSSS exploitability metrics. This could be attributed to the fact that most of the vulnerabilities are accessed locally.
- The majority of the vulnerabilities have been found to have a high impact value by both metrics (CVSS impact SS and Structural Severity DSC).

- More than half of the vulnerabilities have been assigned a high severity value by the Structural Severity metrics, whereas only a few have been assigned a low severity value. This can be explained by the previous two observations. On the other hand, the CVSS severity metrics do not have noticeable variation among their values except for a slightly larger number of medium severity values.
- It should be noted that most of the vulnerabilities require no Authentication except for two, which required only a Single System Authentication (SS). The majority of the vulnerabilities require a local network access vector (L). However, there are 14 vulnerabilities that require a network access vector (N), and there are only two vulnerabilities that require an adjacent network access vector (A). The majority of the Access Complexity metric values are low.
- Neither the low nor the medium Access Complexity metric values have a significant effect on the overall CVSS exploitability subscore. The exploitability total subscore is significantly changed only when the Access Complexity metric value is high. Nonetheless, there are only five vulnerabilities that have a high Access Complexity value.
- Out of the 64 vulnerabilities that have an exploit, seven vulnerabilities have been assigned a high CVSS exploitability subscores and 12 vulnerabilities have been assigned low CVSS exploitability subscores.

It should be notated that the vulnerabilities in Table 11 in the Appendix have been grouped into two groups based on the availability of exploit. As the number of the vulnerabilities of Linux Kernel is large, we will only discuss some noteworthy vulnerabilities.

(a) CVE-2003-0462, CVE-2004-1235, CVE-2006-2629, CVE-2006-5757, and CVE-2010-4258:

According to the CVSS metrics, these vulnerabilities have a high Access Complexity value and hence low overall exploitability subscore. Four of them have a medium overall severity and only one (CVE-2003-0462) has a low overall severity. Based on software structure analysis, we found that these vulnerabilities are reachable and three of them have a DSC. The other two vulnerabilities (CVE-2003-0462 and CVE-2006-5757) do not have a DSC. Additionally, there exists an exploit for these vulnerabilities. Thus, based on the proposed metrics, the three vulnerabilities that have a DSC have been assigned a high Structural Severity value, while the other two vulnerabilities that do not have DSCs have been assigned a medium Structural Severity value.

(b) CVE-2003-0619, CVE-2004-1137, CVE-2006-2444, CVE-2007-1357, CVE-2009-0065, CVE-2009-3613 and CVE-2010-1173:

According to the CVSS metrics, these vulnerabilities have a network Access Vector value and thus have been assigned a high overall exploitability subscore. Six vulnerabilities have been assigned a high overall severity and only one vulnerability, CVE-2003-0619, that has been assigned a medium overall severity. Based on the software structure analysis, these vulnerabilities have been found reachable and three of them have a DSC. The other four vulnerabilities (CVE-2003-0619, CVE-2009-0065, CVE-2009-3613, and CVE-2010-1173) do not have a DSC. Besides, there exists an exploit for these vulnerabilities. Thus, based on the proposed metrics, the three vulnerabilities that have DSCs have been assigned a high Structural Severity value, while the other four vulnerabilities that do not have DSCs have been assigned a medium Structural Severity value.

(c) CVE-2005-0124, CVE-2005-2492, and CVE-2005-2500:

According to the CVSS metrics, these vulnerabilities have a low Access Complexity value. Two of them (CVE-2005-0124 and CVE-2005-2492) have been assigned a low overall exploitability subscore and a low overall severity, and this is because they are locally accessed. On the other hand, CVE-2005-2500 has been assigned a high overall exploitability subscore and a high overall severity, and this is because it is accessed via a network. Based on software structure analysis, these vulnerabilities have been found not reachable and one of them, CVE-2005-0124, does not have a DSC. Moreover, no exploit was found for these vulnerabilities in any of the exploit databases. Hence, based on the proposed metrics, they have been assigned a low Structural Severity value.

(d) CVE-2002-0499, CVE-2003-0462, CVE-2004-1016, CVE-2004-1073, CVE-2004-1333, CVE-2005-0736, CVE-2005-2973, CVE-2006-5757, CVE-2009-0676, CVE-2009-1961, CVE-2010-1636, and CVE-2010-4073:

According to the CVSS metrics, these vulnerabilities have a local Access Vector value and thus have been assigned a low overall exploitability subscore and a low overall severity. All of these vulnerabilities require no authentication. Besides, eight out of these vulnerabilities have a low Access Complexity value. Out of the remaining four vulnerabilities, only two vulnerabilities (CVE-2003-0462 and CVE-2006-5757) have a high Access Complexity value whereas the other two have a medium access complexity (CVE-2009-1961 and CVE-2010-4073). Based on the software structure analysis, we found that these vulnerabilities are reachable and seven of them have a DSC. Besides, there exists an exploit for all of them. Hence, based on the proposed metrics, the vulnerabilities that have a DSC have been assigned a high Structural Severity value, while the vulnerabilities that do not have DSCs have been assigned a medium Structural Severity value.

4.7 Performance evaluation of the proposed metric

Since data is available about the existence of exploits, we can compare the Structural Severity Reachability metric with the CVSS exploitability metrics based on the availability of exploits. To evaluate the performance of these two metrics, we used sensitivity, precision, and F-measure measures. These performance measures are explained using a confusion matrix as shown in Table 9. The confusion matrix table shows the actual vs. the predicted results. For the two-class problem (a vulnerability is either exploitable or not exploitable), the following is defined based on Table 9.

- True Positive (TP): the number of the vulnerabilities predicted as exploitable, which do in fact have an exploit.
- False Negative (FN): the number of vulnerabilities predicted as not exploitable, which turn out to have an exploit.
- False Positive (FP): the number of vulnerabilities predicted as exploitable when they have no exploit.
- True Negative (TN): the number of vulnerabilities predicted as not exploitable when there is no exploit.

The selected performance measures can be derived as follows.

4.7.1 Sensitivity (Recall)

Sensitivity, which also termed recall, is defined as the ratio of the number of vulnerabilities correctly predicted as exploitable to the number of vulnerabilities that are actually exploitable as shown by the following:

$$Sensitivity = \frac{TP}{TP + FN}$$

4.7.2 Precision

Precision, which is also known as the correctness, is defined as the ratio of the number of vulnerabilities correctly predicted as exploitable to the total number of vulnerabilities predicted as exploitable as shown by the following:

$$Precision = \frac{TP}{TP + FP}$$

For convenient interpretation, we express these two measures in terms of percentage, where a 100 % is the best value and 0 % is the worst value. Both precision and sensitivity should be as close to the value 100 as possible (no false positives and no false negatives). However, such ideal values are difficult to obtain because sensitivity and precision often change in opposite directions. Therefore, a measure that combines sensitivity and precision in a single measure is needed. Hence, we will introduce the F-measure in the following section. We believe that it is more important to identify exploitable vulnerabilities even at the expense of incorrectly predicting some not exploitable vulnerabilities as exploitable vulnerabilities. This is because a single exploitable vulnerability may lead to serious security failures. Having said that, we think more weight should be given to sensitivity than precision. Thus, we include F2-measure, which weights sensitivity twice as precision, to evaluate the two metrics.

4.7.3 F-measure

F-measure can be interpreted as the weighted average of sensitivity and precision. It measures the effectiveness of a prediction with respect to a user attaches the β times as much importance to sensitivity as precision. The general formula for the F-measure is shown by the following:

$$F_\beta - \text{Measure} = \frac{\left(1 + \beta^2\right) \times Precision \times Senetivity}{\left(\beta^2 \times Precision\right) + Senetivity}$$

β is a parameter that controls a balance between sensitivity and precision. When $\beta = 1$, F-measure becomes to be equivalent to the harmonic mean, whereas when $\beta < 1$ it

Table 9 Confusion matrix

	Prediction	
	Exploitable	Not exploitable
Actual		
Exploitable	TP = True Positive	FN = False Negative
Not exploitable	FP = False Positive	TN = True Negative

becomes more precision oriented. However, when $\beta > 1$, F-measure becomes more sensitivity oriented. In this paper, β has been chosen to be 2.

4.7.4 Comparison of the performance evaluation results

Table 8 and Table 11 in the Appendix report the total subscore (SS column) for the CVSS exploitability metrics, the values of the Structural Severity Reachability metric, and the availability of the exploits for the Apache HTTP server and Linux Kernel datasets, respectively. The ranges of CVSS exploitability metrics subscores are 1–10, whereas the values of the Structural Severity Reachability metric are R or NR. Figure 12 shows the distribution of the CVSS Exploitability subscore for Apache HTTP server and Linux Kernel datasets. Here, the population is split into two: vulnerabilities with CVSS exploitability subscore ≥ 6 are considered exploitable (positive test), and those <6 are considered not exploitable (negative test). The minimum and maximum exploitability subscores for Apache and Linux are 1.9 and 10, respectively, whereas the mean and the standard deviation for Apache are 8.3 and 1.9, respectively. However, the mean and the standard deviation for Linux are 4.8 and 2.4, respectively.

As can be seen from Fig. 12, most of the vulnerabilities do turn out to have a high Exploitability subscore for Apache, whereas a low Exploitability subscore for Linux. This is because of the Access Vector metric. The majority of the vulnerabilities in Apache is remotely accessed and hence has been assigned a high exploitability subscore (8 out of 10). On the other hand, most of the vulnerabilities in Linux are locally accessed and thus have been assigned a low exploitability subscore (3.9 out of 10). It should be noted that neither the Authentication metric nor the Access Complexity metric values (Low and Medium) have a significant effect on the exploitability total subscore for both datasets. However, when the Access Complexity metric value is high, it notably affects the exploitability total subscore. Nonetheless, only two out of 25 vulnerabilities in Apache dataset and five out of 86 vulnerabilities in Linux dataset have been found to have a high value Access Complexity.

Table 10 compares the two metrics using the Apache HTTP server and Linux datasets. From the table, we can observe that the Structural Severity Reachability metric performs better in terms of all measures than CVSS exploitability metrics for the Apache dataset.

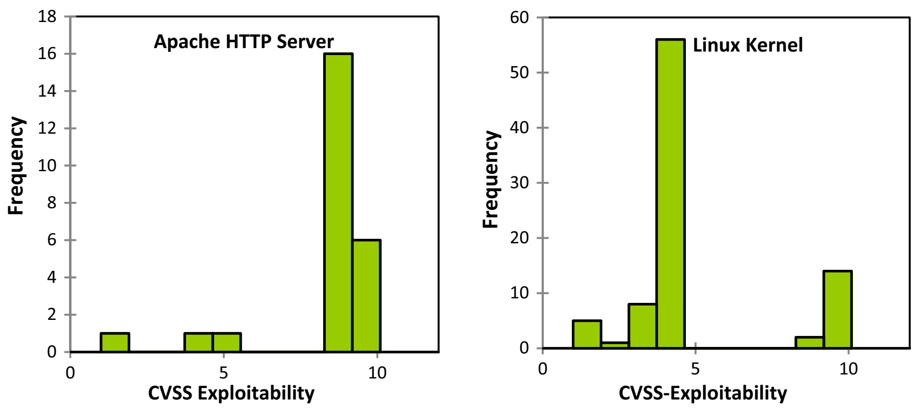


Fig. 12 Distribution of CVSS Exploitability Subscores for Apache HTTP server and Linux Kernel

However, in the Linux dataset, the Structural Severity Reachability metric performs much better than the CVSS exploitability metrics. It has been observed that when the software and the dataset size were changed, the Structural Severity Reachability metric performs better than CVSS exploitability metrics in terms of precision, F1- and F2-measures. This can be attributed to the fact that most of the vulnerabilities in the Linux Kernel require an attacker to have a local access to exploit them and that makes their exploitation harder, and hence they have been assigned a low exploitability subscore.

For the Apache dataset, the false-positive rate for the two metrics is 85.71 %, whereas the false-negative rate is 0 % for the Structural Severity metric and 0.09 % for the CVSS exploitability metrics. Having the 0.09 % false-negative rate is because the vulnerability CVE-2006-3747 has an exploit and the CVSS exploitability metrics predicted it as being not exploitable. On the other hand, for the Linux Kernel dataset the false-positive rate slightly fell for the Structural Severity metric (82 %), whereas it notably reduced for the CVSS exploitability metrics (41 %). This can be explained by the increase in the false-negative rate (89 %). This can be attributed to the fact that there are 86 vulnerabilities with an exploit in Linux Kernel that have been assigned a low exploitability subscore.

We argue that it is better to have more non-exploitable vulnerabilities inspected than to have one exploitable vulnerability being left unchecked. It should be noted that the Structural Severity measure captures the exploitability factor by using only one attribute, reachability, whereas the CVSS exploitability metrics uses three attributes, Access Complexity, Access Vector, and Authentication.

5 Discussion and threats to validity

5.1 Discussion of the case study results

Having an approach that does not depend on the availability of the security experts can be of great value. The proposed measure is based on software properties that reflect exploitability factors. We have observed that the proposed approach has not been found to be less restrictive than its counterpart, CVSS, except in two cases, when the vulnerabilities (CVE-2004-0488 and CVE-2007-5000) were found not reachable.

We have also observed that identifying critical functions can be accomplished by looking at the number of functions that are called by the vulnerable function. The more the functions called by the vulnerable function, the higher the effect if the vulnerable function is exploited. Figure 13 shows the *call graphs* of the vulnerability CVE-2009-1195. This

Table 10 Prediction Performance

Software	Performance measures	CVSS exploitability metrics (%)	Structural severity reachability metric (%)
Apache HTTP server	Sensitivity	90.91	100
	Precision	45.45	47.83
	F1-Measure	61.00	65.00
	F2-Measure	75.76	82.09
Linux Kernel	Sensitivity	10.90	100.00
	Precision	44.00	78.00
	F1-Measure	18.00	88.00
	F2-Measure	13.00	95.00

graph shows the number of functions that are called by the vulnerable function. This vulnerability impacts two functions as explained in subsection 5.3. Figure 13a shows the vulnerable function *process_resource_config_nofmatch* directly calls 10 functions, whereas the other vulnerable function *ap_process_config_tree* directly calls only two functions as shown in Fig. 13b. Bhattacharya et al. (2012) introduced a bug severity metric called Node-Rank. It measures the relative importance of a function or module in the function call or module graphs. It could be interesting if the same can be done for vulnerability severity so the risk of exploitation can be further illustrated.

Sparks et al. (2007) studied the penetration depth of reaching a node in a control flow graph. They found that the nodes at greater depths (>10 edges) become increasingly difficult to reach. In other words, it is hard to craft an input that can lead to such a node at a particular depth. If crafting an input that can reach a vulnerable statement for a single method is difficult, we believe that crafting an input to call a method containing a vulnerable statement from other methods could be even harder. If we further assume that the target system is a closed system, it gets even harder for the attackers to figure out the sequences of calls and inputs that are needed to trigger them. However, it has been also observed that the degree of a call depth of vulnerable functions varies among vulnerabilities. Some of the vulnerabilities have only one degree of depth, while others have 13. Figure 14 shows the degree of depth of the vulnerability CVE-2010-0434. Due to page size limitation, we have only showed the depth up to level 5. We believe that the depth degree of a vulnerable function can inform us about the difficulty of exploiting a vulnerability, because the more the functions an attacker has to invoke to reach a vulnerable function, the harder it is to get it exploited.

Even though most of the property extraction has been automated, it has been noticed that performing vulnerability exploitation assessment remained dependable on the human. For instance, looking at a table with 25 vulnerabilities and comparing the selected properties to decide the risk of a given vulnerability is fairly attainable. However, having a larger dataset can

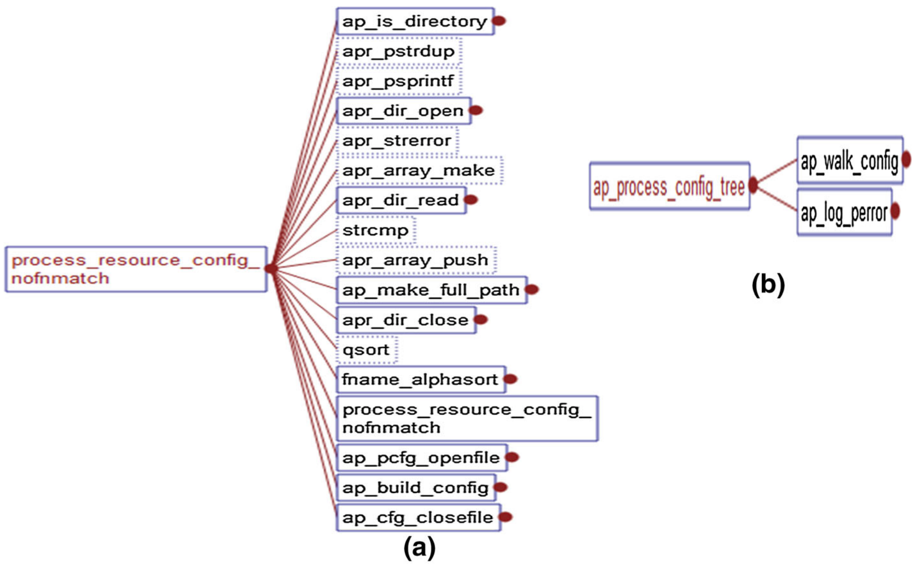


Fig. 13 Functions that are called by the vulnerable functions (Calls Graph)

be challenging and error prone. It would be helpful if this part of the process can be automated by using some techniques from machine learning. Having the exploitability properties extracted from the source code, the machine learning model can automatically assess the exploitability risk based on the data. This idea could also help in managing scalability as machine learning techniques have shown success in dealing with sizable data sets. Moreover, it also makes adding added number of exploitability properties (features) handy.

It has also been noted that even though a vulnerability reachability is an important factor of exploitation, considering the exploit mitigation guards at the function level could be a great addition. For instance, having an exploit guards such as GuardStack (GS) can prevent exploits from exploiting a vulnerability even if the exploit has called the vulnerable function. Sparks et al. (2007) stated that it is possible to identify functions that have not been compiled to use GS through the use of simple static analysis tools. This can be used as an estimator of exploitation impact. Thus, vulnerabilities (vulnerable functions) with exploit mitigation guards could be considered of a low risk.

5.2 Threats to validity

5.2.1 *External validity*

The main threat to external validity is choosing two datasets, the Apache HTTP server and Linux Kernel. However, Apache HTTP server, since its release in 1995, has gone through a number of improvements which led to the release of several versions: 1.3.x, 2.0.x, 2.2.x, 2.3.x, and 2.4.x. For the release 1.3.x alone, there are 47 versions, whereas for the release 2.2, there are 23 versions. The Apache HTTP server has a richer history of public available vulnerabilities, more than 169. Besides, its line of code varies between 50,712 LOC to 358,633 LOC. Moreover, Apache HTTP server has varieties of vulnerabilities: Denial of service (69), Execute Code (24), Overflow (16), XSS (21), Bypass Something (14), Gain Information (13), Directory Traversal (4), Gain Privilege (6), Memory Corruption (1), HTTP Response Splitting (1), and CSRF (1) (National Vulnerability Database 2013). Even though Apache HTTP server has mainly been developed using C/C++, languages such as

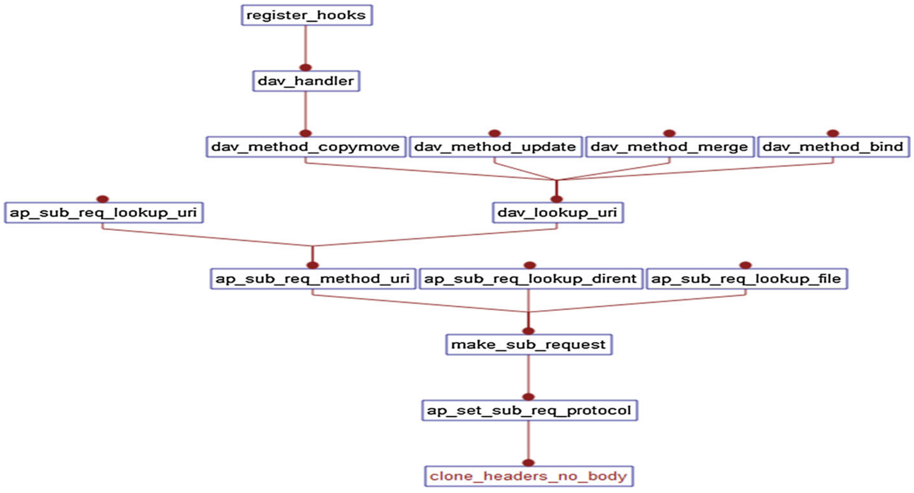


Fig. 14 Depth of a vulnerable function calls (Called By Graph)

HTML, JavaScript, Python, and XML have been used too. Web servers form a major component of the Internet, and they are targeted by attackers (more than 1,000 malicious HTTP requests in 6 months (Imperva 2012)). According to Usage Statistics and Market Share (2013), Apache Web server is used by over 64 % among most commonly used Web servers. On the other hand, Linux Kernel has larger number of vulnerabilities, more than 1200. Its size in line of code has ranged from 10,239 LOC to 15,803,499 LOC. It also has a greater variety of vulnerabilities: Denial of service (750), Execute Code (58), Overflow (213), Bypass Something (67), Gain Information (197), Directory Traversal (2), Memory Corruption (68), and Gain Privilege (149).

5.2.2 *Internal validity*

The main threat to internal validity is the chosen factors for measuring the risk of vulnerability exploitation: reachability and DSCs. As an exploit in its basic form is an input, a vulnerability has to be reachable in order for the input to be able to trigger it. Therefore, reachability can be considered as a major contributor. However, not all vulnerabilities that are reachable are exploitable and this is because of the degree of difficulty of reaching vulnerabilities. For instance, vulnerabilities that are hard to reach are also hard to exploit and should be given less priority. This could introduce false-positive results. Nevertheless, the degree of difficulty is dependent on whether a vulnerability is reachable or not. Once that is decided, we need to assess whether the code path to the vulnerable location is hard to reach or not. We recognize this threat, and we are investigating the possibility of objectively measuring the difficulty of reachability using the following: (1) the number of function calls an attacker has to invoke to reach the vulnerable code, (2) an authentication verification mechanisms along the path to the vulnerable location, and (3) the privilege required to invoke the vulnerable function.

Even though using DSCs help in eliminating subjectivity, we recognize that capturing the technical impact by only considering the existence of a DSC might not be sufficient. Factors such as privilege of the vulnerable function, exploit mitigation, and authentication could be a major contributor too. However, measuring these factors can introduce subjectivity. Ensuring objective measurements of these factors requires further research.

5.2.3 *Construct validity*

We used a commercial tool named Understand to automatically generate call graph between the functions. We also used SVN provided by the Apache project foundation to identify the vulnerabilities location. Moreover, we used the cflow and a python script to identify attack entry points. Usage of the third party tools and a python script represent potential threats to construct validity. We verified the results produced by these tools by manually checking randomly selected outputs produced by each tool.

Part of our measurement of exploitability uses the attack entry points as an estimate of the attack resources that an attacker can use to exploit vulnerabilities. Even though a significant concern resides in areas where the system obtains external data, there are vulnerabilities that can be exploited without sending data to the system. This represents a threat to construct validity. However, our vulnerabilities datasets have been collected from NVD that provides a list of the types of vulnerabilities that it uses. There are 19 types of vulnerabilities in the NVD. Based on their Common Weakness Enumeration (CWE) number and the description provided by the CWE and the NVD, we verified that the selected vulnerabilities are directly influenced by a user input.

6 Related work

In this section, we review the work related to vulnerability exploitation risk assessment. We organize this section based on the method used to assess exploitation risk into: measurement-based, model-based, test-based, and analysis-based approaches.

6.1 Measurement-based approaches

6.1.1 Attack surface metric

The attack surface notion was first introduced by Howard in his Relative Attack Surface Quotient metric (Howard et al. 2005). It was later formally defined by Manadhata and Wing (2011). They proposed a framework that included the notion of entry and exit points and the associated damage potential–effort ratio. They applied their formally defined metric to many systems, and the results show the applicability of the notion of the attack surface. Their new metric has been adapted by a few major software companies, such as Microsoft, Hewlett-Packard, and SAP. Manadhdata et al. (2006) related the number of reported vulnerabilities for two FTP daemons with the attack surface metric along the method dimension. Younis and Malaiya (2012) compared the vulnerability density of two versions of Apache HTTP server with the attack surface metric along the method dimension. However, attack surface metric does not measure the risk of exploitation of individual vulnerabilities. Rather, it measures the exploitability for the whole system, and as a result it cannot help in prioritizing among vulnerabilities. Besides, neither Manadhata et al. (2006) nor Younis and Malaiya (2012), however, related entry points with the location of the vulnerability to measure its exploitability.

6.1.2 CVSS metrics

CVSS metrics are the de facto standard that is currently used to measure the severity of vulnerabilities (Mell et al. 2007). CVSS base score measures severity based on the exploitability (the ease of exploiting vulnerability) and impact (the effect of exploitation). Exploitability is assessed based on three metrics: Access Vector, Authentication, and Access Complexity. However, CVSS exploitability measures have come under some criticism. First, they assign static subjective numbers to the metrics based on expert knowledge regardless of the type of vulnerability, and they do not correlate with the existence of known exploit (Bozorgi et al. 2010). Second, two of its factors (Access Vector and Authentication) have the same value for almost all vulnerabilities (Allodi and Massacci 2013). Third, there is no formal procedure for evaluating the third factor (Access Complexity) (Mell et al. 2007). Consequently, it is unclear whether CVSS considers the software structure and properties as a factor.

6.2 Model-based approaches

6.2.1 Probabilistic model

Joh and Malaiya (2011) formally defined a risk measure as a likelihood of adverse events and the impact of this event. On one hand, they utilized the vulnerability lifecycle and applied the Markov stochastic model to measure the likelihood of vulnerability exploitability for an individual vulnerability and the whole system. On the other hand, they used the impact related metrics from CVSS to estimate the exploitability impact. They applied their metric to assess the risk of two systems that had known unpatched vulnerabilities using actual data. However, the transition rate between vulnerability lifecycle events has not been determined, and the probability distribution of lifecycle events remains to be studied. Moreover, the probability of being in an exploit state requires information about the attacker behavior which might not be available. Additionally, the probability of a patch being available but not applied requires information about the administrator behavior, which has not been considered by the proposed model and also hard to be obtained. In contrast, we assess vulnerability exploitability for individual vulnerabilities based on the source code properties regardless of the availability or unavailability of a patch.

6.2.2 Logistic model

Vulnerability density metric assesses the risk of potential exploitation based on the density of the residual vulnerabilities (Alhazmi et al. 2007). The density of residual vulnerabilities is measured based on the number of known reported vulnerabilities and the total number of vulnerabilities. However, the total number of vulnerabilities is unknown but can be predicted using vulnerability discovery models (VDMs). Alhazmi and Malaiya (2005) proposed a logistic vulnerability discovery model, termed the AML model. AML examines the reported known vulnerabilities of a software system to estimate the total number of vulnerabilities and their rate of discovery. However, considering the number of vulnerabilities alone is insufficient in assessing the risk of an individual vulnerability exploitation. Because different vulnerabilities have different opportunity of being exploited based on their properties such as reachability.

6.2.3 Machine learning-based metric

Bozorgi et al. (2010) aimed at measuring vulnerability severity based on likelihood of exploitability. They argued that the exploitability measures in CVSS base score metric cannot tell much about the vulnerability severity. They attributed that to the fact that CVSS metrics rely on expert knowledge and static formula. To that end, the authors proposed a machine learning and data mining technique that can predict the possibility of vulnerability exploitability. They observed that many vulnerabilities have been found to have a high severity score using CVSS exploitability metric although there were no known exploits existing for them. This indicates that the CVSS score does not differentiate between exploited and non-exploited vulnerabilities. This result was also confirmed by (Allodi and Massacci 2012, 2013; Allodi et al. 2013). However, unlike their work, ours relies on software properties such as the attack surface entry points, the source code structure, and the vulnerabilities location to estimate vulnerability exploitability. This is particularly important for newly released applications that do not have a large amount of historical vulnerabilities.

6.3 Test-based approaches (proof of concept)

6.3.1 Automated exploit-generation system (AEG)

Avgerinos et al. (2014) proposed an automated exploit-generation system (AEG) to assess the risk of vulnerability exploitation. AEG first uses the static analysis to find a potential bug locations in a program, and then uses a combination of static and dynamic analysis to find an execution path that reproduces the bug, and then generates an exploit automatically. AEG generates exploits, which provides an evidence that the bugs it finds are critical security vulnerabilities. However, generating an exploit is expensive and does not scale. AEG has only been applied to a specific type of vulnerabilities and software.

6.3.2 Black box fuzz testing

Sparks et al. (2007) extended the black box fuzzing using a genetic algorithm that uses the past branch profiling information to direct the input generation in order to cover specified program regions or points in the control flow graph. The control flow is modeled as Markov process, and a fitness function is defined over Markov probabilities that are associated with a state transition on the control flow graph. They generated inputs using the grammatical evolution. These inputs are capable of reaching deeply vulnerable code, which is hidden in a hard to reach locations. In contrast to their work, ours relies on the source code analysis, a link between vulnerability location and attack surface entry points, and the DSC analysis that were specifically intended for measuring vulnerability exploitability.

6.4 Analysis-based approaches

6.4.1 Black market data analysis

Allodi and Massacci (2012, 2013) proposed the black market as an index of risk of vulnerability exploitation. Their approach assesses the risk of vulnerability exploitation based on the volumes of the attacks coming from the vulnerability in the black market. It first looks at the attack tools and verifies whether the vulnerability is used by such tool or not. It also analyzes the attacks on the wild to verify whether the vulnerability has been a target of such attacks or not. If the vulnerability is being used by one of the attack tools or being a target of real attacks, they consider this vulnerability as a threat for exploitation. This approach has introduced a new view of measuring the risk of exploitation by considering the history of attacks at the vulnerabilities. This approach does not require spending large amount of technical resources to thoroughly investigate the possibility of vulnerability exploitation. However, this approach requires a vulnerability intelligence provider as the information about the attacks and tools are dynamic in nature. Moreover, if the vulnerability right now is not used by a tool or it is not a target of an attack, it does not mean that it is going to be so continually. Our approach, on the other hand, relies only on software properties and does not make any assumption about the attacks and attackers' resources.

6.4.2 Source code analysis

Brenneman (2012) introduced the idea of linking the attack surface entry point to the attack target to prioritize the effort and resource required for software security analysis. Their approach is based on the path-based analysis, which can be utilized to generate an attack map. This helps visualizing the attack surfaces, attack target, and functions that link them. This is believed to make a significant improvement in software security analysis. In contrast to their work, we do not only utilize the idea of linking the attack surface entry point with the reported vulnerability location to estimate vulnerability exploitability, but also check for the DSCs inside every related entry point to estimate the impact of exploitation. The use of the DSCs is helpful for inferring an attacker's motive in invoking the entry point method.

6.4.3 System calls analysis

Massimo et al. (2002) presented a detailed analysis of the UNIX system calls and classify them according to their level of threat with respect to the system penetration. To control these system calls invocation, they proposed the Reference Monitor for UNIX System (REMUS) mechanism to detect an intrusion that may use these system calls, which could subvert the execution of privileged applications. Nevertheless, our work applies their idea to estimate the impact of exploitation, as attackers usually look to cause more damage to targeted systems. Thus, our work is not about intrusion detection but rather measuring the exploitability of a known vulnerability.

7 Conclusion and future work

Assessing the severity of a vulnerability requires evaluating the potential risk. Existing measures do not consider software access complexity and tend to rely on subjective judgment. In this paper, we have proposed an approach that uses system related attributes such as attack surface entry points, vulnerability location, call function analysis, and the existence of DSCs. This approach requires us to examine some of the structural aspects of security such as the paths to the vulnerable code starting from the entry points. We have demonstrated the applicability of the proposed approach and have compared resulting measures with overall CVSS severity metrics. Our results show that this approach, involving assessment of the system security based on systematic evaluation and not subjective judgment, is feasible.

While the main parts of the analysis have been automated, providing a framework that can automate the entire analysis will be helpful in reducing the effort. We plan to develop techniques to reduce human involvement and thus enhance scalability in assessing exploitability risk by using machine learning techniques. We plan to examine the effectiveness of machine learning for automatically assessing the risk of vulnerability exploitation using the proposed properties as features. Given a vulnerable function and their exploitability features, the machine learning model can predict whether it is an exploitable function and estimate the impact of its exploitation.

Even though measuring the possibility of reaching a vulnerability is important, quantifying the degree of difficulty of reaching a vulnerability is also valuable for comparing the severity among similar vulnerabilities, and thus needs to be examined. We plan to utilize the idea of *the function call graph depth* which has been presented in the discussion section. Devising a way of estimating the impact of reachable vulnerabilities will be valuable for estimating the overall risk of individual vulnerabilities and the whole system. We plan to further study the Node-Rank proposed by Bhattacharya et al. (2012) as an estimator of the vulnerability impact. Finally, identifying whether a vulnerable function is guarded by security control can help better understand the impact of exploitation. We intend to study how function exploitation properties proposed by Skape (2007) can give more information about the risk of vulnerability exploitation.

Appendix

See Table 11.

Table 11 Obtained structural severity metrics compared to CVSS metrics of Linux Kernel dataset

No	Vulnerability	Structural severity			CVSS									Exploit existence
		Reachability	DSC	Severity	Exploitability				Impact				Severity	
					AC	AV	AU	SS	C	I	A	SS		
1	CVE-2002-0499	R	DSC	H	L	L	NR	3.9	N	P	N	2.9	L	EE
2	CVE-2003-0462	R	NDSC	M	H	L	NR	1.9	N	N	P	2.9	L	EE
3	CVE-2003-0619	R	NDSC	M	L	N	NR	10	N	N	P	2.9	M	EE
4	CVE-2003-0961	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
5	CVE-.2003-0985	R	NDSC	M	L	L	NR	3.9	C	C	C	10	H	EE
6	CVE-2004-0424	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
7	CVE-2004-1016	R	DSC	H	L	L	NR	3.9	N	N	P	2.9	L	EE
8	CVE-2004-1073	R	DSC	H	L	L	NR	3.9	P	N	N	2.9	L	EE
9	CVE-2004-1137	R	DSC	H	L	N	NR	10	C	C	C	10	H	EE
10	CVE-2004-1235	R	DSC	H	H	L	NR	1.9	C	C	C	10	M	EE
11	CVE-2004-1333	R	DSC	H	L	L	NR	3.9	N	N	P	2.9	L	EE
12	CVE-2005-0736	R	DSC	H	L	L	NR	3.9	N	P	N	2.9	L	EE
13	CVE-2005-0750	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
14	CVE-2005-1263	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
15	CVE-2005-1589	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
16	CVE-2005-2709	R	NDSC	M	L	L	NR	3.9	P	P	P	6.4	M	EE
17	CVE-2005-2973	R	NDSC	M	L	L	NR	3.9	P	N	N	2.9	L	EE
18	CVE-2005-3257	R	DSC	H	L	L	NR	3.9	P	P	P	6.4	M	EE
19	CVE-2005-3807	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
20	CVE-2005-3808	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
21	CVE-2005-3857	R	NDSC	M	L	L	NR	3.9	N	N	C	6.9	M	EE
22	CVE-2006-2444	R	DSC	H	L	N	NR	10	N	N	C	6.9	H	EE

Table 11 continued

No	Vulnerability	Structural severity			CVSS									Exploit existence
		Reachability	DSC	Severity	Exploitability				Impact				Severity	
					AC	AV	AU	SS	C	I	A	SS		
23	CVE-2006-2451	R	DSC	H	L	L	NR	3.9	P	P	P	6.4	M	EE
24	CVE-2006-2629	R	DSC	H	H	L	NR	1.9	N	N	C	6.9	M	EE
25	CVE-2006-5757	R	NDSC	M	H	L	NR	1.9	N	N	P	2.9	L	EE
26	CVE-2007-1000	R	NDSC	M	L	L	NR	3.9	N	N	C	10	H	EE
27	CVE-2007-1357	R	DSC	H	L	N	NR	10	N	N	C	6.9	H	EE
28	CVE-2007-1388	R	NDSC	M	M	L	SS	2.7	N	N	C	6.9	M	EE
29	CVE-2007-1730	R	DSC	H	L	L	NR	3.9	C	N	C	9.2	M	EE
30	CVE-2007-1861	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
31	CVE-2008-4113	R	DSC	H	M	L	NR	3.9	C	N	N	6.9	M	EE
32	CVE-2008-4302	R	NDSC	M	L	L	NR	3.9	N	N	C	6.9	M	EE
33	CVE-2008-5713	R	NDSC	M	L	L	NR	3.9	N	N	C	6.9	M	EE
34	CVE-2009-0065	R	NDSC	M	L	A	NR	10	C	C	C	10	H	EE
35	CVE-2009-0676	R	NDSC	M	L	L	NR	3.9	P	P	N	2.9	L	EE
36	CVE-2009-0746	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
37	CVE-2009-1337	R	DSC	H	M	L	NR	3.4	P	P	P	6.4	M	EE
38	CVE-2009-1897	R	DSC	H	M	L	NR	3.4	C	C	C	10	M	EE
39	CVE-2009-1914	R	NDSC	M	L	L	NR	3.9	N	N	C	6.9	M	EE
40	CVE-2009-1961	R	NDSC	M	M	L	NR	3.4	N	N	P	2.9	L	EE
41	CVE-2009-2692	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
42	CVE-2009-2698	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
43	CVE-2009-2847	R	DSC	H	L	L	NR	3.9	C	N	N	6.9	M	EE
44	CVE-2009-3001	R	DSC	H	L	L	NR	3.9	C	N	N	6.9	M	EE
45	CVE-2009-3002	R	DSC	H	L	L	NR	3.9	C	N	N	6.9	M	EE

Table 11 continued

No	Vulnerability	Structural severity			CVSS									Exploit existence
		Reachability	DSC	Severity	Exploitability				Impact				Severity	
					AC	AV	AU	SS	C	I	A	SS		
46	CVE-2009-3043	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
47	CVE-2009-3234	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
48	CVE-2009-3613	R	NDSC	M	L	A	NR	10	N	N	C	6.9	H	EE
49	CVE-2009-3888	R	NDSC	M	L	L	NR	3.9	N	N	C	6.9	M	EE
50	CVE-2010-1173	R	NDSC	M	M	N	NR	8.6	N	N	C	6.9	H	EE
51	CVE-2010-1636	R	DSC	H	L	L	NR	3.9	P	N	N	2.9	L	EE
52	CVE-2010-2959	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
53	CVE-2010-3081	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
54	CVE-2010-3437	R	DSC	H	L	L	NR	3.9	C	N	C	9.2	M	EE
55	CVE-2010-3858	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
56	CVE-2010-4073	R	DSC	H	M	L	NR	3.4	P	N	N	2.9	L	EE
57	CVE-2010-4165	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
58	CVE-2010-4258	R	DSC	H	H	L	NR	1.9	C	C	C	10	M	EE
59	CVE-2011-2918	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	EE
60	CVE-2012-3375	R	NDSC	M	L	L	NR	3.9	N	N	C	6.9	M	EE
61	CVE-2013-1959	R	NDSC	M	L	L	NR	3.9	N	N	C	6.9	M	EE
62	CVE-2013-2094	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	EE
63	CVE-2014-0038	R	DSC	H	M	L	NR	3.4	C	C	C	10	M	EE
64	CVE-2014-2851	R	DSC	H	M	L	NR	3.4	C	C	C	10	M	EE
65	CVE-2002-0499	R	NDSC	M	L	L	NR	3.9	N	P	N	2.9	L	NEE
66	CVE-2002-1572	R	DSC	H	L	N	NR	10	C	C	C	10	H	NEE
67	CVE-2003-0986	R	DSC	H	L	L	SS	3.1	N	N	P	2.9	L	NEE
68	CVE-2004-0010	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	NEE

Table 11 continued

No	Vulnerability	Structural severity			CVSS									Exploit existence
		Reachability	DSC	Severity	Exploitability				Impact				Severity	
					AC	AV	AU	SS	C	I	A	SS		
69	CVE-2004-0394	R	DSC	H	L	L	NR	3.9	P	N	N	2.9	L	NEE
70	CVE-2004-1070	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	NEE
71	CVE-2004-1151	R	DSC	H	L	L	NR	3.9	C	C	C	10	H	NEE
72	CVE-2004-2731	R	DSC	H	L	L	NR	3.4	P	P	P	6.4	M	NEE
73	CVE-2005-0124	NR	NDSC	L	L	L	NR	3.9	N	N	P	2.9	L	NEE
74	CVE-2005-0449	R	DSC	H	M	N	NR	8.6	N	N	C	6.9	H	NEE
75	CVE-2005-0531	R	NDSC	M	L	L	NR	3.9	N	P	N	2.9	L	NEE
76	CVE-2005-2456	R	DSC	H	L	L	NR	3.9	N	N	P	2.9	L	NEE
77	CVE-2005-2459	R	NDSC	M	L	N	NR	10	N	N	P	2.9	M	NEE
78	CVE-2005-2492	NR	DSC	L	L	L	NR	3.9	P	N	P	4.9	L	NEE
79	CVE-2005-2500	NR	DSC	L	L	N	NR	10	P	P	P	6.4	H	NEE
80	CVE-2005-3119	R	NDSC	M	L	L	NR	3.9	P	N	N	2.9	L	NEE
81	CVE-2005-3808	R	DSC	H	L	L	NR	3.9	N	N	C	6.9	M	NEE
82	CVE-2005-3848	R	DSC	H	L	N	NR	10	N	N	C	6.9	H	NEE
83	CVE-2005-3858	R	DSC	H	L	N	NR	10	N	N	C	6.9	H	NEE
84	CVE-2005-4635	R	DSC	H	L	N	NR	10	N	N	P	2.9	M	NEE
85	CVE-2006-6106	R	NDSC	M	L	N	NR	10	P	P	P	6.4	H	NEE
86	CVE-2006-6333	R	DSC	H	L	N	NR	10	N	N	c	6.9	H	NEE

EE exploit exist, *NEE* no exploit exist, *SS* subscore

References

Alhazmi, O. H., & Malaiya,Y. K. (2005). Modeling the vulnerability discovery process. In: *Proceedings of the 16th IEEE international symposium on software reliability engineering (ISSRE'05)* (pp. 1–10). doi:10.1109/ISSRE.2005.30.

Alhazmi, O. H., Malaiya, Y. K., & Ray, I. (2007). Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security, 26*(3), 219–228. doi:10.1016/j.cose.2006.10.002.

Allodi, L., & Massacci, F. (2012). A preliminary analysis of vulnerability scores for attacks in wild. In: *Proceedings of the 2012 ACM workshop on Building analysis datasets and gathering experience returns for security (BADGERS 12)* (pp. 17–24). ISBN: 978-1-4503-1661-3. doi:10.1145/2382416.2382427

Allodi, L., & Massacci, F. (2013). My Software has a vulnerability, should I worry? Cornel Univsity Library (pp. 12). arXiv:1301.1275. http://www.arxiv.org/pdf/1301.1275v3.pdf. Accessed 2 Aug 2013.

Allodi, L., Shim, W., & Massacci, F. (2013). Quantitative Assessment of risk reduction with cybercrime black market monitoring. IEEE Security and Privacy Workshops (SPW) (pp. 165–172). doi: 10.1109/SPW.2013.16

Apache-SVN. (2014). the apache software foundation. http://www.svn.apache.org/viewvc/. Accessed 27 Mar 2014.

Arbaugh, W. A., Fithen, W. L., & John, M. (2000). Windows of vulnerability: A case study analysis. *Computer, 33*(12), 52–59. doi:10.1109/2.889093.

Archive.apache.org. (2014). The apache software foundation. http://www.archive.apache.org/dist/httpd/. Accessed 2 Aug 2014.

Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M., & Brumley, D. (2014). Automatic exploit generation. *Communications of the ACM, 26*(3), 74–84. doi:10.1145/2560217.2560219.

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. 419–429). ISBN: 978-1-4673-1067-3.

Bozorgi, M., Saul, L. K., Savage, S., & Voelker, G. M. (2010). Beyond heuristics: Learning to classify vulnerabilities and predict exploits. In: *Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining (KDD '10)* (pp. 105–114). doi:10.1145/1835804.1835821

Brenneman, D. (2012). Improving software security by identifying and securing paths linking attack surface to attack target. McCabe Software Inc. White Paper. http://www.mccabe.com/. Accessed 4 Aug 2014.

Evans, D., & Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software, 19*(1), 42–51. doi:10.1109/52.976940.

Ferrante, J., Ottenstein, K. J., & Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS), 9*(3), 319–349. doi:10.1145/24039.24041.

Frei, S., Tellenbach, B., & Plattner, B. (2008). 0-day Patch: Exposing vendors (in) security performance. Black Hat Europe. http://www.techzoom.net/papers/blackhat 0 day Patch 2008.pdf. Accessed 10 Aug 2013.

GNU Cflow (2013) http://www.gnu.org/software/cflow/manual/cflow.html. Accessed 2 Aug 2013.

Horwitz, S., Reps, T., & Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS), 12*(1), 26–60. doi:10.1145/77606.77608.

Howard, M., Pincus, J., & Wing, J. (2005). Measuring relative attack surfaces. Computer Security in the 21st Century (pp. 109–137). Springer. ISBN 0-387-24005-5, 0-387-24006-3. http://www.link.springer.com/chapter/10.1007/0-387-24006-3_8.

Imperva, a provider of cyber and data security products (2012). http://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed3.pdf. Accesses 19 Apr 2014.

Jansen, W. (2009). Directions in Security Metrics Research. NIST. http://www.csrc.nist.gov/publications/nistir/ir7564/nistir-7564_metrics-research.pdf. Accessed 15 March 2013.

Joh, H., & Malaiya, Y. K. (2011). Defining and assessing quantitative security risk measures using vulnerability lifecycle and CVSS metrics. In: *The 2011 international conference on security and management (SAM'11)* (pp. 10–16).

Kuck, D. J., Muraoka, Y., & Chen, S. (1972). On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *The IEEE Transactions on Computers, 100*(12), 1293–1310. doi:10.1109/T-C.1972.223501.

Manadhata, P. K., & Wing, J. M. (2011). An attack surface metric. *The IEEE Transactions on Software Engineering, 37*(3), 371–386. doi:10.1109/TSE.2010.60.

Manadhata, P. K, Wing, J., Flynn M., & McQueen, M. (2006). Measuring the attack surfaces of two FTP daemons. In: *Proceedings of the 2nd ACM workshop on quality of protection (QoP'06)* (pp. 3–10). doi:10.1145/1179494.1179497.

Massimo, B., Gabrielli, E., & Mancini, L. (2002). Remus: A security-enhanced operating system. *ACM Transactions on Information and System Security (TISSEC), 5*(1), 36–61. doi:10.1145/504909.504911.

Mell, P., Scarfone, K., & Romanosky, S. (2007). A complete guide to the common vulnerability scoring system version 2.0. Published by FIRST-Forum of Incident Response and Security Teams (pp. 1–23). http://www.first.org/cvss/cvss-guide.pdf. Accessed 15 Mar 2013.

Metasploit Database. (2014). http://www.metasploit.com/. Accessed 27 March 2014.

National Vulnerability Database. (2013). http://www.nvd.nist.gov/. Accessed 2 Aug 2013.

OSVDB: Open Sourced Vulnerability Database. (2014). http://www.osvdb.org/. Accessed 19 Feb 2014.

Pfleeger, C. P., & Pfleeger, S. L. (2006). *Security in computing*. New Jersey: Prentice Hall PTR.

Ponemon Institute. (2013). 2013 Cost of data breach study: Global analysis. Benchmark research sponsored by Symantec, Independently Conducted by Ponemon Institute. https://www4.symantec.com/mktginfo/whitepaper/053013_GL_NA_WP_Ponemon-2013-Cost-of-a-Data-Breach-Report_daiNA_cta72382.pdf. Accessed 10 Mar 2013.

Red Hat Bugzilla Main Page. (2014). https://bugzilla.redhat.com/. Accessed 2 Mar 2014.

Scientific Toolworks Understand. (2014). http://www.scitools.com/. Accessed 22 Mar 2014.

SecurityFocus. (2015). http://www.securityfocus.com/archive/1. Accessed 2 Mar 2015.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2009). Operating system concepts. Wiley.

Skape. (2007). Improving software security analysis using exploitation properties. Uninformed. http://www.uninformed.org/?o=about. Accessed 29 Mar 2014.

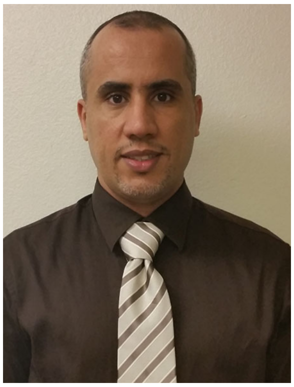
Sparks, S., Embleton, S., Cunningham, R., & Zou, C. (2007). Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In: Computer Security Applications Conference (ACSAC 2007) (pp. 477–486). doi:10.1109/ACSAC.2007.27.

Stoneburner, G., Goguen, A., & Feringa, A. (2002). Risk management guide for information technology systems. NIST. http://www.security-science.com/pdf/risk-management-guide-for-information-technology-systems.pdf. Accessed 23 Mar 2013.

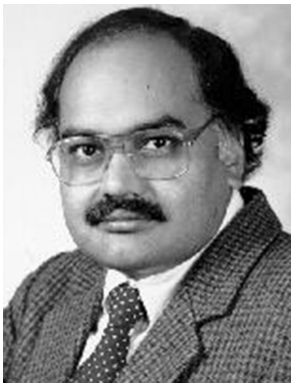
The Exploits Database. (2013). http://www.exploit-db.com/. Accessed 7 Aug 2013.

Usage Statistics and Market Share of Web Servers for Websites. (2013). http://www.w3techs.com/technologies/overview/web_server/all. Accessed 2 Aug 2013.

Younis, A. A., & Malaiya,Y. K. (2012). Relationship between attack surface and vulnerability density: A case study on apache HTTP server. In: *The 2012 international conference on internet computing (ICOMP'12)* (pp. 197–203).



Awad Younis is a Ph.D. student in the Computer Science Department at Colorado State University. His research interests are software security metrics, attack surface analysis, vulnerability analysis, exploitation risk analysis, and modeling the discovery process for security vulnerabilities. He received an M.Sc. in computer science from University of Madras, Chennai, India, in 2008, M.C.S in computer science from Colorado State University 2012, and BSc in computer science from Garyounis University, Benghazi, Libya, in 2000.



Yashwant K. Malaiya is a professor in the Computer Science Department at Colorado State University. He received M.S. in Physics from Sagar University, MScTech in Electronics from BITS Pilani, and PhD in Electrical Engineering from Utah State University. He has published more than 160 papers in the areas of fault modeling, software and hardware reliability, testing and testable design, and quantitative security risk evaluation. He served as the General Chair of 1993 and 2003 IEEE International Symposium on Software Reliability Engineering (ISSRE). He co-edited the IEEECS Technology Series book Software Reliability Models, Theoretical Developments, Evaluation and Applications. He is a recipient of the IEEE Third Millennium Medal and the IEEE Computer Society Golden Core award.

Indrajit Ray is an associate professor in the Computer Science Department at Colorado State University. His main research interests are in the areas of data and application security, network security, attack modeling, risk management, trust models, privacy, and digital forensics. At Colorado State University, he is member of the Data and Applications Security Group, the Network Security Group, and the Software Assurance Laboratory. He was one of the founding members and the first Chair of the IFIP TC-11 Working Group 11.9 on Digital Forensics. He has served on numerous conference program committees as well as organizing committees, on grant review panels, and as external evaluator of academic programs. He is currently serving on the editorial board of three international journals. He is a member of IEEE Computer Society, ACM, ACM Special Interest Group on Security Audit and Control, IFIP WG 11.3 on Data and Applications Security and IFIP WG 11.9 on Digital Forensics.