SOFTWARE VULNERABILITIES: LIFESPANS, METRICS, AND CASE STUDY

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Jason L. Wright

May 2014

Wednesday 16th April, 2014 13:42

Major Professor: Milos Manic, Ph.D.

AUTHORIZATION TO SUBMIT THESIS

This thesis of Jason L. Wright, submitted for the degree of Master of Science with a major in Computer Science and titled "Software Vulnerabilities: Lifespans, Metrics, and Case Study," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor ____________________ Date __________
Milos Manic, Ph.D.

Committee Members ____________________ Date __________
Miles McQueen, M.S.

____________________ Date __________
George Dinolt, Ph.D.

Department Administrator ____________________ Date __________
Gregory Donohoe, Ph.D.

Discipline's College Dean ____________________ Date __________
Larry Stauffer, Ph.D.

Final Approval and Acceptance by the College of Graduate Studies

____________________ Date __________
Jie Chen, PhD

ABSTRACT

It is difficult for end-users to judge the risk posed by software security vulnerabilities. This thesis examines three aspects of the software security vulnerability ecosystem to determine if commonly used metrics are based on sound engineering principals.

First, the decision by several security research firms to decrease the grace period before publicly releasing vulnerability details was examined. No evidence was found suggest that periods less than 6 months are effective.

Second, two new metrics are presented which are more easily computed, repeatable, and verifiable than previous metrics. Both metrics provide the ability to compare software packages based on number of vulnerabilities and vendor response time.

Third, metrics based strictly on known vulnerabilities are brought into question. The number of bugs which represent vulnerabilities is estimated for a particular package and the estimated number of resulting vulnerabilities is found to be far greater than the currently known vulnerabilities.

ACKNOWLEDGMENTS

This thesis would would not have been possible without the help of my advisors, Dr. Milos Manic and Miles McQueen. I would also like to thank Dr. George Dinolt for being a valuable addition to my committee at the last minute, Debbie McQueen in particular for gathering the ZDI and iDefense datasets in Chapter 3, Lawrence Wellman for his analysis help in Chapter 4, Jason Larsen for his help in turning bugs to vulnerabilities in Chapter 5, and Cindy Gentillon for helping with some of the thornier bits of the statistics.

On a personal note, thesis would not have happened without the support of my loving wife, Virginia Wright, and two wonderful daughters, Elizabeth and Phoebe.

TABLE OF CONTENTS

ABSTRACT . . . iii

ACKNOWLEDGMENTS . . . iv

TABLE OF CONTENTS . . . v

CHAPTER

1 Introduction . . . 1

2 Background . . . 3

2.1 Software Bugs and Vulnerabilities . . . 3

2.2 Vulnerability Stakeholders . . . 5

2.3 Software Vulnerability Disclosure . . . 8

2.4 Related Work . . . 9

3 Justification of Vulnerability Deadlines . . . 13

3.1 Introduction . . . 13

3.2 Overview of Vulnerability Disclosure . . . 16

3.3 Grace Periods Compared to Vulnerability Lifespans . . . 17

3.4 Did Vendors Speed Up Their Patch Creation . . . 19

3.5 Conclusion . . . 25

4 Analysis of Two End-User Vulnerability Metrics . . . 26

4.1 Introduction . . . 26

4.2 Two End-User Exposure Metrics . . . 28

4.3 Metrics Case Study . . . 30

4.4 Discussion . . . 43

4.5 Conclusion . . . 48

5 Estimating Software Vulnerabilities 52

5.1 Introduction 52

5.2 Hidden Impact Bugs 53

5.3 Experimental Goals and Setup 58

5.4 Experimental Results 69

5.5 Discussion 72

5.6 Conclusion 78

6 Conclusion 80

REFERENCES 81

APPENDIX

A Identifiers of Sampled Bugs for Each Scoring Group 88

B Published Works 92

C Copyright Information 99

CHAPTER 1

Introduction

The complexity of computing systems is ever increasing, and with each passing day it becomes harder for end-users to judge the risk posed by vulnerabilities in software. The impacts of exploitation of modern software system vulnerabilities include simple denial of service attacks, identity theft or destruction of physical devices. Denial of service attacks are temporary in nature with no long term impact once resolved, however, recovering from identity theft is difficult for the end user. The cost of damage to physical devices can be millions of dollars.

The Stuxnet malware is an example of the malware that was specifically designed to target critical infrastructure components and cause physical harm to attached devices. It is speculated that the Stuxnet malware set the Iranian nuclear program back years [1, 2]. One of the most important aspects of Stuxnet is that it exploited several previously unknown software vulnerabilities, and there is no reason to believe that other critical infrastructure systems are immune to similar attacks.

Stephan Frei, et al. describe the "security ecosystem" as being composed of "a wide variety of actors and processes" which effect the "security of information technology and computer networks" [3]. The subset of the security ecosystem involving vulnerabilities introduced during the development of software will be referred to as the "software vulnerability ecosystem" or more succinctly "vulnerability ecosystem."

This thesis examines several under-studied aspects of the software vulnerability ecosystem to determine if currently used decisions and metrics are based on sound engineering principals. To accomplish this, several experiments were conducted and the results are accumulated here.

The rest of this thesis is organized as follows:

Chapter 2 provides background information. The various actors and processes in the software vulnerability ecosystem are defined. Related work of general value as well

as analysis techniques are also discussed.

Chapters 3–5 are case studies which examine aspects of the software vulnerability ecosystem. Each chapter provides unique insights into the current state of software security.

Chapter 3 analyzes the impact of a change in the grace period offered by vulnerability research organizations on vendor behavior. It has been supposed that shorter grace periods force vendors to release patches more quickly. This chapter examines the practical effects of changes in grace period on patch creation and demonstrates that the above supposition appears to be correct, within limits.

Chapter 4 defines two software vulnerability metrics: Median Active Vulnerabilities (MAV) and Vulnerability Free Days (VFD) capture some of the behavior vulnerability researchers reporting vulnerabilities and the associated vendor response. Previous metrics did not illuminate the process of vulnerability patch creation and allow for comparison across vendors and products.

Chapter 5 introduces the concept of hidden impact bugs. These vulnerabilities are ones which are first reported as bugs, but their security impact is revealed some time after being reported and fixed as a bug. The bug database for an open source product is then analyzed to identify bugs likely to be vulnerabilities and then try to turn the identified bugs into vulnerabilities. The primary goal of this study was to determine whether statistics based on the known vulnerabilities in the National Vulnerability Database or other public repositories are on a sound foundation. If the number of known vulnerabilities is far less than the number of unknown vulnerabilities, it calls into question whether comparisons between products can be made using statistics based on known vulnerabilities.

Finally, Chapter 6 concludes the thesis and provides avenues for future work.

CHAPTER 2

Background

Software vulnerabilities have been around as long as software itself, but they have gained an elevated status particularly over the past 20 years with the growing interconnectedness of computers and the amount of sensitive data stored on them. For this research:

> a *software vulnerability* is an instance of [a mistake] in the specification, development, or configuration of software such that its execution can violate the [explicit or implicit] security policy.

This definition comes originally from Ivan Krsul [4] and later modified by Andy Ozment [5], who added the parenthetical modifiers.

The presence of mistakes that constitute vulnerabilities is primarily the result of the human processes by which software is created. The study of software flaws falls into the much larger field of software engineering or even more broadly into engineering as a whole. However, this thesis is concerned with the flaws that can lead to compromise of the security of a software system. In particular, the research case studies that constitute this thesis provide insights into estimating how many software flaws are in fact vulnerabilities.

2.1 Software Bugs and Vulnerabilities

This thesis is concerned with software "bugs" that have a security impact. A "bug" is an instance of a mistake in the specification, development, or configuration of software such that its execution produces an incorrect result or causes the software to behave in an unintended manner. The key difference between a software bug and a software vulnerability is whether the "[explicit or implicit] security policy" of the program is violated.

By way of example, a computer program which computes: $2+2=5$ obviously has a bug; whether this bug is a vulnerability depends on the context of the calculation within the program. In fact, one class of vulnerability, integer overflows, takes advantage of this

type of miscalculation. In an integer overflow, a mathematical operation is applied to two numbers and the result exceeds the maximum representable integer. By carefully controlling the inputs, attackers may be able to cause the execution arbitrary code in a vulnerable program.

During the development of products such as the Linux kernel and the MySQL database, periodic releases are made and then the developers split into branches: one set of developers maintains the stable code base producing patches in response to user and internal bug reports, and the other set of developers concentrate on major new functionality which will then become the next release.

As bugs are reported, they are triaged; some reports are erroneous, some reports are requests for additional functionality, and some reports are software flaws. Within the set of bugs reports that are flaws, a subset of those are security vulnerabilities.

There is a special class of bugs-turned-vulnerabilities that are most interesting to this work. These are bugs that are only discovered to be vulnerabilities substantially after they are patches are made available. The first authors to study these bugs were Arnold, et al. who examined these so called hidden impact vulnerabilities in the Linux kernel over a 3 year period [6]. They found that there were a significant quantity of hidden impact vulnerabilities bugs and that third party vendors such as RedHat took longer to incorporate and distribute non-security related patches to their users than those identified as vulnerabilities when they were reported. The conclusion was that the quicker the status of vulnerability was assigned to a bug, the quicker a patch is supplied, but significant vulnerabilities would fail to be incorporated with this policy. To put a more fine point on the terminology, this thesis will refer to bugs in this category as "hidden impact bugs" since they have already been identified as bugs and their true impact is as yet unknown at the time a patch is created.

2.2 Vulnerability Stakeholders

It is important to consider the various stakeholders in software vulnerabilities because each stakeholder observes different effects as vulnerabilities are discovered, reported, and then mitigated. This section discusses each stakeholder in the process.

There are four primary stakeholders in the vulnerability disclosure process:

- vendors who produce software products,
- vulnerability researchers: individuals or firms, who actively search for vulnerabilities or buy them, and then report the vulnerability to the vendor,
- end-users: enterprises or individuals, who are confronted with the potential for loss from vulnerabilities.
- vulnerability repositories: who collect and disseminate the data required for accurate calculation of metrics.

Each of these stakeholders will be discussed in the following subsections.

2.2.1 Software vendors

Software products have vulnerabilities. The absolute number of vulnerabilities within any given software product is currently not measurable with any degree of confidence [7, 8]. What can be determined, and what software vendors must confront, is the number of vulnerabilities being reported and how long it takes to produce a patch. The length of time it takes to produce a patch is directly under the control of the vendor and can be directly influenced by the quality and quantity of resources devoted to the task. It is a business decision, and each vendor (perhaps each vendor's product line) has their own unique costs and benefits to consider.

The number of vulnerabilities being reported for the product is, at best only indirectly influenced by the vendor. The vendor can adopt some form of more secure software

development process such as Microsoft's Secure Development Life Cycle [9], which in principle would reduce the number of vulnerabilities which would have otherwise occurred. But the vendor can control neither the level of attention of nor the tools available to vulnerability researchers. As the quantity and quality of researchers looking at the deployed product increases, we would expect the number of vulnerabilities reported to also increase. As the tools available to researchers for aiding the identification of vulnerabilities improve or represent new types of attack, the number of vulnerabilities being reported would also be likely to increase.

From a business cost and end-user use perspective, vendors would prefer that vulnerabilities never be announced or even found [10]. However, they have little opportunity to control the release of vulnerability information unless they develop contracts with those researchers identifying and demonstrating vulnerabilities. While this has occurred, there are difficulties such as the fact that buying the information does not imply control; for example, other researchers may find the same vulnerability. Consequently, vendors must balance resources expended to develop and deploy patches for vulnerabilities against the potential losses of revenue due to reduced end-user choice of their product.

2.2.2 Vulnerability researchers

Vulnerability research firms actively search for or buy vulnerabilities. In this thesis, only the researchers who intend to report vulnerabilities to the vendor are addressed. The purpose and motivation for searching for vulnerabilities may be to gain notoriety in the hopes of increasing business volume, develop relationships which lead to increased recognition and security related business opportunities, or perhaps, altruistically, to improve the security of software products. In many cases recognition of the security firm, whether organization or individual, seems to be important.

For example, the security firm Secunia earns money by publishing advisories to its customers based on the vulnerabilities discovered by its research. The parent company of the Zero Day Initiative (ZDI), TippingPoint, produces advisories and intrusion detection

system (IDS) signatures for its paying subscribers based on their internal research and the vulnerabilities purchased from other researchers.

2.2.3 End-users

The end-users of software products which have vulnerabilities that have been discovered but remain unpatched expose themselves or their firms to risk. Ideally, end-users could know how many vulnerabilities exist in the software products they are using, determine the probability they will be exploited, and effectively determine the potential losses. But as discussed in Section 4.1, this information is neither dependably available nor verifiable. So new techniques are needed to help end-users assess their risk.

Vulnerabilities which have been publicly announced help end-users make rational decisions about whether to apply a patch if available, institute a workaround such as disabling the service or reconfiguring the process, or accept the risk. Vulnerabilities which have not been privately reported to the user, publicly announced, or mitigated by a third party such as Tippingpoint supplying IDS signatures for vulnerabilities they have purchased, leave the end-user relatively blind to the particular risk from these vulnerabilities. Vulnerabilities which have been discovered and reported to the vendor but not yet fixed constitute a risk which is mostly undetermined at this time but may present opportunity for improved estimation. The end-user exposure to these software vulnerabilities are discussed in detail in Section 4.2.

The MAV and VFD measure the relative exposure of end-users to vulnerabilities in products from software vendors. As a result, the metrics defined here, along with required features, could form the basis for choosing to use one software product versus another.

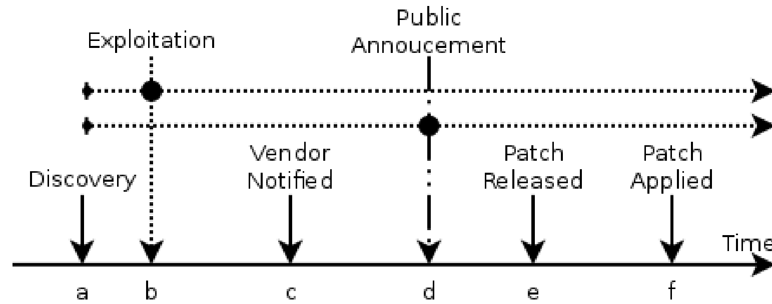


Figure 2.1. Vulnerability lifetime model

2.2.4 Vulnerability Repositories

Vulnerability repositories such as the National Vulnerability Database (NVD)[1] or the Open Source Vulnerability Database (OSVDB)[2] collect vulnerability information and disseminate it to the public. There role in the vulnerability ecosystem is simply as a repository of known vulnerabilities. Much of the research on the discovery of vulnerabilities uses statistics derived from these known vulnerabilities. The metrics defined in Chapter 4 rely on the availability of these databases; specifically, we require information to be collected on the time of report of a vulnerability until the time a corresponding patch is released.

2.3 Software Vulnerability Disclosure

The general model used for vulnerabilities in this work is depicted in Figure 2.1. Vulnerabilities are discovered internally by vendors or by external entities (security researchers described in Section 2.2.2). If discovered by a security researcher, he may choose to create an exploit, turn over the information to the affected vendor (responsible disclosure), or immediately publish it (instant disclosure). At some point, however, the affected vendor is notified and begins development of a patch. Once a patch is released, end users are responsible for applying the patch based on their local policies.

In responsible disclosure path, a researcher offers the vendor a grace period for

[1] http://nvd.nist.gov

[2] http://www.osvdb.org

addressing the vulnerability. The expiry of the grace period means the instant disclosure of the vulnerability, with or without a corresponding patch being available from the affected vendor.

2.4 Related Work

Arora et al. produced a number of works examining the vulnerability disclosure process as an economic decision [11, 12, 13]. In particular, [11] provides the basis for the model described in Section 2.3. Their primary work was the influence of disclosure grace period on vendor patch generation. However, the model proposed consisted of a number of terms that are simply not easily derived; their model is sound, but the parameters cannot be readily quantified. For instance, their model depends on knowing the time when a vulnerability is discovered which is complicated by the possibility of rediscovery [8] and by the difficulty of understanding the impact of a vulnerability [6]. The model presented in Chapter 4 relies only on the date a vulnerability is reported to the vendor and the date a patch is released to end-users; each of these dates are simple to collect and verify. In later work, they examined the behavior of vendors in response to instant disclosure and found that vendors patch instantly disclosed vulnerabilities much more quickly than those that go through the responsible disclosure path [13]. They also examined the effect of vulnerability announcement on attack frequency and found that patched and announced vulnerabilities attract more attacks than either secret (unpublished) or published (not patched) vulnerabilities [12].

Rescorla [8] questions the social utility in finding vulnerabilities. Specifically, whether finding vulnerabilities improves mproves the overall quality of software. He examined vulnerabilities found the NVD for four operating systems (Microsoft Windows NT 4.0, Sun Solaris 2.5.1, FreeBSD 4.0, and RedHat Linux 7.0) and applied several software reliability models and he concluded that there was no evidence that searching for vulnerabilities was, in fact, improving software reliability. His conclusion is based on the failure to reject the hypothesis that software reliability is constant despite changes in the

rate of vulnerability discovery.

Alhazmi and others examined vulnerability discovery models based on those used for software reliability and measured their predictive capability [14] for vulnerability density. This density is the ratio of vulnerabilities to code size presuming that larger codebases naturally have more vulnerabilities. Their model is tested against several vendors and products [15].

Ozment [16] examined vulnerability discovery models as well and describes a similar vulnerability discovery model to Rescorla. As with Arora, et al., not all of the parameters for the model can be easily measured for all vulnerabilities. This work also describes some of the potential pitfalls in using software reliability models in the vulnerability discovery process. Ozment and Schechter [7] examined the OpenBSD operating system to decide whether its security is increasing over time by comparing vulnerability reports to source code changes. They conclude that the rate of discovery of vulnerabilities in "foundational" code declined over the study time and that the foundational code is likely to greatly influence the overall rate of vulnerability reporting.

In addition to the previous works, several other authors have proposed predictive models and metrics for explaning discovery of vulnerabilities. In [17], Kandek describes four metrics describing the effects of software vulnerabilities: half-life (interval of reducing the number of vulnerable computers after a patch is released), prevalence (turnover rate in "Top 20" vulnerabilities per year), persistence (total life span of vulnerabilities), and exploitation (time between exploit announcement and first attack). These metrics concentrate on the results of exploits stemming from software vulnerabilities and less on the vulnerabilities themselves.

In [18], Acer and Jackson propose a novel metric for web browsers that measures the percentage of users visiting the authors' website that have at least one announced but unpatched high-severity vulnerability. It is unclear how this technique may be further generalized from web browsers, but it encourages disclosure and takes into account patch

deployment time. Clark et al. [19], examines the "honeymoon" period for new releases of software which is the time from release until the fourth vulnerability is discovered. They argue from their observations that "properties extrinsic to the software play a much greater role in the rate of vulnerability discovery than do intrinsic properties such as software quality."

Shryen [20] defines called the mean time between vulnerability discoveries (MTBVD) which is the ratio of the number of days since the release of a product and the total number of vulnerabilities found in that product. This metric predates the "honeymoon" work of Clark et al. [19], but is closely related, and it is also similar to the metrics defined in Chapter 4. Shryen's primary purpose is to examine the question of whether open source software is more secure than closed source software. He finds no significant difference, where Clark et al. finds that attackers both take longer to find bugs in open source software and familarity with the open source systems does not accelerate as quickly as with closed source counter parts.

Zhang et al. [21] defines the Time To Next Vulnerability (TTNV) metric and uses several different data mining approaches to create a predictive model for this metric. However, after a great deal of parameter tuning and feature construction techniques were applied, they concluded that because of limitations in the NVD data, no predictive model can be practically built. These authors are not alone in their laments over the limitations in the NVD data, for example [5, 8, 19, 21]. In other work, Zhang et al. [22] experiments with aggregating several metrics to quantify security risk of particular configurations.

Lamkanfi et. al. [23, 24] assessed automatic methods for classification of the severity reported bugs. Earlier, Cubranic and Murphy proposed textual analysis for bug triage [25]. Both sets of authors applied linguistic techniques to determine the severity of bug reports based on their textual content. The bug scoring system in Chapter 5 was inspired by these works.

Some authors have applied automated techniques such as static analysis and au-

tomated penetration analysis to the problem of discovering vulnerabilities. Austin and Williams [26] applied several such techniques to a healthcare software package and found that no one technique provided sufficient coverage. Khoo et al. [27] used static analysis software employed by two first year students to increase the number of known vulnerabilities in a software package by 10%.

Several authors have examined static analysis tools for various environments and problem sets, for example [28, 29, 30, 31, 32]. In the last of these, Zitser [32] used various static analysis tools on known vulnerabilities and found the performance of each of the then available open source tools to be far from ideal.

CHAPTER 3

Justification of Vulnerability Deadlines

This chapter examines whether the length of the grace period allotted to vendors by vulnerability research firms causes firms to release patches sooner. This chapter is a revised and extended of the work originally published in [33].

3.1 Introduction

For the purposes of this chapter, grace period and vulnerability lifespan are defined as follows:

- *grace period* s the amount of time the discoverer of a vulnerability allots to the vendor for providing a fix, after which the researcher may independently announce the vulnerability;

- *vulnerability lifespan* is the time from when a vulnerability is reported until the vendor provides a patch.

From 2002 through 2011, CERT/CC stated that they allow vendors a 45 day grace period [34]. In 2005 Phil Zimmerman, of PGP renown, was quoted as stating that the vendor should be allowed 30 days to fix a vulnerability [35]. In late 2010 three security organizations that perform vulnerability research, among other business functions, very publicly announced new grace periods they would give vendors. Rapid7 insisted on 15 days followed by a report to CERT/CC. Thus, Rapid7 effectively allows for a 60 day grace period [36]. Google announced they would allow a 60 day grace period [37]. The one notable outlier during late 2010 was the Zero Day Initiative (ZDI) which announced that they would allow vendors a 6 month, approximately 182 day, grace period [38].

The explanations and justifications from the vulnerability researchers emphasized the intent to protect end-users. Aaron Portnoy, a representative of ZDI, was quoted as saying “For every day a vulnerability goes unpatched, end users are susceptible, vendors

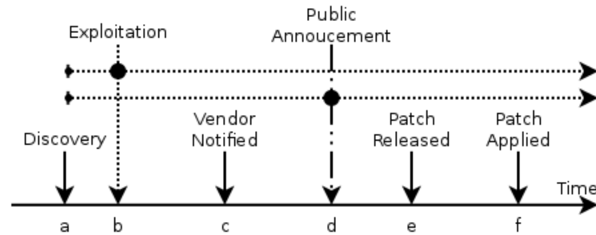


Figure 3.1. Vulnerability lifecycle events.

are being a little bit irresponsible by not patching them" [39]. Google Security team members posted on the Google Online Security Blog an article titled "Rebooting Responsible Disclosure: a focus on protecting end users" [37]. And in August 2010, HD. Moore, CSO of Rapid7, stated "The core issue is that the product has a security flaw; debating about the correct disclosure process shouldn't take away from the fact that the vendor is indeed responsible for anyone exploiting a problem in their product... The argument for disclosure is simple; the more the end user knows about the problem, the better they can defend against it" [40]. Unfortunately, none of these vulnerability researchers provided verifiable quantitative evidence supporting their chosen grace periods.

Thus, the announced grace periods once again raise questions about the appropriate disclosure timelines. The most important events in the vulnerability lifecycle are shown in Figure 3.1. Ideally, each event could be independently observed and validated. Unfortunately, most events are not credibly and verifiably known.

The initial discovery of a vulnerability cannot be firmly known by anyone, even a discoverer, since there is always the possibility it has been previously discovered. The date when exploitation begins is after initial discovery but otherwise unknown since exploitation detection mechanisms for previously unseen exploits have questionable detection rates. The rediscovery of a vulnerability is also problematic since the rediscoverer may not take action that makes that fact directly observable. The total cost of a vulnerability, which would include the total losses from its exploitation, the cost to mitigate by end-users, and the cost for the vendor to create a patch, is not dependably and verifiably

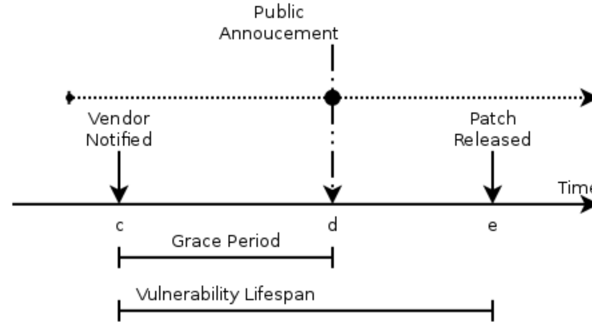


Figure 3.2. Measurable vulnerability lifecycle events.

known for similar reasons.

Only the events shown in Figure 3.2 remain: the date of vendor notification and the day of public announcement. The date reported to the vendor can in principle be known and verified, through oversight of the security researcher who reports the vulnerability. Of course, the disclosure date, independent of patch availability, is publicly known. And patch release dates for a vulnerability are usually publicly disclosed but may be determined through patch reverse engineering if necessary (e.g. [41]). These three events represent the dependably known and measurable aspects of the vulnerability disclosure process debate, and thus they should become the solid foundation on which the grace period and other vulnerability disclosure questions should be discussed.

The analyses is focused on comparing and using the grace periods, and the vulnerability lifespans. The analyses in this chapter do not focus on individual products nor on the rate of vulnerability reporting, both of which are considered in Chapter 4.

3.1.1 Contributions

For assessing the value of grace periods specified by vulnerability researchers, some indirect evidence that the shorter grace periods of 45 and 60 days do not appear as realistic as the 182 day grace period if applied to most vulnerabilities is provided.

Strong evidence that vendors still do respond, as previous work suggests [10], to the threat of disclosure is then provided. An analysis of the imposed 182 day grace period demonstrates that vendors do modify their patch process so that they are more likely to

have a patch available within 182 days. Unfortunately, the 182 grace periods also results in a significant increase in the number of vulnerability announcements made without a patch being available.

3.1.2 Organization

The rest of this chapter is organized as follows. In Section 3.2 the two primary disclosure processes are reviewed. In Section 3.3, the announced grace periods are compared to current vulnerability lifespans. In Section 3.4 the impact of the ZDI, 182 day, grace period on the speed of patch creation is assessed. Then, Section 3.5 provides the conclusion.

3.2 Overview of Vulnerability Disclosure

Software products have vulnerabilities. The absolute number of vulnerabilities within any given software product is currently unmeasurable with any degree of confidence [7], [8]. What can be determined, and what software vendors must confront, is the number of vulnerabilities being reported and how long it takes to produce a patch. The length of time it takes to produce a patch is directly under the control of the vendor and can be directly influenced by the quality and quantity of resources devoted to the task. It is a business decision, and each vendor (perhaps each vendors product line) has their own unique costs and benefits to consider.

The current vulnerability disclosure process has two primary forms. The first form is usually referred to as full disclosure and in effect means that upon discovery the vulnerability researcher may publicly announce full details of the vulnerability. The vendor is given no forewarning. The second form, responsible disclosure, generally means that the vulnerability researcher reports the vulnerability to the vendor and gives the vendor time to create a patch. A coordinated public disclosure of the vulnerability is then often made when a patch has been created by the vendor and is ready to be released.

Responsible disclosure has been the topic of heated debate. Some argue that vendors

are much to slow at patch development unless they are threatened with the potential of public announcement of the vulnerability, independent of whether a patch is available (supporting evidence may be found in [10]). Consequently, some vulnerability researchers and firms allot a specific amount of time, the grace period, for vendors to create and release a patch. At the end of the grace period these researchers feel free to partially or fully disclose the vulnerability with the idea that end-users may find ways to mitigate the problem even without a patch.

This raises the question of whether these vulnerability researcher specified grace periods are sensible from an end-user vulnerability exposure perspective. A start to answering this question is provided by analyzing publicly accessible lifespan data of vulnerabilities used in the Pwn2Own competition[1], and then analyzing the lifespans of a much more general set of vulnerabilities. All lifespan data was collected from ZDI.

3.3 Grace Periods Compared to Vulnerability Lifespans

3.3.1 Pwn2Own Vulnerability Lifespans

Pwn2Own is a well known computer hacking competition held every year since 2007 as a part of the CanSecWest security conference. The contest has high visibility and the vulnerabilities exploited by the contestants gain a fair amount of attention from the vendors of the exploited products. For example, Daniel Veditz, the Security Group Moderator for Mozilla made this comment about a vulnerability exploited at Pwn2Own in 2009: "... Since this is a high profile bug (Firefox cracked during a public hacking contest) we need to focus on it. If we had a fix I'd like to shoehorn it into 1.9.0.8 even though we're past codefreeze (April release) but May's 1.9.0.9 is more realistic. Needs to make 3.5b4." [42] Increased vendor attention seems like it should decrease the time for the vendor to produce a patch.

Table 3.1 shows all 15 of the previously undisclosed vulnerabilities that could be

[1] `http://cansecwest.com/post/2012-02-23-20:00:00_New_PWN2OWN_Rules` [Accessed: January 21, 2013]

Lifespan	(days)	Product	Year	CVE
< 45	8	Apple QuickTime	2007	CVE-2007-2175
	10	Firefox	2010	CVE-2010-1121
	11	Firefox	2009	CVE-2009-1044
	19	Safari	2010	CVE-2010-1120
	20	Safari (WebKit)	2008	CVE-2008-1026
< 60	55	Safari (WebKit)	2009	CVE-2009-0945
	55	Mac OS X	2009	CVE-2009-0154
< 182	61	Adobe Flash Player	2008	CVE-2007-601
	72	Safari (WebKit)	2010	CVE-2010-1119
	83	Internet Explorer 8	2009	CVE-2009-1532
> 182	310+	Internet Explorer 8	2010	CVE-2010-1118
	310+	Internet Explorer 8	2010	CVE-2010-1117
	676+	Safari	2009	CVE-2009-1060
	676+	Safari	2009	CVE-2009-1042
	676+	Internet Explorer 8	2009	CVE-2009-1043

Table 3.1. Pwn2Own Vulnerability Lifespans

identified as being exploited at Pwn2Own. Of these 15, 10 have patches available and their lifespans range from 8 days to 83 days. 50% of the lifespans were 45 days or less, 70% were 60 days or less and all of them had patches available in 182 days or less. These vulnerability lifespans seem to be in life with the shorter grace periods alloted by vulnerability researchers.

However, looking at the complete data in Table 3.1, the picture becomes more muddled. There are 5 Pwn2Own vulnerabilities (30%) which, have yet to be fixed. Including these vulnerabilities in the analysis, only 33% of Pwn2Own vulnerabilities are fixed in 45 days, 47% are fixed in 60 days and 67% are fixed in 180 days.

3.3.2 ZDI Vulnerability Lifespans

The lifespans for 473 vulnerabilities were collected from ZDI. The lifespans included all vulnerabilities which were initially sold to ZDI and then disclosed sometime between November 7, 2009 and April 30, 2011. Around 15% of the lifespans were less than 45 days, 17% less than 60 days, and fully 64% of lifespans were less than 182 days. These lifespans can be seen in Figure 3.3.

These lifespans do raise the question of whether the 45 and 60 day grace periods are practical at this time. With a median of 140 days, a mean of 197 days, and a maximum

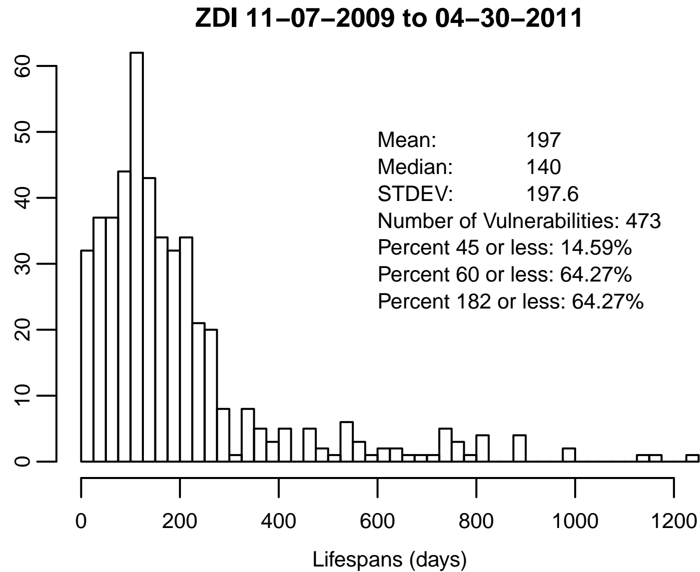


Figure 3.3. ZDI vulnerability lifespans: Nov, 2009-April, 2011.

of over 3 years, it may be unreasonable to expect vendors to be able to meet the grace periods. Even if 10% of the vulnerabilities are granted an exception by the vulnerability researchers, the range of lifespans is 2 days up to 421 days. The 182 day grace period may be more hopeful since it would require a maximum vendor patch creation speedup of 57%. Not easy, but not as difficult as the shorter grace periods.

Of course the adoption of grace periods assumes that vendors actually will speed their patch creation process when confronted with the possibility of an independent disclosure before they have a patch available. Work by Telang and others in 2005 provides some indication that vendors do indeed speed up in these circumstances, in order to protect business value [10]. But it was decided to see if there was evidence in the collected ZDI data which indicated that is still the case.

3.4 Did Vendors Speed Up Their Patch Creation

On August 4, 2010, ZDI announced their intention of imposing a grace period of 6 months on vendors. ZDI indicated that the new policy would begin immediately.

3.4.1 Impact on Initial Pool of Vulnerabilities

The initial pool of vulnerabilities (InitPool) is defined to be those ZDI acquired vulnerabilities which, as of August 4, 2011, had been reported to the vendor but were yet to be publicly disclosed. ZDI stated that each of the vulnerabilities in the InitPool would be treated as newly acquired. Thus February 4, 2011 would be the grace period deadline for all InitPool vulnerabilities.

When publicly announced, there are three different states for a vulnerability. At announcement, the vulnerability might have a patch available, it might have a vendor specified fix other than a patch, or it might not have any mitigation at all. Since some of the vendor specified fixes did not, in the opinion of the authors, credibly address the vulnerability, it was decided to conservatively group together all vulnerabilities announced without a patch and treat them as unmitigated.

The InitPool started with vulnerabilities which had ages ranging from 14 to 1170 days. The mean age was 183 days and the median age 64 days. There were a total of 172 InitPool vulnerabilities, of which 29 (16.9%) were eventually announced without patches being available. All InitPool vulnerabilities were publicly announced no later than 2 weeks after the grace period deadline. The week in which InitPool vulnerabilities were publicly announced are shown in Figure 3.4. The number of patched and unpatched vulnerability announcements are given for each week. It is interesting to note that 16 vulnerability announcements for which there were no patches occurred just days before and after the grace period deadline of February 4, 2011. ZDI enforced their 6 month grace period.

The 16.9% of InitPool vulnerabilities which were announced without a patch, can be thought of as having exceeded the maximum 6 month lifespan imposed by the grace period. In the 87 days immediately preceeding the ZDI announcement of a grace period, over 32.8% of announced vulnerabilities had lifespans exceeding 6 months. So it does appear that some patch speedup was attained after ZDI imposed their grace period on

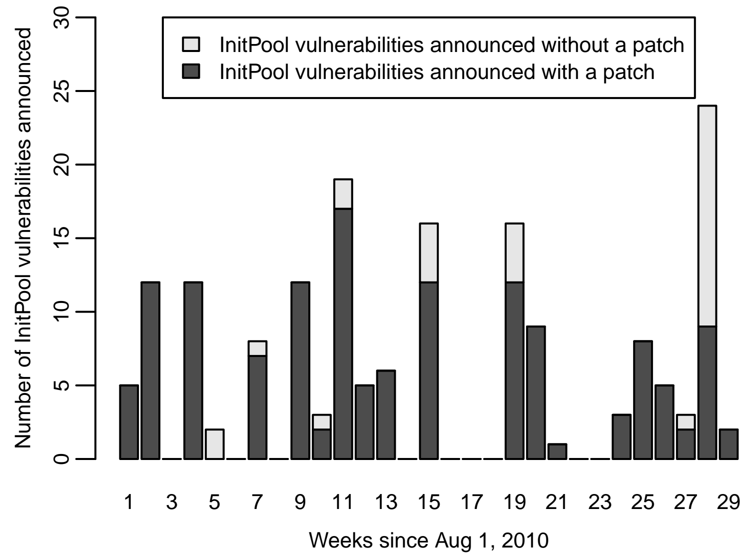


Figure 3.4. Number of InitPool vulnerability public announcements (each week after ZDI grace period announcement.

vendors. However, the InitPool vulnerabilities might have been biased towards more difficult to patch vulnerabilities or vendors more susceptible to speeding up their patch creation when confronted with a threat of disclosure. Since InitPool vulnerabilities might not be representative of all vulnerabilities, two other sets of vulnerabilities were compared.

3.4.2 Lifespan Comparison Before and After ZDI Announcement

In Figure 3.5 the ZDI lifespan data is partitioned into four equal length periods. Period 1 is the 269 days immediately before the ZDI grace period announcement on August 4, while Period 2 is the 269 days immediately after the announcement, and Period 3 is the 269 days following Period 2, and Period 0 is the 269 days before Period 1. Each period has an 87 day time frame at the beginning. Vulnerability Set 0 represents the group of vulnerabilities acquired by ZDI and reported to the vendor in the first 87 days of Period 0. Vulnerability Set 1 is the set of vulnerabilities acquired by ZDI and reported to the vendor in the first 87 days of Period 1, and so on. The expectation was that with Set 2 and Set 3 vulnerabilities the vendors would be more likely to have produced patches before the 6 month grace period elapsed than with the Set 0 or Set 1

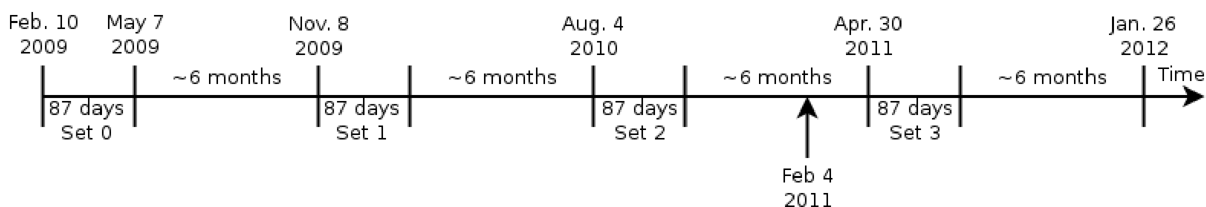


Figure 3.5. Three sets of ZDI acquired vulnerabilities for lifespan comparison.

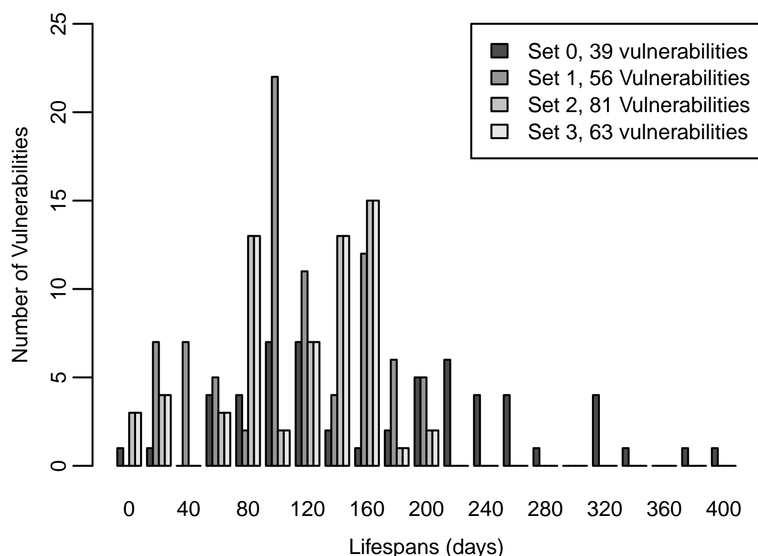


Figure 3.6. Distribution of vulnerability lifespans for Sets 0, 1, 2, and 3.

vulnerabilities for which there had not been a grace period deadline. The vulnerability lifespans of Set 0, Set 1, Set 2, and Set 3 were compared to determine if this effect did in fact occur.

The lifespan distributions for both sets of vulnerabilities are shown in Figure 3.6. The lifespan statistics for Set 1 and Set 2 can be seen in Table 3.2. In Set 2, the vulnerability lifespans should not exceed 6 months, but 9 vulnerabilities were announced with patches less than one month after their 6 month grace periods, and 12 vulnerabilities were announced without patches. Together these were counted as 21 vulnerabilities which exceeded the grace period of 6 months. Of note is that Set 1 had 51.8% of lifespans exceed 6 months while Set 2, those vulnerabilities acquired immediately after imposition of the grace period, had only 25.9% of vulnerability lifespans exceed 6 months.

Table 3.2. Did ZDI announcement have an impact?

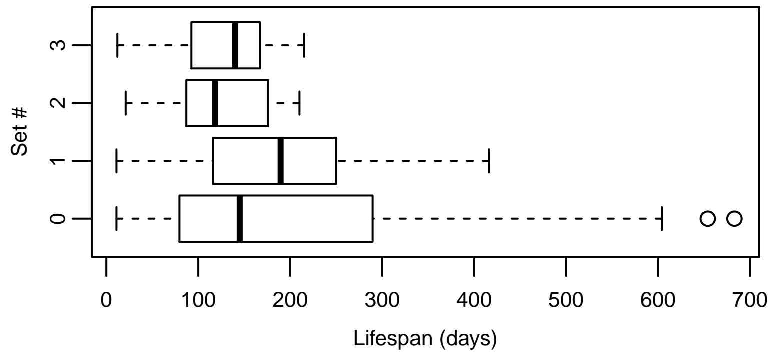
Set (i)	0	1	2	3
Total Vulnerabilities (A_i)	39	56	81	63
Range of lifespans	11–683	11–416	21–210	12–215
Mean lifespan	215	189	122	125
Median lifespan	145	190	118	140
Stdev of lifespan	182.16	93.37	52.60	49.35
Vulnerabilities announced without a patch	XXX	1	12	0
Lifespans > 6 months (B_i)	16	29	21	3
Lifespans < 6 months (C_i)	23	27	60	60
Proportion > 182 days ($P_i = {}^{B_i}/_{A_i}$)	0.410	0.518	0.259	0.048

ANOVA Table for Sets 0, 1, 2, and 3.

	df	Sum Sq	Mean Sq	F value	Pr($> F$)
set	3	352843	117614	13.083	6.247e-08
Residuals	235	2112608	8990		

Choosing a null hypothesis of no difference between the means of the sets, using One-Way ANOVA, the hypothesis that the means of the populations are the same is rejected ($p = 6.2 \times 10^{-8}$). However, in the presence of skewed data, the Kruskal-Wallis test may be more appropriate and the hypothesis that the sets came from the same distribution is rejected as well ($\chi^2 = 18.71$, $df = 3$, $p = 0.00031$). The ANOVA table for the previous test is summarized in Table 3.2, and the Kruskal-Wallis results are in Figure 3.7 along with a boxplot of the lifespan data. Using Tukey's procedure, Sets 0 and 1 are found to have an equivalent mean and Sets 2 and 3 are found to have an equivalent mean. However, Sets 0 and 1 are found to be greater than Sets 2 or 3. Using standard notation for this: $\underline{\overline{x_2}\ \overline{x_3}}\ \underline{\overline{x_1}\ \overline{x_0}}$, where $\overline{x_n}$ is the mean of set n.

It should be noted that the ZDI grace period deadline is not entirely firm. Exceptions may be made if ZDI, through discussion with the vendor, decides there are mitigating circumstances. Consequently, a fairer analysis of the impact of the ZDI grace period might focus entirely on the publicly announced vulnerabilities for which no patch was available. Rather than 21 vulnerabilities exceeding the grace period in Set 2, there would be just 12 vulnerabilities, 14.8%, announced without a patch. This is because 9 of the 21



Set	0	1	2	3
Total Vulnerabilities	39	56	81	63
Median	145	189.5	118	140
Kruskal-Wallis χ^2	18.71			
p-value	0.00031			

Figure 3.7. Summary of Wilcoxon Rank Sum test results

vulnerabilities exceeding the 6 month grace period were announced with a patch in the seventh month. Those 9 vulnerabilities reflect some flexibility in the grace period when ZDI believes the vendor is responding to fix the problems.

To make a corresponding change to the 29 Set 1 vulnerabilities which exceeded the 6 month grace period, only 2 were removed since they were also announced in the seventh month. Thus there were still 27 vulnerabilities in Set 1, 48.2%, which had lifespans longer than the extended grace period of 7 months. Following the same analysis as above this leads to a new P-value of 0.00001. Once again the null hypothesis of equivalent lifespans for both Set 1 and Set 2 can be rejected for any reasonable level of significance.

Because Set 1 lifespans (before the ZDI announcement) are significantly greater than Sets 2 and 3 (both after the announcement), there is strong evidence that in the general pool of ZDI acquired vulnerabilities, the 6 month grace period did result in vendors speeding up their patch process. There is also evidence that the grace period results in more vulnerability announcements without a patch being available.

3.5 Conclusion

Vulnerability research organizations such as Rapid7, Google Security team, and ZDI have imposed grace periods for public disclosure of vulnerabilities with or without an effective mitigation from the affected software vendor. At this time not data was found which either firmly support or refute the usefulness of the shorter grace periods of 45 and 60 days. There is evidence that the ZDI grace period of 182 days yields some benefit in speeding up the patch creation process. From a risk perspective it is important to note that even after the new grace period there were still 25.9% of ZDI reported vulnerabilities which did not have patches available in the specified time frame.

The ZDI grace period of 182 days appears to have had a lasting effect. Vulnerabilities announced well after the announcement in the change in grace period, the lifespan of vulnerabilities has stayed constant with those reported just after the announcement. Both of these sets show significantly lower lifespan than vulnerabilities report before the change.

CHAPTER 4

Analysis of Two End-User Vulnerability Metrics

This chapter describes two metrics for describing the exposure risk of software vulnerabilities to end-users. It is a revised and extended version of the papers published in [43, 44].

4.1 Introduction

Every week new software vulnerabilities are discovered in many applications and patches are issued fixing previously discovered vulnerabilities. Various measurements of this effect have been proposed, but comparisons between similar products from different vendors or different products with the same vendor have been difficult. This chapter proposes two new end-user focused metrics that allow for cross product or cross vendor comparison. The metrics are based on measurements of the number and rate of vulnerabilities reports, and the patch development rate for individual software products. These measurements are related to events which are part of the vulnerability life cycle.

To quantitatively characterize the time between events in the vulnerability life cycle model, depicted in Figure 4.1 and fully described in [11], ideally one would measure the time from discovery of a flaw until the time all end-user machines have been patched to address the issue. In practice, this has been demonstrated to be difficult since the times and dates for most events along the life cycle are not credibly and verifiably known.

For instance, it is difficult to accurately record the initial discovery of a vulnerability

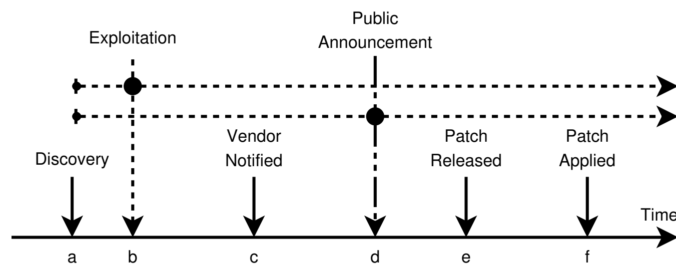


Figure 4.1. Vulnerability lifetime model

(a), even for a discoverer because it is possible the vulnerability has been independently discovered by another party [8]. On the other end of the life cycle, it has been shown that applying security patches (f) involves a half-life behavior and finally tapers off at approximately 5–10% of machines that will remain unpatched [17].

In practice, we can measure the time from when a vulnerability is reported to a vendor (c) until the time when a patch is issued by that vendor (e). For instance, ZDI and iDefense both buy vulnerabilities from the security research community and then report them to the appropriate vendor. In doing so, they record the time from report to patch release. Essentially, this leaves only two stages in the vulnerability life cycle that can be accurately known:

- birth: vulnerability reported to the vendor (c), and
- death: patch issued by the vendor (e).

We define vulnerability lifespan in the same way as in Chapter 3: the time from when a vulnerability is reported until the vendor provides a patch. This is the time between the birth and death of the vulnerability. A vulnerability is considered "active" from the time it is reported to or discovered by the vendor until a patch is supplied by the vendor. Metrics based on the number of "active" vulnerabilities in a vendor's queue can be used to aid quantitative estimation of end-user exposure.

Simply examining the raw quantity of vulnerabilities reported for a product in databases like the National Vulnerability Database (NVD) or the Open Source Vulnerability Database (OSVDB) neglects the effect of the vendor response time to addressing vulnerabilities. Likewise, examining the lifespans of vulnerabilities from sources such as the Zero Day Initiative (ZDI) or iDefense neglects the number of vulnerabilities. New metrics which combine both quantity and lifespan of vulnerabilities for individual products would be useful.

In this chaper, two new metrics are presented that capture the effect of the number

and rate of new vulnerabilities being found and their lifespans. The first metric, median active vulnerabilities (MAV), is the median number of software vulnerabilities which are known to the vendor of a particular piece of software but for which patch has been publicly released by the vendor. The second metric, vulnerability free days (VFD) captures the probability that a given day has exactly zero active vulnerabilities.

4.1.1 Summary of contributions

Two new metrics that focus on end-user software vulnerability exposure from individual products are defined. The two metrics are then used in a case study of four browsers (across vendors) and two other products (within vendor) to discuss and demonstrate that:

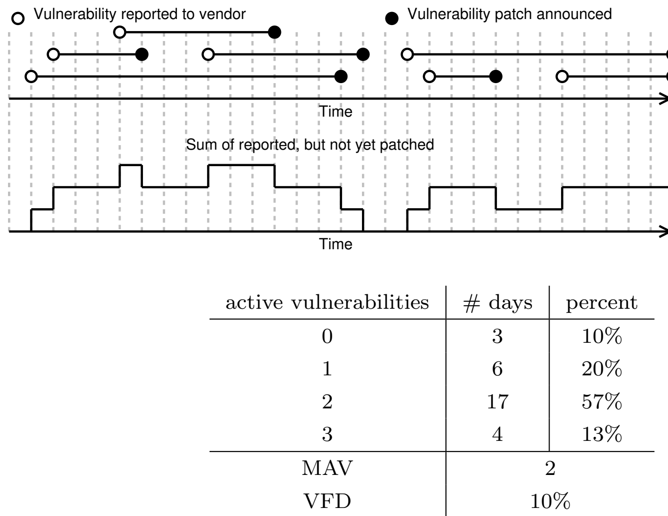
- end-user vulnerability exposure should be considered as a combination of lifespans and vulnerability announcement rates (not lifespans alone), the proposed metrics capture both aspects;
- the two metrics may be easily estimated with reasonable accuracy, and thus are usable by end-user security practitioners and decision makers;
- individual products with the same functionality, e.g. browsers, may yield distinctly different end-user vulnerability exposure levels.

4.1.2 Organization of Article

The rest of this article is organized as follows. In Section 4.2, the two new end-user focused metrics are expanded upon and then in Section 4.3, the metrics are applied to four web browsers and the results of applying the metrics are examined. Section 4.4 contains discussion and describes related work. Finally, Section **??** provides the conclusion.

4.2 Two End-User Exposure Metrics

It is useful to provide all stakeholders (vulnerability researchers, vendors, and end-users) with security metrics which support accountability and decision making. To this



active vulnerabilities	# days	percent
0	3	10%
1	6	20%
2	17	57%
3	4	13%
MAV	2	
VFD	10%	

Figure 4.2. Example of MAV and VFD calculation

end, two vulnerability exposure metrics are defined as proxies for a product's contribution to an end-users level of vulnerability exposure. The first metric, Vulnerability Free Days (VFD), is the percent of days in which the vendor's queue of reported vulnerabilities for the product is empty; viewed over the long term this is the probability that there are no active vulnerabilities on a given day. The second metric, Median Active Vulnerabilities per day (MAV), is the median number of vulnerabilities per day in a vendor's product queue. The median is used instead of mean because the median is less sensitive to extreme outliers and skewed distributions.

Figure 4.2 shows a hypothetical example. At the top, vulnerabilities are reported and patched as time moves from left to right. The bottom shows the running sum of active vulnerabilities. If each horizontal division is taken to be a day, there are 3 days with no vulnerabilities, 6 days with exactly 1 active vulnerability, 17 days with 2, and 4 days with 3. The MAV is the median number of active vulnerabilities: 2 (in other words, on a given day there is a 50% chance the vendor is working on 2 or fewer vulnerabilities). The VFD is $^3/_{30} = 10\%$.

These metrics are primarily intended for consumption by end-users, particularly those in charge of making policy decisions as to which software vendors and products

should be purchased, or which should form part of an “allowed use” policy. Comparative evaluation of software products, or vendors as a whole, can be expressed by calculating and examining their MAV and VFD values. A product with a small median number of active vulnerabilities should have some preference over one with a higher median. The inverse is true with the vulnerability free days metric where a large number is preferred to a small number.

The information needed to calculate these two metrics for a product are the lifespan of each reported vulnerability, and the number and rate of vulnerability disclosures. While not currently easy to obtain, in principle this information would be easy to produce and verify by the vendors. The data could also be verified by the independent researchers who reported vulnerabilities, and the vendors could be induced to make the information free and easily accessible if end-user pressure is brought to bear.

4.3 Metrics Case Study

The proposed VFD and MAV vulnerability exposure metrics were estimated for several different software products: web browsers (Apple Safari, Google Chrome, Mozilla Firefox and Microsoft Internet Explorer) as well as Microsoft Office and Apple QuickTime. To develop the metrics for each product, data was collected in order to characterize their respective vulnerability lifespans, and number and rate of vulnerability disclosures. After some success in characterizing this information for each product, a simulation was written and used to estimate the metrics. The possibility for quick and easy short cuts for approximating the metrics are discussed at the end of the case study (Section 4.3.5).

In particular, several data sources were used to estimate the:

- arrival rate of vulnerability announcements,
- number of vulnerabilities announced, and
- lifespan of vulnerabilities.

The arrival rate of vulnerability announcements is the time between two different announcements of vulnerabilities for a given product. The number of vulnerabilities announced represents the integral number of vulnerabilities disclosed as part of a specific announcement. It is common that more than one vulnerability for a given product is announced on a given announcement day (e.g. Microsoft "patch Tuesday"). The lifespan of a vulnerability is the same as defined previously. It begins when the vulnerability is reported or discovered by the vendor, and ends when the vendor supplies a patch.

4.3.1 Data Sources

Data was gathered from the National Vulnerability Database (NVD) [45], iDefense Vulnerability Contributor Program (VCP) [46], and the Zero Day Initiative (ZDI) [47]. The NVD data was used to characterize the arrival rate of vulnerability announcements and the number of vulnerabilities announced per instance. The ZDI and iDefense data were used to characterize vulnerability lifespans. In all cases, descriptive statistics are provided to give an idea of the behavior of the data harvested from each source.

The NVD consists of approximately 46,000 unique vulnerabilities enumerated by an identifier called a Common Vulnerability Enumeration Identifier (CVE). The database is freely available and further breaks down vulnerabilities by vendor, product, version, etc. (Common Platform Enumeration, CPE). For this research, the XML data feed provided by NVD was downloaded and imported into an SQL database so that arbitrary queries could be executed. The data was used for computing the arrival rates of vulnerabilities, and determining the number of vulnerabilities disclosed at each announcement.

The National Vulnerability Database has been widely criticized for the inaccuracies it contains. For example, [48, 19, 21] all describe various inconsistencies in the NVD and other vulnerability databases. In this chapter, the primary concern is describing the concept and potential usage of the metrics presented, so there is less concern with the absolute consistency of the existing sources.

To minimize the effects of the erroneous data in the NVD, the time span of anal-

Product	N	Time Start	Time End
MS Internet Explorer	209	Jan 1, 2001	Jun 6, 2010
Mozilla Firefox	113	Jan 1, 2004	Jun 6, 2010
Google Chrome	30	Dec 12, 2008	Jun 6, 2010
Apple Safari	92	Jun 22, 2003	Jun 6, 2010
MS Internet Explorer 6	180	Jan 1, 2001	Jun 6, 2010
MS Internet Explorer 7	85	Jan 1, 2004	Jun 6, 2010
MS Internet Explorer 8	20	Jan 1, 2009	Jun 6, 2010
MS Office	246	Jan 1, 2005	Jun 6, 2012
Apple QuickTime	138	Jan 1, 2005	Jun 6, 2012

Table 4.1. Number of points and time span for each product in NVD.

ysis is limited for each product and only two fields were used: the Common Platform Enumeration (CPE) and the "first published" date. The vendor and product fields of the CPE were used to discriminate between products. Other parts of the CPE were ignored, except when making the distinction between Internet Explorer versions. The "first published" field of the NVD is used to examine the arrival rate of announcements and the number of vulnerabilities announced per day.

Limiting the dates for which vulnerability data are collected provides the ability to ignore the start-up effects of the NVD. As pointed out in [21], the early years of the NVD were unstable. Table 4.1 shows the time span considered for each product and the number of data points available in the time span. Before 2004, the Firefox browser was a product of Netscape Communications called Netscape Communicator. It is difficult to determine whether vulnerabilities were inherited from the Communicator product or introduced during the transition to the Mozilla Foundation Firefox product, so the study of this product is limited to vulnerabilities discovered after the transition. Google Chrome was introduced in December 2008, and Apple Safari was released in January of 2003. In addition to Microsoft Internet Explorer as a whole, individual versions are broken out separately.

The data from iDefense and ZDI is considered to be more reliable since they can directly observe the time between when they notify a vendor and when a corresponding patch is produced. The former time being directly controlled by iDefense/ZDI and the

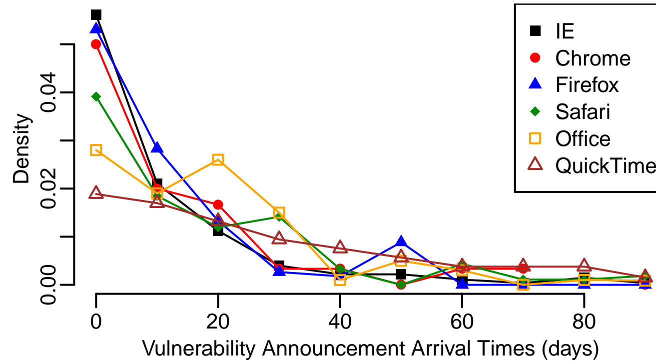


Figure 4.3. Histogram of vulnerability announcement rates.

latter time being publicly observable. For vendors like Microsoft which produce their own security advisories, it might be possible to gather the required data from the issued advisories. However, using the simulation method described in Section 4.3.2 is a more general solution and integrates more reliable observations of vulnerability lifespans than those provided using the NVD alone.

Vulnerability Announcements

Figure 4.3 shows the histogram of vulnerability announcements for all of the studied products; while the mean and median values differ substantially, the histograms have roughly the same shape. Table 4.2 summarizes the statistical properties of the announcement rate. If one were to choose a web browser simply by the arrival rate of new vulnerability announcements, one would choose Apple Safari because the expected time between new vulnerability announcements is slightly over 25 days (more than 3 weeks), and the other browsers are less than 3 weeks. Firefox does not fare well at all with new vulnerabilities announced about 12 days apart.

Number of Announcements per day

However, because arrival rate is actually an announcement of at least one vulnerability and possibly more, the distribution of the number of vulnerabilities on an announcement day is examined. The distributions for each studied product are shown in

Product	mean	median	σ	min / max
MS Internet Explorer	14.95	9.0	16.3	1 / 98
Mozilla Firefox	12.09	10.0	10.5	1 / 51
Google Chrome	17.17	10.5	18.4	1 / 80
Apple Safari	25.47	15.5	28.5	1 / 125
MS Internet Explorer 6	17.36	10.0	18.6	1 / 97
MS Internet Explorer 7	23.14	13.0	41.7	1 / 365
MS Internet Explorer 8	21.15	14.0	16.5	1 / 54
MS Office	24.11	22.0	20.0	1 / 113
Apple QuickTime	46.83	33.0	45.4	1 / 202

Table 4.2. Properties of vulnerability announcement rates (days).

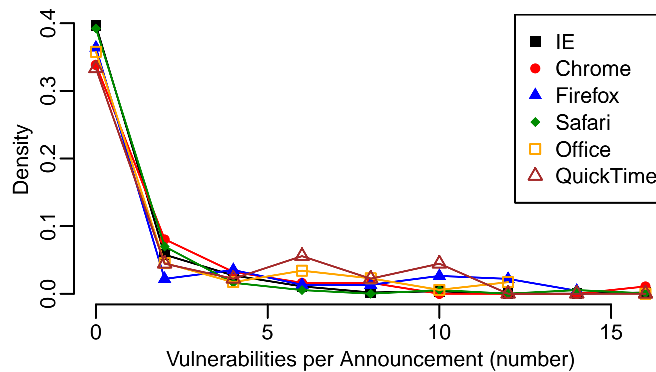


Figure 4.4. Histogram of vulnerabilities announced on announcement day.

Figure 4.4. Table 4.3 summarizes the number of vulnerabilities per announcement. Internet Explorer and Safari are close to 2 vulnerabilities per announcement on average where as Firefox averages more than 3 vulnerabilities per announcement. However, the median number of vulnerabilities announced on an announcement day for all products is 1, meaning that at least half of the announcements are for a single vulnerability.

Vulnerability Lifespans

The ZDI and iDefense databases consist of vulnerabilities for which the corresponding firm has paid a security researcher for a vulnerability. ZDI or iDefense then works with the affected vendor to responsibly disclose the vulnerability. Both companies provide free and online access to the data including the date the company reported the vulnerability to the vendor and the date at which the vulnerability was publicly disclosed. The collected data were used for computing the distribution of vulnerability lifespans.

Product	mean	median	σ	min / max
MS Internet Explorer	2.105	1.0	2.075	1 / 17
Mozilla Firefox	3.158	1.0	3.811	1 / 16
Google Chrome	2.871	1.0	3.667	1 / 19
Apple Safari	2.279	1.0	4.108	1 / 36
MS Internet Explorer 6	2.188	1.0	2.121	1 / 15
MS Internet Explorer 7	1.733	1.0	1.332	1 / 6
MS Internet Explorer 8	1.221	1.0	1.221	1 / 5
MS Office	2.795	1.0	3.231	1 / 14
Apple QuickTime	3.067	1.0	3.055	1 / 11

Table 4.3. Properties of vulnerability announcement rates (number of announcements).

Product	N	mean (days)	σ (days)	min / max (days)
MS Internet Explorer	33	182.1	106.9	47 / 489
Mozilla Firefox	20	91.6	50.7	11 / 184
Google Chrome	5	114.6	41.3	56 / 146
Apple Safari	10	106.8	55.5	20 / 210
MS Office	61	235.3	186.5	21 / 876
Apple QuickTime	53	113.4	79.4	3 / 372

Table 4.4. Distribution of ZDI/iDefense lifespans for each product.

Table 4.4 shows the descriptive statistics for the distribution of the ZDI and iDefense lifespan data. Firefox has the clear lead at 91.6 days to address vulnerabilities and Internet Explorer lags far behind with a mean of 182 days to address vulnerabilities. Figure 4.5 shows a diagram of the empirical cumulative distribution functions of the lifespans for each browser. For each observed sample lifespan, the graph rises $1/N$ at that point along the horizontal axis. A rapid vertical rise shows a clustering of observed lifespans and small slope shows few observed lifespans of that value. Figure 4.5 is a more detailed examination of the distribution information in Table 4.4. For instance, MS Internet Explorer is shown to have an overall slower distribution of lifespans; part of this is caused by a small number of high value lifespans ($>$ 450 days). The other three browsers have similarly positioned and shaped lifespan distributions.

4.3.2 Model for Simulation

To facilitate estimation of the MAV and VFD metrics, a model and corresponding simulator were constructed. A simulation was employed because the exact data are not

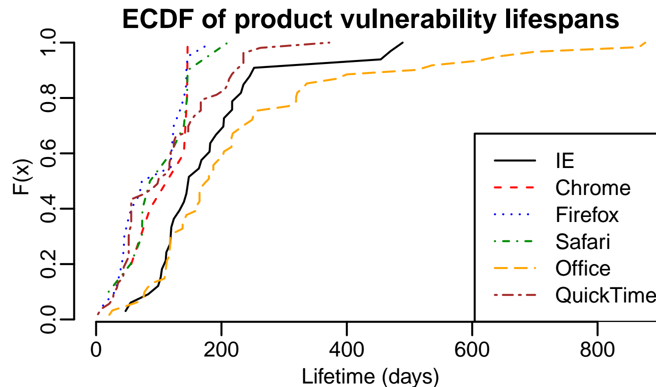


Figure 4.5. Empirical cumulative distribution functions of product vulnerability lifespans.

known and a closed form solution based on the empirical distributions is not yet available (though an approximation is found and discussed in Section 4.3.5). To generate a single simulation run, time is set to t_0 and a sample is taken from the announcement arrival rate distribution for the browser under study, Δt. Then, at time $t = t_0 + \Delta t$, a sample is taken from the distribution of the number of vulnerabilities announced on an announcement day. This determines how many vulnerabilities are terminated with the announcement, n. For each $i \in 1, \ldots, n$, a sample is taken from the lifetime distribution, l_i.

For the discrete event simulation, two events are generated:

- a vulnerability birth at time $t - l_i$ and
- a vulnerability death at time t.

Finally, t_0 is set to t and event generation continues until $t_0 > t_{\text{end}}$ where t_{end} is the simulated time.

To compute the MAV metric, the discrete number of vulnerabilities estimated to be in the vendors queue each day was put in rank order and the probability of each was computed. Finding the median is then a matter of finding number of vulnerabilities corresponding to the 50th percentile. The VFD metric is calculated by counting the number of days in the simulation with exactly zero vulnerabilities, then dividing by the simulation days to obtain the probability of no vulnerabilities. To minimize simulation

warm-up and wind-down, the simulation was run for 100 different random seeds and over a simulated time of 100 years.

This simulation model is a $G/G/\infty$ queuing model: generalized arrival process, generalized service time, and an infinite number of servers. The arrival process is complicated by the fact that multiple vulnerabilities can be announced at a single point in time. Even if the underlying data could be mathematically modeled, the authors believe that there is no closed form solution for the MAV or VFD metrics.

Various statistical models were tried for each of the different probability distribution functions required by the simulation. Since the model parameters were not equally well characterized by the statistical models, the simulations were run using the raw data collected for each parameter as a discrete distribution function. The results of the simulations were used to calculate the VFD and MAV for each browser.

4.3.3 Estimation of MAV and VFD Across Vendors

For estimating the MAV metric, the arrival rate of announcements, number of vulnerabilities disclosed per announcement, and the vulnerability lifespans are random variables distributed as described in Section 4.3.1. The distributions were derived from the collected data. The simulation provided the results shown in Figure 4.6. The horizontal axis is the number of vulnerabilities in a vendors queue and the vertical axis is the percentage of days which had that number of vulnerabilities. The MAV metric was then calculated as the median number of active vulnerabilities.

The MAV estimate for each of the four browsers was 9.55 for Safari, 19.1 for Chrome, 23.9 for Firefox, and 23.2 for Internet Explorer (this data is summarized in Table 4.5). So the estimated vulnerability exposure, MAV, due to deployment of a web browser is distinctly different depending on which web browser is in use. Safari is clearly superior to the other three browsers.

However, there is a question of whether it is reasonable to group the data from Internet Explorer versions 6, 7, and 8 together since each version might have distinctly

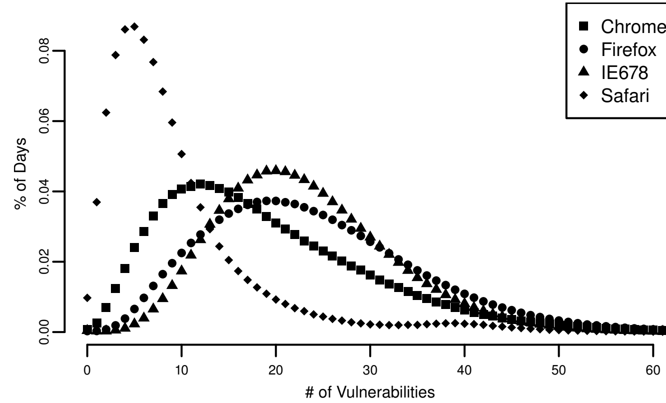


Figure 4.6. Percent of days with the given number of vulnerabilities in a vendors queue

Browser	MAV	σ
Apple Safari	9.55	8.58
Google Chrome	19.1	11.3
Mozilla Firefox	23.9	11.1
Internet Explorer (all)	23.2	8.94
Separate treatment of IE versions		
Internet Explorer 6	20.7	8.70
Internet Explorer 7	12.2	6.90
Internet Explorer 8	13.5	4.76

Table 4.5. Browser Median Active Vulnerabilities

different values for the model parameters and thus different MAV metric values. So Internet Explorer was further decomposed and the MAV was recalculated for each version. Grouping the three versions together results in a higher overall MAV because the sets of vulnerabilities are not independent; a vulnerability may affect one or more major versions of the browser. This in turn affects the sampling of report rate, announcement rate, and lifespan.

The simulation results for Internet Explorer versions 6, 7, and 8 are shown in Figure 4.7. The MAV estimate was 20.9 for Internet Explorer version 6, 12.2 for Internet Explorer version 7, and 13.5 for Internet Explorer version 8. Internet Explorer 6 is clearly the poorest performer according to the MAV estimates. This is in line with the general security community expectations. The cause for Internet Explorer 6 showing so poorly while versions 7 and 8 are have quite similar MAV values is unknown. The difference is

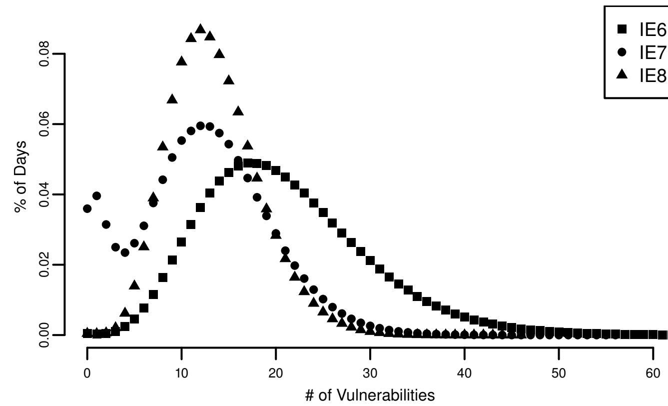


Figure 4.7. Days with given number of vulnerabilities (MS IE)

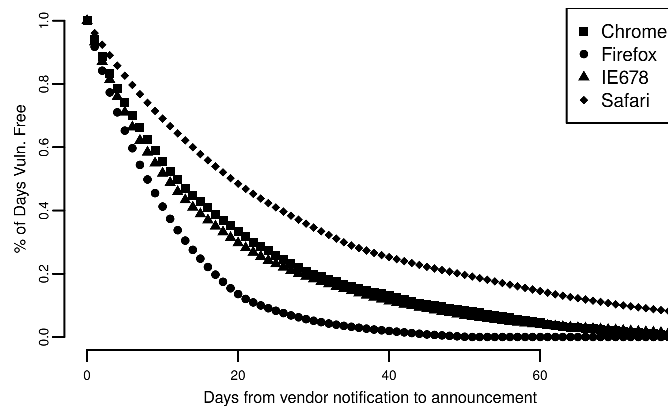


Figure 4.8. Vulnerability Free Days (VFD), as a function of lifespans

speculated to be the fact that Internet Explorer 7 and 8 have more common code than either have with version 6. Also, the Microsoft Security Development Life Cycle became a mandatory policy at Microsoft in 2004 (three years after the release of IE6, 2001, and two years before IE7, 2006) [9].

For estimating the VFD metric, the arrival rate of announcements and number of vulnerabilities announced per announcement are random variables distributed as described in Section 4.3.1. In Figure 4.8 the lifespan of vulnerabilities was varied from 1 day (vulnerabilities are addressed practically as soon as they are reported) to 182 days. The lifespan is varied along the horizontal axis, and the percentage of vulnerability free days is shown on the vertical axis. The goal was to examine the behavior of the VFD

metric as the result of different vulnerability lifespans for products.

The results are provided for Safari, Chrome, Firefox, and Internet Explorer (versions 6, 7, and 8 are treated as an aggregate since vendor behavior is being examined). The most interesting result is that even for a vulnerability lifespan of 45 days, the percent of days which are vulnerability free are less than 20% for Safari and less than 6% for the three other browsers. Even Safari, the best performing browser as judged by this metric, does not do well. When the lifespan is the length of those actually measured, approximately 75 days for Safari and 146 days for Internet Explorer, the VFD for all browsers is less than 10%. A poor performance by all browsers.

4.3.4 Comparisons Within a Vendor

As mentioned earlier, the metrics can be used to compare different products from within the same vendor. The allows for a measurement of the relative effectiveness of corporate coding standards and policies.

Table 4.6 shows the MAV metric simulated for both Apple Safari and QuickTime. The difference in MAV shows that, on average, the QuickTime developers are working on 2 fewer vulnerabilities than the Safari team. There are several possible explanations for this. It might be that the Safari browser, being the default browser for MacOS systems, is under more scrutiny by security researchers (with more time dedicated to examining it perhaps more vulnerabilities are found). From Table 4.4, the lifespan of vulnerabilities is not largely different (106.8 and 113.4 days for Safari and QuickTime, respectively), and the number of vulnerabilities announced per announcement day (Table 4.3) is 2.279 and 3.067. The biggest difference is in the mean days between announcements: 25.47 and 46.83 days for Safari and QuickTime, respectively (Table 4.2). Therefore the time between announcements of vulnerabilities is the single largest factor determining the MAV difference for these products. Figure 4.9 shows the distribution of MAV from the simulation; the curves almost overlap except for a small shift in peak.

Microsoft Internet Explorer and the Microsoft Office suite were also compared (Ta-

Vendor	Product	MAV	σ
Apple	Safari	9.55	8.58
	QuickTime	7.41	5.51
Microsoft	IE (all)	23.2	8.94
	IE 6	20.7	8.70
	IE 7	12.2	6.90
	IE 8	13.5	4.76
	Office	27.1	10.3

Table 4.6. Comparison of products within a vendor

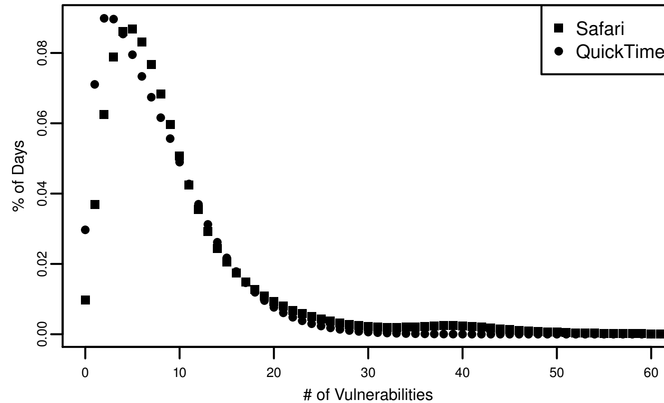


Figure 4.9. Comparison of Microsoft Products

ble 4.6. In this case, the browser (IE) has a considerably smaller MAV than the application suite (27.1 and 23.2 for Office and IE, respectively). If compared to individual releases of IE, the difference is even more pronounced. Analysis of IE data will be restricted to to comparisons against all IE data as this allows comparison over the same time period and presumably the same changes in corporate culture within Microsoft. Comparing the data for the two products from Table 4.2, 4.3, and 4.4, there are two primary factors in the higher MAV: vulnerabilities per announcement and vulnerability lifespan. In general there is a longer gap between announcements of vulnerabilities in office (14.95 and 24.11 days for IE and Office, respectively), but the number announced on an announcement day and the mean lifespan is much higher. On average, Microsoft takes 53.2 days longer to fix a vulnerability in the Office suite than for IE. Figure 4.10 shows the distribution of MAV from the simulation for the two products.

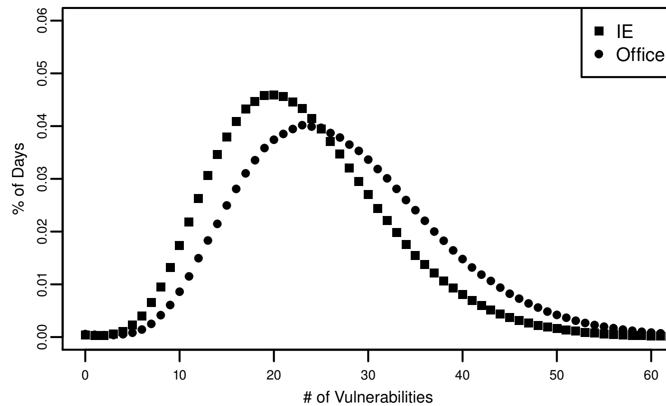


Figure 4.10. Comparison of Microsoft Products

4.3.5 Simplification of Metrics Calculations

Both the MAV and VFD metrics could be used by end-users when making software product purchasing or allowed use decisions. However, to gain use, they need to be able to be quickly calculated when the proper information is available. For end-users who are unable to deploy a simulation to calculate MAV and VFD it would be useful if there were short cut calculations to make first order estimates of the metrics. Two short cuts were formulated and compared the results to those from the simulation. The formulas may be found in (4.1) and (4.2).

$$MAV = \frac{\text{(Average Lifespan)(Average Reported)}}{\text{Average report rate}} \tag{4.1}$$

$$VFD = (1 - e^{-1})^{\text{MAV}} \approx 0.632^{\text{MAV}} \tag{4.2}$$

Equation (4.1) comes from treating the MAV as an average; the average number of vulnerabilities announced per day is the product of the number of announcements on an announcement day (Table 4.2) and the inverse of the days between announcements (Table 4.3). To get active vulnerabilities, that quantity is scaled by the average lifespan of a vulnerability (Table 4.4).

Equation (4.2) was derived using a least squares fit of an exponential model to the data and observing the resulting base was close to $1-e$. Additional terms could be added

Product	Simulated MAV	Short Cut MAV	% Error
MS Internet Explorer	23.2	24.6	6.09%
Mozilla Firefox	23.9	23.9	0.01%
Google Chrome	19.1	19.2	0.34%
Apple Safari	9.55	9.56	0.10%
MS Office	27.1	27.3	0.66%
Apple Quicktime	7.41	7.43	0.23%

Table 4.7. Comparison of simplified calculation of MAV to simulated.

to the model, but the goal here was to provide a simple calculation for use in estimation.

Table 4.7 shows the result of using the simulation data versus the simplified calculation using (4.1). The simplified version does a reasonable job of estimating the results of the simulation and is easy calculated directly from available data. The worst estimation performance from Table 4.7 is Internet Explorer (6% error), yet even this calculation is less than 1.5 vulnerabilities in magnitude.

The idea behind using MAV to compute VFD is from a software vendor point of view; namely, a vendor has some control over the number of developers assigned to addressing vulnerability reports. By adjusting the speed with which vulnerabilities are patched, a vendor can pick a target VFD probability and find the average lifespan needed to achieve it.

Figure 4.11 displays the simulated versus estimated VFD values. Ideally, the lines for each product would follow the line $y = x$; the departure from this is the estimation error. Generally, the curves follow a linear shape meaning that the first order effects of the simulation are captured by the estimation. The model fits well the behavior of VFD for Internet Explorer and Safari and somewhat less for Chrome and Firefox.

4.4 Discussion

This section discusses several observations made from examination of the ZDI/iDefense data and the NVD. In particular, the effect of severity of a vulnerability versus lifespan is examined (Section 4.4.1) and possible difference of severity scores based on dataset (Section 4.4.2). Finally, related work in the area is presented and contrasted with

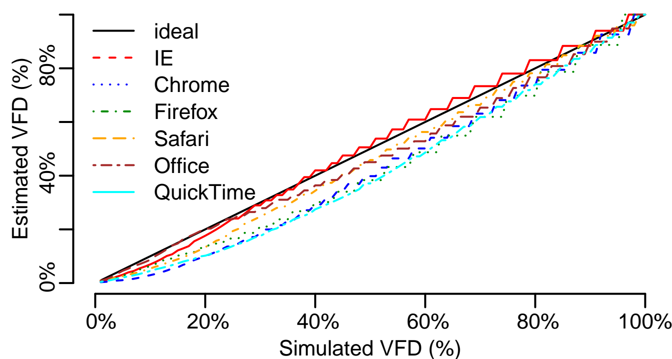


Figure 4.11. Estimation of VFD metric using shortcut formula

the research presented here (Section 4.4.3).

4.4.1 Severity Versus Lifespan

One of the widely used metrics for vulnerability severity is the Common Vulnerability Scoring System (CVSS) maintained by NIST[1]. For all vulnerabilities collected from iDefense and ZDI, the CVSS as recorded in the NVD was gathered. CVSS scores vulnerabilities along several dimensions: impact to confidentiality, integrity, and availability; access vector; access complexity; etc. The resulting score is meant to be an ordinal score of the severity of a vulnerability. The question was then, do vulnerabilities with higher impact (higher CVSS score) take longer to fix (have a longer lifespan from report to patch)?

Figure 4.12a shows the distribution of lifespans for each ordinal value of CVSS score for all products in the ZDI and iDefense data set. From this graph, it is not at all clear that CVSS score predicts lifespan; in other words, the severity of a vulnerability does not clearly predict how long it will take the vendor to fix it. It is true that vulnerabilities with the highest CVSS scores (9.3 and 10.0) also account for the most extreme lifespans, the distribution is highly skewed towards lower lifespans.

Figures 4.12b through 4.12g are subsets of the data set separated by product. There is no clear indication in any of the graphs that CVSS score and lifespan are related. In

[1] `http://www.first.org/cvss` [Accessed: January 21, 2013]

the case of Google Chrome, all 5 vulnerabilities have a CVSS score of 9.3, and Apple Safari has one vulnerability scored at 6.8 and the remaining 9 are scored at 9.3.

The conclusion is then that CVSS score is not a good indicator of the lifespan of a vulnerability. Practically this means that vendors take approximately the same time to fix vulnerabilities without respect to the impact on end-users.

4.4.2 Comparison of CVSS in Data Sets

It is possible that vulnerabilities collected by ZDI and iDefense differ in some way from those otherwise reported in the National Vulnerability Database. One of the few ways to compare the two data sets is by CVSS score. To test the hypothesis of a difference, a Kruskal–Wallis test was performed on CVSS scores separated by the factor of source (either directly from NVD or from the ZDI/iDefense data), and Table 4.8 shows the results. This particular test was chosen because of its non-parametric nature (less sensitivity to skewed distributions).

Interestingly, for non-browser products (Apple QuickTime and Microsoft Office), the hypothesis of a difference in location between CVSS scores in the NVD and those vulnerabilities bought by ZDI/iDefense failed to be rejected. However, in all cases, for the browsers the hypothesis that the CVSS scores is not shifted in location is rejected. The distribution of CVSS scores for the web browsers is significantly shifted upwards in the data from ZDI/iDefense; this means that the severity of those vulnerabilities is considerably higher.

Product	χ^2	p-value
Apple Safari	13.63	2.2×10^{-4}
Google Chrome	5.63	0.017
Microsoft Internet Explorer	27.54	1.5×10^{-7}
Mozilla Firefox	6.43	0.011
Apple QuickTime	0.67	0.411
Microsoft Office	0.42	0.517
All Products	657.1	$< 2.2 \times 10^{-16}$

Table 4.8. Tests for difference in CVSS scores based on data source ($df = 1$).

Also in Table 4.8, all CVSS scores from the NVD versus all scores of vulnerabilities

bought by ZDI/iDefense is compared. As with browsers, there is a significant shift upwards in severity for vulnerabilities in the ZDI/iDefense data set. This can be clearly seen in Figure 4.13. This figure also demonstrates the skewed nature of CVSS scores in both NVD and ZDI/iDefense. One possible explanation for the difference between NVD and ZDI/iDefense CVSS scores is that ZDI/iDefense may not be willing to purchase low impact vulnerabilities.

4.4.3 Related Work

Software life cycle metrics are a well studied aspect of development. These metrics concentrate on the rate at which defects are detected in the various stages of the life cycle of software. Less well understood are metrics for the security vulnerability life cycle.

Several approaches to understanding the life cycle of vulnerabilities have been undertaken over the past few years. The approaches fall mostly into two methods: examining one or a few software packages in detail or looking for large scale trends.

Ozment and Schechter [7], for example, falls into the former category. They examined the discovery of vulnerabilities in the OpenBSD operating system across several years and versions to determine whether it is getting fundamentally more secure over time. Their conclusion was that the rate of newly discovered vulnerabilities appeared to be slowing for "foundational" code.

Also in this category is Schryen, who examined 17 different products (open source and closed source) [20]. This work concentrated on the question of whether open source products are more secure than closed source products. Schryen concludes that there is no empirical evidence that open source products and closed source products differ significantly. Comparing Mozilla Firefox (open source) against Internet Explorer (closed source) based on the MAV and VFD, the same conclusion might be drawn.

Frei, et al. [48] is an example of the latter category where all vulnerabilities in the NVD and other sources are examined to find global trends. This work does not help, though, when considering individual products or vendors and comparing them.

Arnold, et al. [6] examined a single product: the Linux kernel. They found a significant number of software bugs that were later discovered to be vulnerabilities. These delayed impact vulnerabilities highlight the difficulties in obtaining accurate and verifiable dates for discovery of vulnerabilities. In the case of delayed impact vulnerabilities, the discoverer either did not check whether a bug was also a vulnerability or its impact was not realized until well after the bug was reported.

More recently, Clark, et al. [19] took a new approach where the first four vulnerabilities for a particular release of a particular piece of software were examined. Using this approach, they claim that extrinsic properties to software development are more indicative of vulnerability discovery than are intrinsic properties like software quality. Their approach is applied across vendors, open source versus closed source, etc.

Arora, et al. [11] examined the vulnerability life cycle by concentrating on an optimal policy for disclosure. Their work provides the model used for discussion of the life cycle in Section 4.1. However, the approach of optimizing the disclosure policy based on economic factors relies on many variables which are simply not credibly known.

This research is not concerned with an examination of the entire life cycle of vulnerabilities. Instead, a method for ranking products across vendors or products within a single vendor on the basis of their raw number of vulnerabilities and the speed with which they address them is examined. The result thus far has been to demonstrate the applicability of the metrics against a small set of products.

As far as vulnerability metrics are concerned, several reports concentrate on the total number of vulnerabilities announced over a given time (per year or per half year) and the number of fixed vulnerabilities over the same time for example: [49, 50]. At a gross level, this information is similar to our MAV metric, but it is not as granular. A vulnerability can last for a year or a day between report and patch and the total announced minus the number fixed will stay the same using this type of counting. The MAV metric takes both the total number of announced vulnerabilities and their lifespan into account in per

day units.

Finally, an interesting metric was proposed by Acer and Jackson [18], which attempts to combine: patch deployment, vulnerability severity, and user-installed browser plug-ins. The authors gather “user-agent” strings reported by browsers visiting a site created by the authors. From this, the number of users who are not completely up to date with patches are counted, and the “best” browser is the one with the fewest number of users who are not fully patched. However, this method depends on random sampling (possibly achievable with strategically placed collectors) and only addresses software which report complete version information. For non-browser products, it is not clear how measurements could be conducted, and even for browsers, the authors found that Internet Explorer does not report all of the necessary information.

4.5 Conclusion

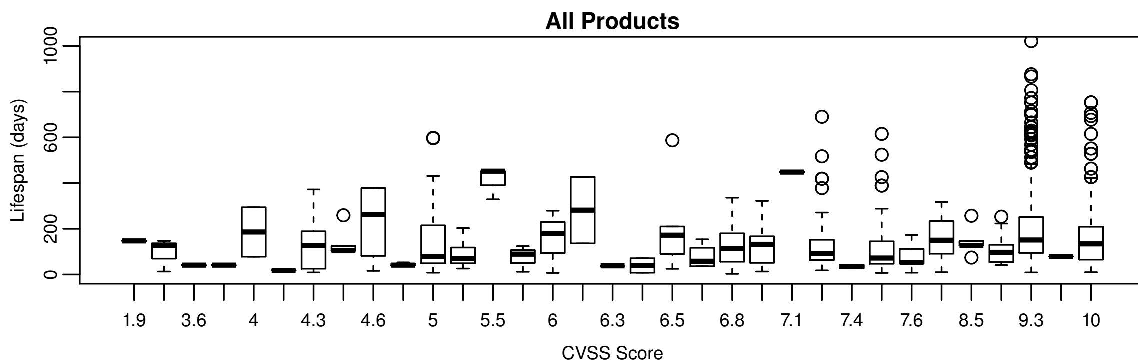
Two new software vulnerability exposure metrics were proposed with the end-user in mind. Both Vulnerability Free Days and Median Active Vulnerabilities were demonstrated in a case study of the four browsers Safari, Chrome, Firefox, and Internet Explorer. Estimation values for the metrics were generated through simulation. Short cut estimations were shown to be practical. Based on the derived exposure metrics for each browser, there are large differences in vulnerability exposure, with Safari having the lowest exposure.

The metrics defined in this article provide a benchmark to compare one vendor against another. Internally, product groups can compare themselves with other product groups within the same vendor. Just as companies doing hazardous work strive for long stretches with no safety accidents, striving for high vulnerability free days or low median active vulnerabilities could be a development goal itself.

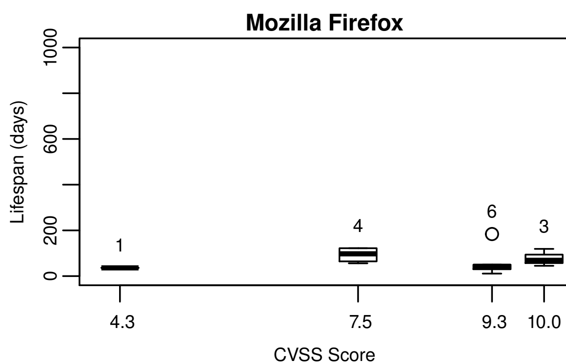
In terms of the metrics defined in this article, vulnerability researchers have the ability to keep the software vendors honest. The researchers know when they discovered a vulnerability and more importantly when they reported it to the vendor. They are also

best positioned to determine whether a particular patch or solution fixes the problem. Currently, estimating VFD and MAV requires no help from software vendors, but the estimates are not as precise as could be with more comprehensive data.

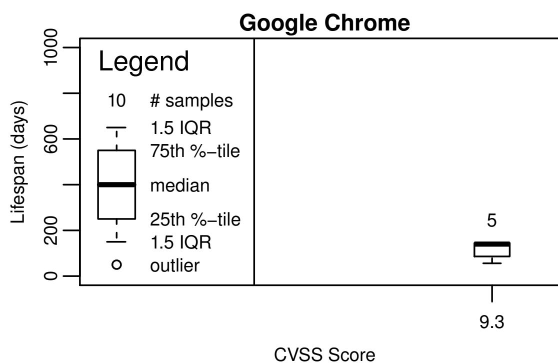
The exposure metrics are sensitive to both lifespans and the number of vulnerabilities being discovered and reported. Firefox, which produces patches quickest, still has one of the worst vulnerability exposures because so many vulnerabilities are discovered and reported. It was also noted that it may not be realistic for any of the browsers to get to even 50% Vulnerability Free Days.



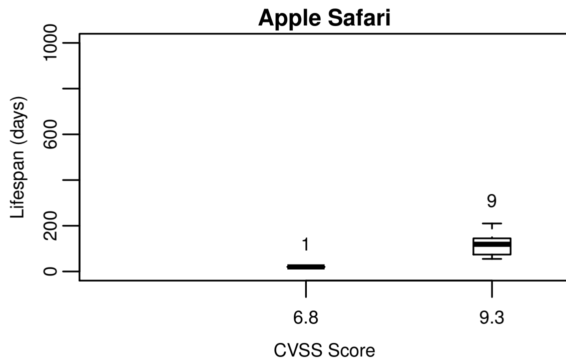
(a)



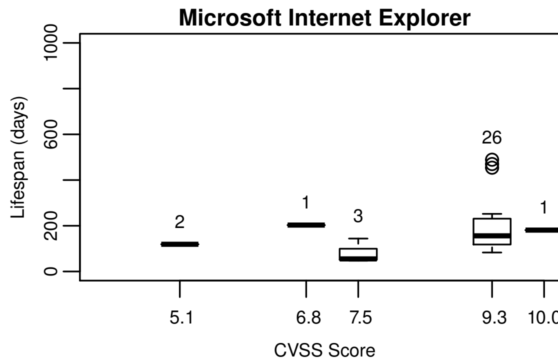
(b)



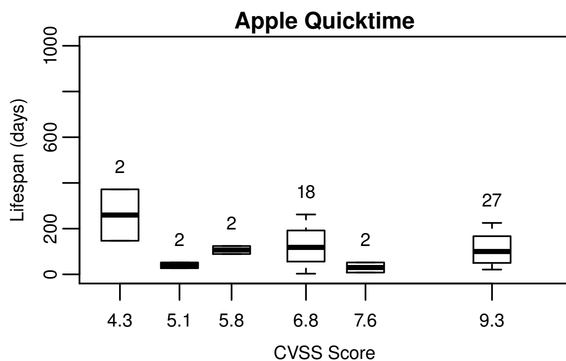
(c)



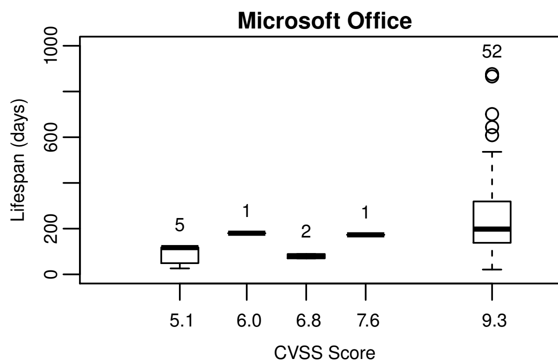
(d)



(e)



(f)



(g)

Figure 4.12. Comparing CVSS score to vulnerability lifespan

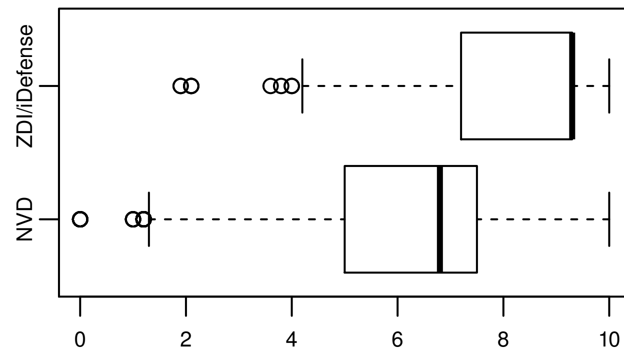


Figure 4.13. Distribution of CVSS scores: NVD versus ZDI/iDefense.

CHAPTER 5

Estimating Software Vulnerabilities

This chapter uses the bug database of an open source product to estimate the number of bugs which have yet to discovered to be vulnerabilities. It is revised and extended from the paper published in [51].

5.1 Introduction

Software vulnerabilities are an important aspect of managing current and future information technology infrastructures. New vulnerabilities are discovered in software products every day and publicly reported through a variety of publicly accessible vulnerability databases. What is not known, however, is whether the number of publicly reported vulnerabilities for a software product can reasonably be used as a stand-in for the security of the product. Without knowing this information, risk estimates for individual products are problematic.

The definition of software vulnerability for this p comes from Krsul [4] and Ozment [5]: "an instance of [a mistake] in the specification, development, or configuration of software such that its execution can violate the [explicit or implicit] security policy." All software defects are not vulnerabilities by this definition, but vulnerabilities are a subset of software defects; the important factor is whether the security policy may be violated. As discussed in Section 5.3.5, this definition is further constrained to err on the side of conservative estimates of the yet to be identified vulnerabilities.

This chapter describes an experiment, performed on a popular open source product, to estimate the number of bugs misclassified as not being vulnerabilities. A subset of the product's bug reports were selected for detailed scrutiny. The selected bug reports and the associated source code were then carefully analyzed to determine what portion of the selected bugs had been misclassified as not vulnerabilities. Based on these results, an extrapolation to the full population of bug reports is performed to provide an estimate

of the total population of misclassifed bugs (those which should have been identified as vulnerabilities). Based on these estimates, it is estimated that there exist between 657% and 672% more vulnerabilities than the 76 that are currently documented for the product in the National Vulnerability Database (NVD)[1] and the Open Source Vulnerability Database (OSVDB)[2]. (Note: The few MySQL vulnerabilities listed in the OSVDB all mapped to vulnerabilities found in the NVD, so for the rest of this chapter only the NVD vulnerabilities will be reported).

As a consequence, there is reason to believe that the number of publicly reported vulnerabilities in a software product is far less than those that have been discovered as bugs but misclassified as not vulnerabilities. This leads to the conclusion that human efforts in identifying bugs as vulnerabilities may not be very effective, and that using the number of reported vulnerabilities as a measure of a software product's relative risk is highly questionable.

The remainder of this chapter is organized as follows. Section 5.3 describes the experimental goals and setup including a description of the product evaluated, its bug database, and the evaluation process for each bug. Section 5.4 describes the results of the experiment, and Section 5.5 includes further analyses and discussion. Section 5.6 provides the conclusion.

5.2 Hidden Impact Bugs

Arnold et al. coined the phrase "hidden impact vulnerabilities" to describe bug reports which are later discovered to also describe software vulnerabilities [6]. This research refers to the same thing as hidden impact bugs since it is the bugs which have the (initially) hidden impact of being vulnerabilities. Arnold et al. examined the Linux kernel bug database from January 2006 to December 2008 and found 56 bugs that had at least two weeks between a patch being committed and the discovery that the bug

[1] National Vulnerability Database, `http://nvd.nist.gov/`

[2] Open Source Vulnerability Database, `http://www.osvdb.org/`

represented a security vulnerability.

In [52], Arnold et al.'s work was extended to include the timespan from January 2009 to 30 April 2011 and to include the MySQL Database Management System. The question was whether the the number of hidden impact bugs was increasing over time and whether a significant number existed in other products. The findings discussed below support the idea that there is an increasing number of hidden impact vulnerabilities and that a significant number exist in other products.

5.2.1 Linux Kernel Vulnerability Analysis

In their study Arnold et al. used a database of Linux kernel vulnerabilities for the first time period (i.e. from the 1st of January 2006 to the 31st of December 2008). For this time period the Linux kernel had 218 vulnerabilities reported out of which 56 (25.69%) had an impact delay of at least 2 weeks. Impact delay was defined as the time from the public disclosure of the bug in the form of a patch to the time a CVE was assigned to the bug because it had now been identified as a vulnerability. It was also shown that for any given day in the time period there was an average of 8.5 hidden impact vulnerabilities present that affected the Linux kernel.

The number of reported vulnerabilities in software has been increasing over the past few years [53, 54]. In order to evaluate whether the number of hidden impact vulnerabilities has also increased over time, a similar analysis was performed for Linux kernel vulnerabilities for the second time period (i.e. from the 1st of January 2009 to the 30th of April 2011). For this analysis specific rules were applied to the vulnerability database downloaded from [53]. Vulnerabilities that affected 1) multiple processors, 2) multiple distributions and 3) Linux kernel 2.6 and above, were selected for the vulnerability database for the time period. Vulnerabilities that affected only a single processor were excluded because these vulnerabilities affected only a small subset of users and it is difficult to identify whether they were caused by a kernel issue. Similarly, vulnerabilities that affected only one distribution were excluded because there is no way of clarifying if

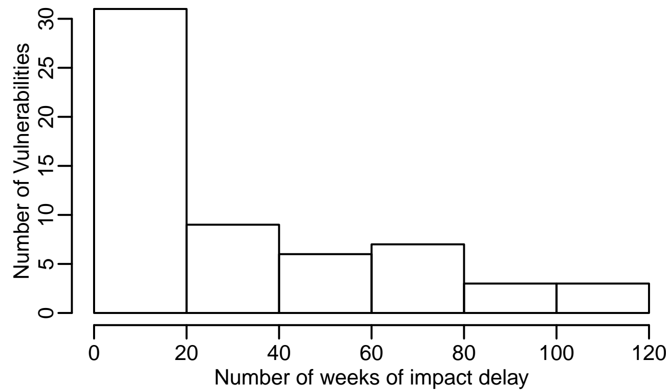


Figure 5.1. Number of hidden impact vulnerabilities by impact delay for Linux kernel (January 2009 to April 2011)

Table 5.1. Hidden impact vulnerabilities (Linux kernel)

	2006 Jan. – 2008 Dec. (First time period)	**2009 Jan. – 2011 Apr. (Second time period)**	**Total**
Total	218	185	403
At least 2 weeks of impact delay	56 (25.69%)	73 (39.46%)	129 (32.01%)
At least 4 weeks of impact delay	38 (17.43%)	55 (29.73%)	93 (23.08%)
At least 4 weeks of impact delay	31 (14.22%)	29 (15.68%)	60 (14.99%)

the vulnerability was due to a kernel issue. Vulnerabilities that affected Linux kernel 2.6 and above were selected because it was the latest version available in 2006. These rules also seem to match the rules applied in [6]. Thus the vulnerability database contained 185 vulnerabilities for the second time period, which is a 15% reduction from the first time period. However, the number of vulnerabilities with at least 2 weeks of impact delay increased to 73 (39.46%). Figure 5.1 shows the number of hidden impact vulnerabilities with different impact delays. Table 5.1 shows the number of vulnerabilities with at least 2, 4 and 8 weeks of impact delay for the two time periods.

Further, on any given day, there were 9.8 hidden impact vulnerabilities in existence on average during the second time period. Figure 5.2 shows the number of hidden impact vulnerabilities that existed on each day for the second time period.

Thus, the number of hidden impact vulnerabilities in the Linux kernel has increased

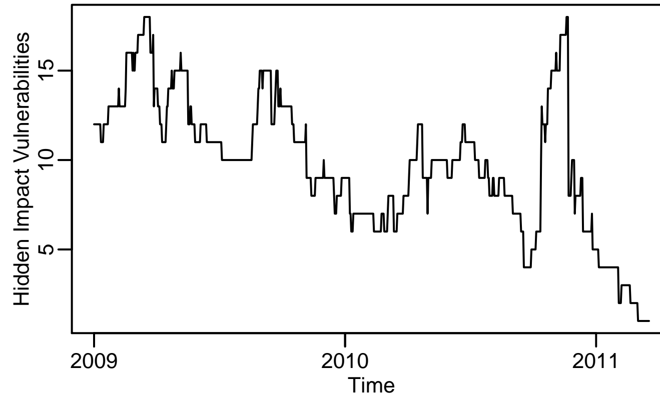


Figure 5.2. Number of hidden impact vulnerabilities that existed per day for the Linux kernel (January 2009 to April 2011)

Table 5.2. Hidden impact vulnerabilities (MySQL)

	2006 Jan. – 2008 Dec. (First time period)	**2009 Jan. – 2011 Apr. (Second time period)**	**Total**
Total	37	29	66
At least 2 weeks of impact delay	22 (59.46%)	19 (65.52%)	41 (62.12%)
At least 4 weeks of impact delay	21 (56.76%)	19 (65.52%)	40 (60.62%)
At least 4 weeks of impact delay	17 (45.95%)	16 (55.17%)	33 (50%)

in both percentage and magnitude for the 2009 to 2011 time period. Furthermore, the average number of hidden impact vulnerabilities in existence per each day has also increased for the same time period.

5.2.2 MySQL Vulnerability Analysis

To expand on the knowledge gained from examining a single product (the Linux kernel), the MySQL database server was analyzed. Like Linux, MySQL has a public database of bugs and a significant number of vulnerabilities in the MITRE CVE database.

Using the same criteria as discussed in Section 5.2.1 for the first time period, there were 37 vulnerabilities in the MITRE CVE database out of which 22 (59.5%) had an impact delay of at least 2 weeks (see Table 5.2). An average of 3.45 hidden impact vulnerabilities affected the MySQL database server per day for the same time period.

For the second time period, 29 vulnerabilities were reported and 19 (65.5%) of these

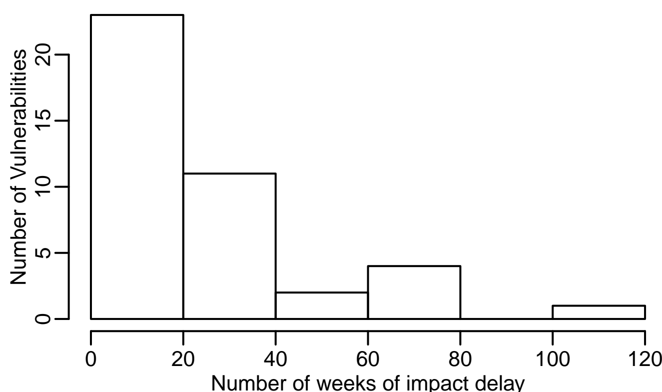


Figure 5.3. Number of hidden impact vulnerabilities by impact delay for MySQL (December 2003 to April 2011)

were hidden impact vulnerabilities that had an impact delay of at least 2 weeks. Although the number of hidden impact vulnerabilities has not increased in absolute terms, it has increased percentagewise in the 2009 to 2011 time period.

Figure 5.3 shows the number of vulnerabilities by impact delay for the MySQL database server. Comparing Figure 5.1 and Figure 5.3 shows that the median impact delay time for MySQL is much higher (11 weeks for Linux and 20 weeks for MySQL). Also, the distribution is of a different shape which may reflect the different priorities of the developers of the two projects.

Finally, Figure 5.4 shows the number of hidden impact vulnerabilities on a given day for MySQL for the time period from January 2009 to April 2011. At any given day during the second time period, on average there existed 3.75 hidden impact vulnerabilities for the MySQL database server.

Thus, similar to Linux, MySQL hidden impact vulnerabilities account for a significant portion of the total number of vulnerabilities and the percentage of hidden impact vulnerabilities has increased in the second time period.

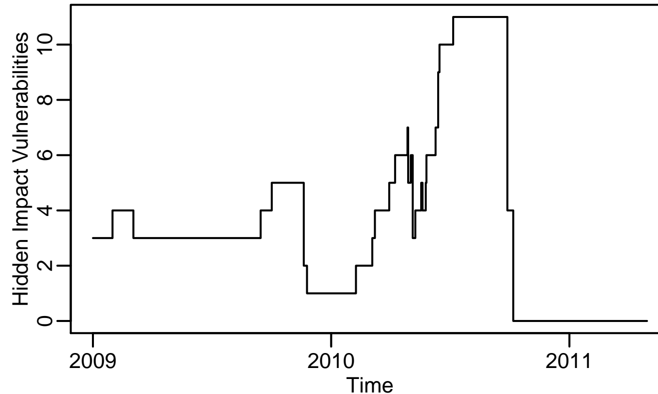


Figure 5.4. Number of hidden impact vulnerabilities that existed per day for the MySQL database (January 2009 to April 2011)

5.3 Experimental Goals and Setup

The goal of the experiment was to determine if the total number of discovered vulnerabilities for individual software products (including those bugs misclassified as not being vulnerabilities) is much larger than the currently reported number of vulnerabilities in the NVD.

If the number of misclassified bugs is large then many analysis based on reported vulnerabilities, such as the half life of vulnerabilities [13] or rate of vulnerability discovery [15], are called into question. The experimental results may also suggest a need for improved bug classification in order to provide much stronger vulnerability data sets for vulnerability research; reduce vulnerability attack time windows; and allow more effective risk comparisons between software products.

Further, the results may also indicate whether or not the largely human process of identifying which bugs are also vulnerabilities is effective, and perhaps suggest a need for improved bug triage tools to aid more effective identification of vulnerabilities by software developers. In addition, a significantly large number of misclassified bugs could potentially provide new insight into vulnerability attributes and an associated opportunity for improvement in vulnerability identification tools such as static analyzers.

Each of the sub-sections below describes a part of the process used to conduct the

experiment: product selection (5.3.1), MySQL bug database overview (5.3.2), selected subset of MySQL server software (5.3.3), MySQL bug scoring process and results(5.3.4), and determining the number of misclassified bugs (5.3.5).

5.3.1 Product Selection

The first step in this experiment was to select a software product for evaluation. The ideal product would satisfy a number of properties. These properties consist of:

- 1 – product pervasiveness,
- 2 – large number of announced vulnerabilities,
- 3 – source code availability,
- 4 – publicly accessible bug reports.

Pervasiveness was desired so that there would be some confidence that the product's code base, and its associated bugs, had received significant security attention. A fairly large number of publicly announced vulnerabilities was needed to both confirm that the code base had received security scrutiny, and to allow credible comparisons relative to the number of new vulnerabilities identified (if any). The product's source code needed to be available so that a bug could be reproduced and effectively evaluated for its security impact. And, of course, the bug reports were required to be publicly accessible so that those bugs classified as not vulnerabilities could be reassessed.

As discussed below, the MySQL product reasonably meets each of the four desired properties.

First, MySQL is widely deployed. It is part of the "LAMP" configuration of Linux (Linux, Apache, MySQL, and PHP) which is used on a large number of deployed web sites.

Second, at the time this experiment was conducted, the National Vulnerability Database listed 107 vulnerabilities for MySQL announced from 1998 through 2011. The

Table 5.3. Number of first published vulnerabilities and bug reports by year for MySQL

Year	Vulnerabilities	Bugs
1998	1	—
1999	0	—
2000	3	—
2001	6	—
2002	8	15
2003	5	2225
2004	9	5354
2005	11	8282
2006	14	8295
2007	14	7558
2008	7	7596
2009	7	7392
2010	6	8241
2011	16	—
Sub Total	107	54958
No such bug		9
No access to bug		4480
Total bug reports		59447

number of vulnerabilities published per year is shown in the second column of Table 5.3.

Third, the source code for all versions of the MySQL software was available for analysis. Although MySQL does not maintain old software source code on their servers, it was possible to reconstruct the state of the code at each release using the available source code management system (Bazaar).

Fourth, although the preference would be to have all bug reports available for evaluation this did not appear feasible after a quick evaluation of a number of products. Fortunately, over 92% of MySQL bug reports were publicly accessible and this seemed to reasonably meet the desired property.

Consequently, the MySQL software product was selected for evaluation.

5.3.2 MySQL Bug Database Overview

A public bug database was setup for MySQL on September 12, 2002, and contains 59447 unique bug identifiers as of the end of 2010. The distribution of bug reports by

year is shown in the third column of Table 5.3. Of the 59447 bug reports, 4480 (7.5%) are not accessible to the public, and 9 (0.02%) simply do not exist.

The non-public bug reports perhaps contain either private customer data or information that may contain information regarding vulnerabilities. Since there is no information regarding these bugs publicly available, these bug identifiers were discarded. This left 54958 valid bug identifiers for consideration.

Unfortunately, no small team can examine that many bug reports with sufficient diligence to credibly identify those bugs which are also vulnerabilities. The analysis required is too detailed and the rate of vulnerability occurrence is too small. For instance, during the years 2002 to 2010 (Table 5.3), the ratio of reported vulnerabilities to bug reports is approximately 0.0018 (about 1 : 556). For this and a few other reasons, as explained below, it was decided to further narrow the evaluation focus to a subset of MySQL server software.

5.3.3 Selected Subset of MySQL Server Software

It was decided to focus on bugs affecting the MySQL server itself (not client programs, support scripts, or third party applications) based on matching the category field in the bug database. The decision to exclude client bug reports was made because in a given installation, a wide variety of client applications may be used to access the server and the choice was to focus on the most commonly deployed aspects of the package (the part of the package with the most potential impact on end-users).

Bugs relating only to release 3.X of the database server software were discarded because the bug database barely covers the development of this branch; the final release of the 3.X branch was version 3.23.58 on September 11, 2003 (slightly less than a year after the bug database was created).

Bugs only affecting the “telco” and “Falcon” branches of MySQL server software were also discarded. Neither of these branches were released officially and thus vulnerabilities in these branches would not have wide spread implications and the software may not

have undergone a rigorous security vetting.

The *telco* branch was a fork of MySQL 5.1, integrating the MySQL Cluster technology of highly available distributed operation. At least some of the code has since been integrated into official releases so only bugs in the early development branch are excluded from this study.

The Falcon storage engine was a part of the MySQL 6.0 branch which had only one "official" release. Since the merger with Oracle, this development branch has been abandoned [55]. Thus all bugs in this branch are excluded from this study.

These reasonable exclusions allowed a focus on the four major development branches of MySQL server (4.X, 5.0, 5.1, and 5.5) all of which were formally released for general use; not on development branches which may have been publicly available but not intended for production environments (so called alpha and beta releases); and overlap the time period covered by the bug database.

Applying the above exclusions to the MySQL bug database eliminated 24230 (44.1%) of the 54958 bugs leaving 30728 bugs still under consideration (second column of Table 5.4). The same exclusions were applied to the MySQL vulnerabilities listed in the NVD and additionally discarded the few disputed vulnerabilities. This process resulted in the elimination of 31 of the 107 vulnerabilities in NVD, which left 76 reported vulnerabilities in the MySQL server software under evaluation (third column of Table 5.4). Note that with these exclusions, the ratio of reported vulnerabilities to bug reports has increased to approximately 0.0025 (about 1 : 404).

Table 5.4. Applying rules to Bugs/NVD CVEs.

	Bugs	CVEs
Starting	54958	107
Non-server	20893	13
Wrong version	3337	15
Disputed	—	3
Remaining	30728	76

5.3.4 MySQL Bug Scoring Process

In the remaining set of 30728 bug reports under consideration, it was not known how many, if any, were unidentified vulnerabilities. Since there were still too many bugs to carefully evaluate, the decision was made to create a scoring system intended to indicate the likelihood that a bug might be misclassified and actually be a vulnerability. The scoring system was created by two expert vulnerability researchers and involved a two step process.

In the first step, the experts specified a set of text strings which, if found in a bug report, might indicate the bug was more or less likely to be a vulnerability (e.g. a bug involving illegal instruction exceptions). The set of text strings can be found in column one of Table 5.5.

In the second step, each bug's score was initialized to 0 and the bug report was then searched for the specified text strings. When a text string was matched the associated weight of the text string was added to the bugs overall score. A given bug report could have matches with zero or more text strings. The full listing of the text strings assigned weights can be found in the second column of Table 5.5. The table also shows in column three how many of the 30728 bug reports were found to have each of the associated text strings.

Before applying the scoring system to the 30728 bugs under consideration it was tested to assess potential effectiveness. The set of bug reports associated with publicly identified MySQL server vulnerabilities were scored, and then the set of remaining bug reports which were not identified as vulnerabilities were scored. These two sets of scores were then compared with the expectation that the scores of the publicly identified vulnerabilities would be higher than the scores of the bugs which had not been identified as vulnerabilities.

Table 5.5. Text string weighting and occurrence counts

Text Strings	Weight	How Many	Description
submitted < Jan 1, 2003	-100	10	Bugs submitted before Jan 1, 2003
'%signal 11%', '%sig=11%', '%egfault%', '%handle_segfault%'	+100	1755	Reports showing signal 11 (SEGV)
'%signal%', '%sig=%'	+50	2433	Reports showing POSIX signal aborts
'%pthread_kill%'	+50	342	Calls to `pthread_kill()`
'%signal 6%', '%sig=6%'	-50	524	Calls to `abort()`
'%__assert_fail%'	-20	326	Calls to `__assert_fail()`
'%-I../../..%'	-50	33	Compiler errors
'%write_core%'	+100	312	Calls to `write_core()`
'%corruption%'	+100	967	Reports mentioning "corruption"
'%deref%'	+100	82	Reports mentioning "deref"
'%double free%'	+120	45	Double free assertion failures
'%Error::%'	+100	9	Calls to error routines
'%exploit%'	+120	73	Reports mentioning "exploit"
'%gcc%'	-20	1134	GCC errors
'%GDB is free software%'	+100	46	GDB backtraces
'%signal 4%', '%sig=4%'	+200	9	Illegal instruction traps
'%mysqldump%'	-50	1171	Uses of "mysqldump"
'%0x000000%'	+20	977	NULL addresses in backtrace
'%pointer%'	+20	2041	Mentions "pointer"
'%raise%'	-50	747	Raises exception
'%main_security_ctx%'	+100	7	Uses of `main_security_ctx()`

Scoring System Test

To assess the effectiveness of the scoring system, the references of the 107 known CVEs for MySQL server (described in Section 5.3.1) were examined. Specifically, each known vulnerability for was searcjed fpr references to specific MySQL bug reports. References from the NVD to other vulnerability databases (OSVDB, ISS Xforce, etc.) were searched for references to bug reports. The MySQL bug database was searched for references to CVEs. In all, 74 mappings from bug report to CVE and/or vice versa were found and the results are in the top half of Table 5.6).

Some of the 74 bugs where then expluded for a variety of reasons. Bugs were unrelated to MySQL were exluded (e.g. a CVE identifier appears in a Perl version string in bug 19532). Multiple CVEs (CVE-2008-4097, CVE-2008-4098, and CVE-2009-4030) point to MySQL bug 32167 so the latter two mappings were excluded. CVEs referencing bugs that were not publicly accessible were also excluded (e.g. CVE-2010-3838 maps to bug 54461 which is not accessible to the public). All of these exclusions are summarized in the bottom half of Table 5.6.

The scoring system as described in Section 5.3.4 was then applied separately to the

Table 5.6. Break down of CVE/Bug identifier mappings

Mappings	Number
CVE $\Rightarrow$ bug	26
bug $\Rightarrow$ CVE	9
CVE $\Leftrightarrow$ bug	39
Total References	74
Exclusion	
not MySQL related	3
duplicate (CVE $\Rightarrow$ bug)	2
third party	4
disputed in NVD	3
client application	4
no access to bug	5
wrong version	3
Total Excluded	26
Remaining	48

48 bug reports for identified vulnerabilities, and the bug reports for which there is no associated vulnerability. A one-sided Wilcoxon Rank Sum Test of the two populations was performed. The hypothesis that the medians of the two distributions are equal is rejected (Table 5.7). The scoring system scored bug reports significantly higher for known vulnerabilities than for other bugs. This means that on the surface at least, the scoring system shows some preference for vulnerabilities. Figure 5.5 shows the distribution, mean, and median for both populations.

Table 5.7. Test of scoring system preference for vulnerabilities.

		score	
population	N	mean	median
mapped	48	75.83	0
not mapped	30680	12.23	10
Total	30728		
Wilcoxon W	1006742		
p-value	2.145×10^{-10} (one tail)		

The scoring system was applied to the MySQL server bug reports under consideration.

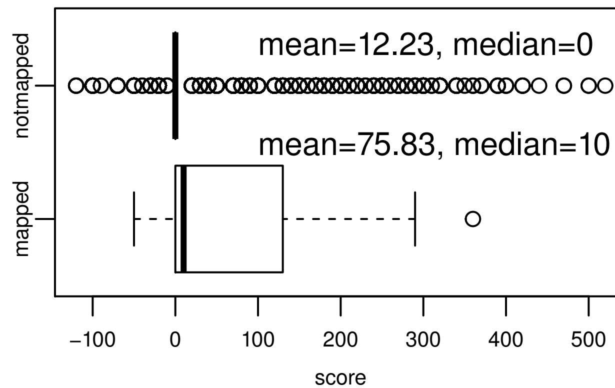


Figure 5.5. Distribution of bugs mapped to vulnerabilities and otherwise.

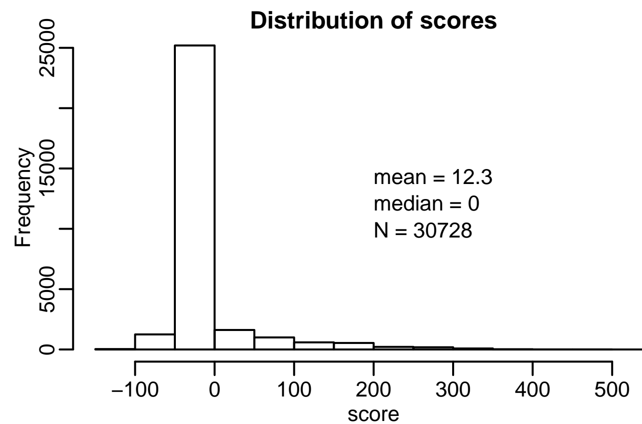


Figure 5.6. Distribution of MySQL server bug scores

Scoring System Results for MySQL Server

The application of the bug report scoring system produced the distribution of scores shown in Figure 5.6. The vast majority of scores fall in the interval $(-50, 0]$. Figure 5.7 zooms in on the distribution of the 4252 bugs with scores greater than 0.

Unfortunately, 4252 (13.8% of the 30728 server bugs under consideration) was still too many to evaluate. The bugs were divided into four disjoint groups based on score (second column of Table 5.8). The percent of misclassified bugs (i.e. vulnerabilities) within each group was then estimated.

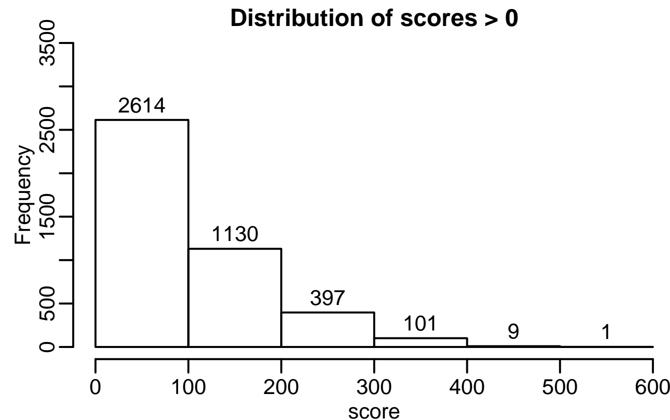


Figure 5.7. Distribution of MySQL server bug scores greater than 0

5.3.5 Determining Number of Misclassified Bugs

The population of bugs in each of the four groups mentioned above in Section 5.3.4 were separately sampled except for Group 3 where the population was small enough to examine all bugs in the Group. The population size and sample size for each group is shown in the third and fourth columns of Table 5.8 respectively. Each of the sampled bugs was carefully analyzed to determine if it was also a vulnerability.

Table 5.8. Examined bug report totals

Group	Score	Total Size	Sample Size
0	≤ 0	26476	160
1	$(0, 100]$	2614	46
2	$(100, 400]$	1628	55
3	> 400	10	10
Total		30728	271

The analyses assumes that the adversary would be able to execute arbitrary queries as an authenticated user. The reason for this assumption is the fact that a typical deployment of MySQL involves creation of database and user, and then web software is used to provide access to the database. The created user often has full access to the database.

Creation of a reliable exploit for a vulnerability requires a significant amount of work. For the purposes of this experiment, it was decided to stop the evaluation short of this

point. Instead, using terminology similar to Microsoft's Exploitability Index (EI) [56], evaluation stopped at the point of determining whether an exploit was likely to be possible or not. The Microsoft's EI can be one of three values[3]:

- 1 – Consistent exploit code likely,
- 2 – Inconsistent exploit code likely, and
- 3 – Functioning exploit code unlikely.

To error on the side of conservative estimates, vulnerabilities for which it was believed exploit code was unlikely (EI 3) were not included in the vulnerability counting and estimations.

Similarly, to error on the conservative side and also proactively dampen potential criticisms that primarily uninteresting vulnerabilities were found, it was decided to not include Denial-of-Service vulnerabilities in counts and estimations. For instance, a basic NULL pointer dereference is, without additional resource manipulation (e.g. memory), not exploitable [57]. Dereferencing a NULL pointer will simply cause the program to crash. Such bugs, by themselves, can be used to do little else than create a Denial-of-Service. In brief, only those vulnerabilities which are likely to allow violations of security policy and/or arbitrary code execution were counted.

To summarize the overall conservative approach taken for determining whether a bug was a vulnerability, the bug:

- must affect the MySQL server software (client programs out of scope),
- must be in a released version of the software,
- must likely be susceptible to exploit code,
- must not be a denial-of-service.

[3]The Exploitability Index was "clarified" by Microsoft on December 13, 2011 and the definitions used here are the original definitions.

5.4 Experimental Results

After careful evaluation of the selected bug reports from each scoring group, 12 bug reports were identified that had been misclassified as not vulnerabilities. Without even extrapolating to the entire bug population, this represents a 15.8% increase over the current publicly reported vulnerabilities for the MySQL server code under evaluation. After extrapolation to the total set of MySQL server bug reports, it is estimated that there are between 499 and 587 bugs which have been misclassified. This represents an estimated increase in vulnerabilities ranging from 657% to 672%.

In Section 5.4.1 the evaluation steps for the bugs sampled from each scoring group are described. Each step represents the identification of one or more bug attributes. In Section 5.4.2 the empirical results from examining the sampled bug reports to determine the number of vulnerabilities present are provided. In order to be sensitive to the concerns of end users, the newly identified vulnerabilities are not described. However, the bug identifiers for the sampled bugs within each scoring group are provided in Appendix A in the hope that other research groups will be motivated to duplicate and extend this research. In Section 5.4.3, these empirical results are used to extrapolate the statistical bounds on the number of misclassified bugs (i.e. vulnerabilities yet to be identified as such) in MySQL server code.

5.4.1 Vulnerability Evaluation of Sampled Bugs

The number of sampled bugs from each bug report scoring group are presented in the third column of Table 5.9. A maximum four step process was instituted for evaluating each of the sampled bugs.

The first evaluation step was to read each sampled bug report and determine if the bug was either a feature request or assessed to really not be a bug. A ”feature request” attribute, column two of Table 5.10, is simply a request from a user to modify software that does not currently cause any unintended behavior of the server (e.g. Bug #26602 where a user suggests using a single lock variable instead of several). Outside of Group 0

Table 5.9. Newly identified vulnerabilities by scoring group

Group	Score	Sampled	Vulnerabilities
0	≤ 0	160	1 (0.6%)
1	$(0, 100]$	46	4 (8.7%)
2	$(100, 400]$	55	5 (9.1%)
3	> 400	10	2 (20.0%)
Total		271	12 (4.4%)

(lowest scored vulnerabilities) feature requests were not found.

The "not a bug" attribute, column three of Table 5.10, indicates that while the user reporting the bug believes the observed behavior is abnormal, it is not considered to be so by the MySQL developers. For example, bug #29033 describes a user expectation problem; the database was behaving as intended but not as the user hoped. This bug attribute was only found in the sampled bugs of the lowest two scoring groups.

All sampled bugs identified as a feature request or not a bug were set aside with no further analysis devoted to them. This first step was relatively quick and easy.

The second evaluation step was to determine if bug could be reproduced if not already excluded in step 1. The "not reproducible" attribute, column four of Table 5.10, describes bugs that could not be reproduced by the authors. In the case of bug #10918, a machine running SCO UNIX was not available to the authors, and for bug #24429, there was simply not enough information provided in the bug report to reproduce the problem described by the submitter.

Those sampled bugs which could not be reproduced received no further analysis. This second step varied in difficulty with some bugs easy to reproduce; some bugs difficult to reproduce; and some which were not reproducible.

The third evaluation step was to review the sampled bugs remaining after the first two steps and determine if they represented NULL pointer dereferences, column five of Table 5.10. As described in Section 5.3.5, these bugs are generally not exploitable without additional resource manipulation. So if the bug was a NULL pointer dereference then further analysis was done to determine if the bug could be exploited through some

manipulation of memory such as is done in heap spraying [58]. This third step generally required a significant investment of time.

In the fourth evaluation step all remaining sampled bugs, column five of Table 5.10, received in depth vulnerability analyses. Each required an investment of time and expertise roughly similar to, or exceeding, that used in analysis of NULL pointer dereferences.

Table 5.10. Types of bugs within each group.

Group	Feature Request	Not A Bug	Not Reproducible	NULL Pointer	Deeper Analyses
0	8	12	10	3	127
1	0	1	8	1	36
2	0	0	23	8	24
3	0	0	6	1	3
Total	8	13	47	13	190

5.4.2 Newly Identified Vulnerabilities

In all, 12 previously unidentified vulnerabilities were found using the MySQL bug database by first searching through the sampled bug reports and then examining the affected code. The process for doing this was laborious, requiring duplication of the relevant portion of the environment in which the bug was reported and source code analysis by an experienced vulnerability researcher.

Table 5.9 shows the break down of how the newly identified vulnerabilities fell within each of the scoring groups described in Section 5.3.4. Most of the vulnerabilities fell in groups 1 through 3 (further evidence that the scoring system was effective); however, one vulnerability was found in Group 0 (score ≤ 0) after analyzing 160 bug reports in that Group.

5.4.3 Extrapolation to Entire Bug Population

To extrapolate to the entire population of MySQL server bugs under consideration, a simulation of 500,000 experiments was run using a population size of 30,728 bugs stratified as defined in Table 5.8. Within each strata, random Bernoulli trials were conducted with

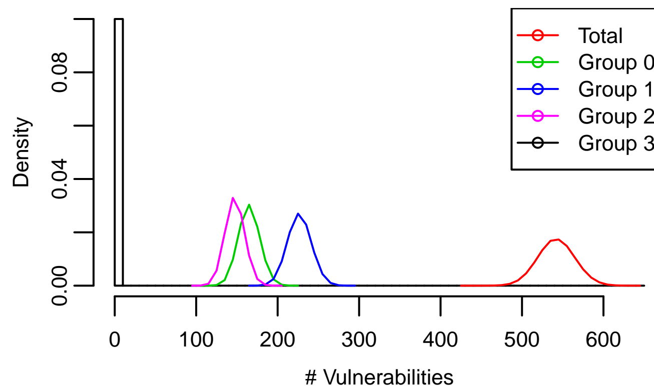


Figure 5.8. Simulated probability density functions

a probability of success equal to the probabilities in Table 5.9. The expected number of bugs that are also vulnerabilities ($E(y)$) can be calculated in a straightforward manner given $N = 30728$, $x_i = (1, 4, 5, 2)$, $n_i = (160, 46, 55, 10)$, and $N_i = (26476, 2614, 1628, 10)$ and Equation 5.1.

$$E(y) = N \sum_{i=0}^{3} \frac{N_i x_i}{N n_i} \approx 543 \tag{5.1}$$

However, the simulation allows for the combination of the the mixture of stratified binomial experiments into a single experiment and to obtain a 95% confidence interval on the number of remaining vulnerabilities. The simulation resulted in a mean number of vulnerabilities of 543 and a 95% confidence interval $(499, 587)$. Figure 5.8 shows the PDFs obtained from the simulation for each group and the aggregate.

5.5 Discussion

In this section, the newly identified MySQL vulnerabilities (in addition to those previously reported) are analyzed from two different perspectives: distribution across time and by CVSS score (Section 5.5.1). An estimate of the number of machines on the Internet which potentially could be affected by the new vulnerabilities we identified is presented (Section 5.5.2).

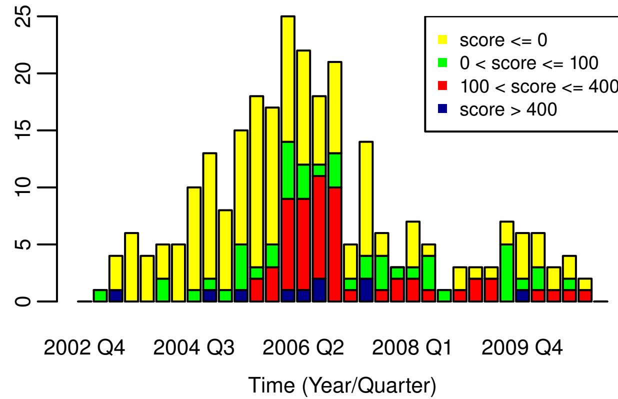


Figure 5.9. Time distribution of examined bugs

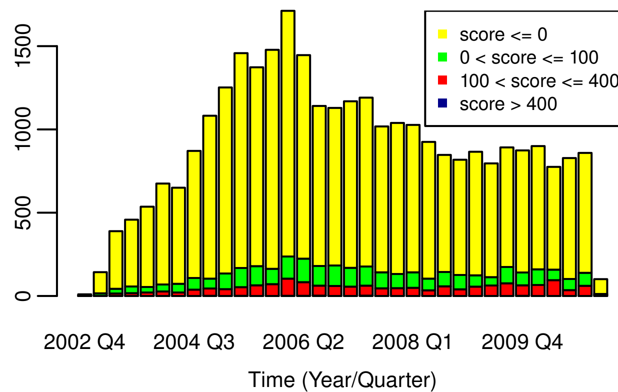


Figure 5.10. Time distribution of all server bugs

5.5.1 Data Analysis

To check the sampling mechanism, the time distribution of the sampled bugs is shown in Figure 5.9, and the distribution of all MySQL server bugs is shown in Figure 5.10. The two distributions share a similar shape until Q2 of 2006, after which the sampled bugs have a smaller relative population. This change in overall shape is due to the stratified bug sampling based on each bug scoring group.

CVSS (Common Vulnerability Scoring System) scores were applied to the identified vulnerabilities and their time distribution is shown in Figure 5.11. Most of the newly identified vulnerabilities were scored at 6.0 (medium severity). While scoring the new vulnerabilities, partial scores for the confidentiality, integrity, and availability dimen-

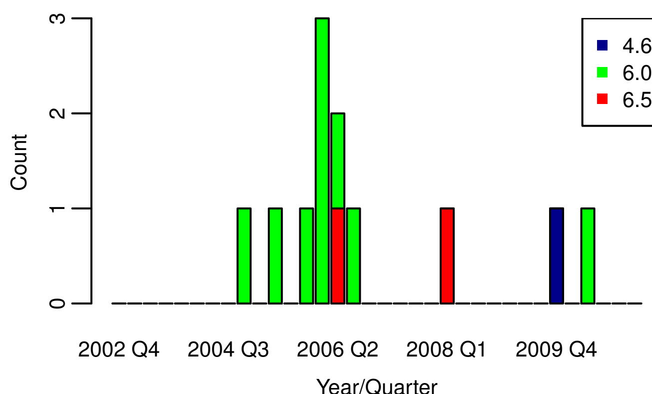


Figure 5.11. Time distribution of examined bugs and CVSS score

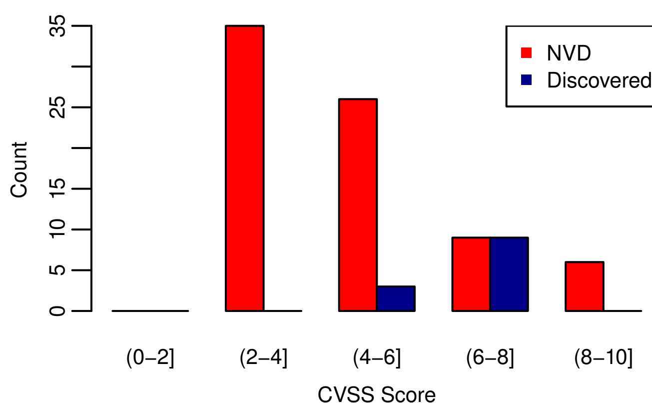


Figure 5.12. Known vulnerabilities and new vulnerabilities distribution by CVSS

sions (CIA dimensions) were given. This reflects a conservative approach and is justified because there was no "weaponize" the exploits for the vulnerabilities found.

Figure 5.12 shows the distribution of CVSS scores for known vulnerabilities from the NVD. The new vulnerabilities identified as a result of this study fall above the median CVSS score, yet below the maximum score of 10. If the CIA dimensions were ranked as 'complete' instead of 'partial' compromise, as would be expected them to be for weaponized exploits, then even the lowest CVSS score for these vulnerabilities would be 7.1 (this is despite requiring authenticated access and having medium access complexity).

5.5.2 Affected Machines

One might argue that the newly identified vulnerabilities primarily occur in older versions of the software and thus the number of affected machines is likely to be small. The assumption is that most MySQL server installations will have been upgraded to the newest version of the server software. To assess this argument, the Shodan computer search engine[4] which regularly scans the Internet collecting banner strings from applications was used. To ensure freshness of the results, the query was limited to machines discovered after Jan 1, 2012. Fortunately, MySQL reports its version string immediately after a client connects. The version string takes the following form: `5.0.16-standard-log` where 5.0.16 is the version number and the remaining portion are compiled in options.

In all, Shodan found 1.5 million hosts that report a MySQL version string. Combining that information with the versions affected by the newly identified vulnerabilities, an estimate of the number of affected machines identified by Shodan was possible. The results are shown in Table 5.11 for 2 of the 12 vulnerabilities identified as a result of this study.

These 2 vulnerabilities represent the range of values for the number of machines affected by each of the 12 vulnerabilities. Interestingly, it is against best practices to expose a MySQL server directly to the Internet; the database server provides the ability to restrict, by IP address, which machines can connect and any modern firewall (host based or hardware based) could be used to prevent similar connections. So it seems reasonable to assume that the 1.5 million hosts is a limited subset of machines actually running the MySQL server versions of interest in this study.

This analysis suggests that even vulnerabilities in older version of the MySQL server software may still be of potential impact to a large number of machines.

[4]SHODAN – Computer Search Engine, `http://www.shodanhq.com`

Table 5.11. Number of affected machines by vulnerability.

Vuln.	# affected	%
V1	17790	1.16%
V2	413065	26.92%
Total	1534188	

5.5.3 Study Limitations

The primary threat to validity comes in generalizing the experimental results from MySQL server software to a wider population of software products. This work examined the one software package and thus it is not known whether such high estimates for the number of misclassified bugs holds true for all software products. There is some evidence however to suggest that at least one other product has a similar incidence of misclassification. In [6], Arnold, et al. examined the Linux kernel for hidden impact bugs (bugs reported and fixed long before they were identified as vulnerabilities). They found a surprising number of bugs misclassified in this fashion. [52] found similar results for newer bugs reported in the Linux kernel and also for MySQL bugs. This later work also indicates that the incidence of these hidden impact bugs appear to be increasing with time. Consequently, there is at least some preliminary evidence that the problem of misclassified bugs may actually be getting worse, rather than better, with time.

To improve the misclassified bug estimates, increasing sample size and number of newly identified vulnerabilities would be beneficial. This could, in principle, be accomplished by increasing the number and diversity of expert vulnerability researchers devoted to the vulnerability identification effort. This study primary used one researcher extensively and a second one intermittently to identify and validate new vulnerabilities. Neither researcher was expert in database vulnerability discovery. Further, even these two experts were strongly constrained by time and availability. Consequently, there is some reason to suspect that the estimates made in this case study for the total number of misclassifed bugs have errored on the conservative side.

In addition, the vulnerability liklihood scoring process for bugs described in Sec-

tion 5.3.4 could likely be improved in several ways. Examination of Table 5.5 reveals a bias towards UNIX-like systems and this is due to the professional expertise of the authors: we simply know more about vulnerabilities affecting UNIX-like systems than, for example, the Microsoft Windows family of operating systems. Getting an accurate count of the number of bugs affecting UNIX-like operating systems versus other operating systems referenced in the MySQL bug database is complicated by the fact that the field capturing affected operating system is free-form and user-specified, so it is difficult to measure the effect of this sampling bias.

The text strings in Table 5.5 are used for static lexical matching and this has the possibility of lexically divergent but semantically equivalent phrases for the same idea. Because of its static nature, the bug scoring process is not resilient to the introduction of new phrases for the same idea and the retirement of old phrases (neologism). Note however, that attempts were made to choose phrases commonly used to describe vulnerability precursors over the time period of the study.

5.5.4 Related Work

The approach in this chapter is unique primarily because bug report databases were used to narrow the search for bugs not (yet) classified as vulnerabilities. Most previous work in the area of software vulnerability estimation has generally focused on the publicly reported vulnerabilities in software: known through online vulnerability databases such as the NVD or OSVDB.

For instance, Rescorla attempted unsuccessfully to fit the various statistical models (linear and Goel-Okumoto software reliability model) to vulnerability data from what is now the NVD [8]. He examined four operating systems: Windows NT, Solaris 2.5.1, RedHat Linux 7.0, and FreeBSD 4.0.

Likewise, using data including publicly reported vulnerabilities, Alhazmi, et al. proposed several statistical models for predicting vulnerability discovery: an S-shaped model, time-based logistical model, and an effort-based model [14, 59, 15]. Their primary work

focuses on application software, specifically Microsoft Internet Information Server (IIS) and the Apache HTTP server. In this one way, the work here is similar (this work also focuses on application software not on the underlying operating system).

Ozment and Schechter [7] examined the OpenBSD operating system and defined the term foundational vulnerabilities to describe those vulnerabilities present in the code base that did not change over the course of the study. They found that the rate of vulnerability reports in foundational code was decreasing over time (a potential contradiction with Resorla [8]).

In other work, Ozment compiled a database of vulnerabilities in OpenBSD from various sources (NVD, Bugtraq, OSVDB, etc.) and attempted to apply software reliability models to software security questions and had modest success in predictive accuracy [60]. However, his models were based on the rate of vulnerabilities identified and reported publicly. Given the predicted number of misclassified bugs presented in this chapter, it is not reasonable to assume that Ozment's models reasonably describe the actual number of vulnerabilities which have been discovered. Many may have been misclassified as not being vulnerabilities.

In [19], it is argued that vulnerabilities have different properties than software defects (at least during the first phase of a product's existence). However, other research suggests that many vulnerabilities do, in fact, show up as bug reports and are not classified as vulnerabilities until some time later [6, 52]. Further, in this chapter, it is demonstrated that a very significant portion of bug reports remain misclassified as not also being vulnerabilities. Thus it may be that the software properties of many vulnerabilities are, in general, similar to that of other software defects.

5.6 Conclusion

The estimation of the number of vulnerabilities in a software product is an important factor for assessing its quality. Many previous attempts at estimation have relied on publicly available vulnerability databases or software reliability models derived from these

databases. This work calls into question the quality of estimates based on this type of information.

We conducted an experiment to determine if the number of publicly identified vulnerabilities was a good estimate of the total number of discovered vulnerabilities (including those discovered bugs which have been misclassified as not being vulnerabilities). The experimental results provide some indication that even with the conservative approach outlined in this chapter, the quantity of discovered vulnerabilities in software is considerably higher than what previous work would estimate for the same code base. Consequently, previous counts and estimates of vulnerabilities in a software product may be seriously flawed, and may be inappropriate for use in risk comparisons between competing software products.

Additionally, the results presented in this chapter provide evidence that classifying bugs properly is a difficult task. Thus there is a need for both improved automated bug classifiers and a potential need for a deeper understanding of the software properties of vulnerabilities.

Future work will be to apply the approach described in this chapter to one or more additional code bases to determine if the results hold. Other reseach groups are encouraged to join in this endeavor. Further, works has begun to apply computational intelligence techniques to build a more robust bug classifier [52]. Eventually the hope is to identify a large number of misclassified bugs, determine reasons for the misclassifications, and to identify new software attributes which can be used for vulnerability identification. The ultimate goal is to improve the security, and estimates of security, for software products as early in their life cycles as possible.

Acknowledgment

The authors would like to thank John Matherly of Shodan for helping to gather the number of affected machines.

CHAPTER 6

Conclusion

This thesis has examined several under-studied aspects of the software security vulnerability ecosystem. Chapter 3 examined whether changing the grace period offered by vulnerability researchers to vendors of vulnerable products would influence the rate at which patches are produced. We found that less than 10% of bugs have patches in 90 days or less, so it is reasonable to wonder whether vendors can actually meet this deadline.

In Chapter 4, we defined two metrics median active vulnerabilities (MAV) and vulnerability free days (VFD) that can aid in the comparison between similar products coming from two different vendors and also between differing products within a single vendor. The metrics were studied using known data from various web browsers (across vendors) and other software products (within vendor).

The hidden impact of vulnerabilities was examined in Chapter 5. This chapter calls into question statistics based on known vulnerabilities. The bug database for the MySQL database software was examined and a significant number of previously unknown vulnerabilities were identified. Extrapolating from the proportions found, the expected number of yet to be identified vulnerabilities dwarfs the number of known vulnerabilities. Given the large difference between the two, it is hard to say that the known vulnerabilities is representative for MySQL. The study only focused on one product, and future work should attempt to duplicate the experiment on one or more products.

Software will continue to have vulnerabilities as long as it is designed and written by humans. Systems are too complex to be modeled completely enough to ensure no violations of the [explicit or implicit] security model are possible. There are possibly metrics to compare given products, but research based on known vulnerabilities are likely to be on a weak foundation.

REFERENCES

[1] D. P. Fidler, "Was stuxnet an act of war? decoding a cyberattack," *Security and Privacy*, vol. 9, no. 4, pp. 56–59, July–August 2011.

[2] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet dossier," Symantec Security Response, Tech. Rep., February 2011, version 1.4.

[3] S. Frei, D. Schatzmann, B. Plattner, and B. Trammel, "Modeling the security ecosystem - the dynamics of (in)security," in *Workshop on the Economics of Information Security WEIS*, R. Anderson, Ed., Cambridge, UK, June 2009, Accessed: January 21, 2013. [Online]. Available: http://www.techzoom.net/publications/security-ecosystem/

[4] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, Purdue, May 1998, Accessed: January 21, 2013. [Online]. Available: http://www.krsul.org/ivan/articles/main.pdf

[5] A. Ozment, "Vulnerability discovery and software security," Ph.D. dissertation, University of Cambridge Computer Laboratory, 2007.

[6] J. Arnold, T. Abbott, N. Elhage, G. Thomas, and A. Kaseorg, "Security impact ratings considered harmful," in *12th workshop on Hot Topics in Operating Systems*. USENIX, May 2009.

[7] A. Ozment and S. E. Schechter, "Milk or wine: Does software security improve with age?" in *15th USENIX Security Symposium*. USENIX, July 2006, pp. 93–104.

[8] E. Rescorla, "Is finding security holes a good idea?" *IEEE Security and Privacy*, vol. 3, no. 1, pp. 14–19, January 2005.

[9] Microsoft. "Evolution of the Microsoft SDL." Accessed: January 21, 2013. [Online]. Available: http://www.microsoft.com/security/sdl/learn/evolution.aspx

[10] R. Telang and S. Wattal, "An empirical analysis of the impact of software vulnerability announcements on firm stock price," *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 544–557, June 2007.

[11] A. Arora, R. Telang, and H. Xu, "Optimal policy for software vulnerability disclosure," *Management Science*, vol. 54, no. 4, pp. 642–656, April 2008.

[12] A. Arora, A. Nandkumar, and R. Telang, "Does information security attack frequency increase with vulnerability disclosure? an empirical analysis," *Information Systems Frontiers*, vol. 8, no. 5, pp. 350–362, December 2006.

[13] A. Arora, R. Krishnan, R. Telang, and Y. Yang, "An empirical analysis of software vendors patching behavior: Impact of vulnerability disclosure," *Information Systems Research*, vol. 21, no. 1, pp. 115–132, March 2010.

[14] O. H. Alhazmi and Y. K. Malaiya, "Modeling the vulnerability discovery process," in *International Symposium on Software Reliability Engineering.* IEEE, December 2005.

[15] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers Security*, vol. 26, no. 3, pp. 219–228, 2007.

[16] A. Ozment, "Improving vulnerability discovery models: Problems with definitions and assumptions," in *3rd Workshop on Quality of Protection (QoP).* ACM, October 2007.

[17] W. Kandek, "The laws of vulnerabilities 2.0," in *BlackHat*, Las Vegas, NV, USA, July 2009, Accessed: January 21, 2013. [Online]. Available: https://www.qualys.com/research/vulnlaws/

[18] M. Acer and C. Jackson, "Critical vulnerability in browser security metrics," in *Web 2.0 Security and Privacy*, ser. IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 2010.

[19] S. Clark, S. Frei, M. Blaze, and J. M. Smith, "Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities," in *Annual Computer Security Applications Conference ACSAC*, December 2010, pp. 251–260.

[20] G. Schryen, "Is open source security a myth?" *Communications of the ACM*, vol. 54, no. 5, pp. 130–140, May 2011.

[21] S. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," in *International Conference on Database and Expert Systems*, August 2011.

[22] S. Zhang, X. Ou, A. Singhal, and J. Homer, "An empirical study of a vulnerability metric aggregation method," in *International Conference on Security and Management*, Las Vegas, NV, USA, July 2011.

[23] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proc. of the 7th IEEE Working Conf. on Mining Software Repositories (MSR)*, May 2010, pp. 1–10.

[24] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Proc. of the 15th European Conf. on Software Maintenance and Reengineering (CSMR)*, March 2011, pp. 249–258.

[25] D. Cubranic and G. C. Murphy, “Automatic bug triage using text categorization,” in *Proc. of the 16th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE).* KSI Press, June 2004, pp. 92–97.

[26] A. Austin and L. Williams, “One technique is not enough: A comparison of vulnerability discovery techniques,” in *Proc. of the 2011 Int. Symp. on Empirical Software Engineering and Measurement (ESEM)*, September 2011, pp. 97–106.

[27] W. M. Khoo, S. Aloteibi, R. Anderson, and M. Meeks, “Hunting for vulnerabilities in large software: the OpenOffice suite,” Cambridge University, Tech. Rep., June 2010.

[28] P. Li and B. Cui, “A comparative study on software vulnerability static analysis techniques and tools,” in *Proc. of the IEEE Int. Conf. on Information Theory and Information Security (ICITIS)*, December 2010, pp. 521–524.

[29] F. Yamaguchi, F. F. Lindner, and K. Rieck, “Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning,” in *Proc. of the 5th USENIX Workshop on Offensive Technologies (WOOT).* USENIX, August 2011.

[30] D. Kester, M. Mwebesa, and J. S. Bradbury, “How good is static analysis at finding concurrency bugs?” in *Proc. of the 10th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, September 2010, pp. 115–124.

[31] L. Torri, G. Fachini, L. Steinfeld, V. Camara, L. Carro, and É. Cota, “An evaluation of free/open source static analysis tools applied to embedded software,” in *Proc. of the 11th Latin American Test Workshop (LATW)*, March 2010, pp. 1–6.

[32] M. Zitser, R. Lippmann, and T. Leek, “Testing static analysis tools using exploitable buffer overflows from open source code,” in *Proc. of the 12th Int. Symp. on Foundations of Software Engineering (FSE).* ACM SIGSOFT, November 2004, pp. 97–106.

[33] M. A. McQueen, J. L. Wright, and L. Wellman, “Are vulnerability disclosure deadlines justified?” in *Security Measurements and Metrics (METRISEC)*, Banff, Alberta, Canada, September 2011.

[34] CERT Coordination Center. “CERT/CC vulnerability disclosure policy.” Accessed: January 21, 2013. May 2008. [Online]. Available: http://www.cert.org/kb/vul_disclosure.html

[35] J. E. Dunn. IDG News. “’Serious’ Microsoft Office encryption flaw discovered.” Accessed: January 21, 2013. January 2005. [Online]. Available: http://www.pcworld.com/article/119483/article.html

[36] Rapid7. “Vulnerability disclosure policy.” Accessed: January 21, 2013. June 2010. [Online]. Available: http://www.rapid7.com/disclosure.jsp

[37] C. Evans, E. Grosse, N. Mehta, M. Moore, T. Ormandy, J. Tinnes, and M. Zalewski. Google Security Team. “Rebooting responsible disclosure: a focus on protecting end users.” Google Online Security Blog. Accessed: January 21, 2013. July 2010. [Online]. Available: http://googleonlinesecurity.blogspot.com/2010/07/rebooting-responsible-disclosure-focus.html

[38] A. Portnoy. Zero Day Initiative. “ZDI disclosure process changes.” Accessed: January 21, 2013. Aug 2010. [Online]. Available: http://dvlabs.tippingpoint.com/blog/2010/08/03/zdi-disclosure-changes

[39] K. McLaughlin. CRN Technology News. “HP’s zero day initiative gives vendors patching deadline.” Accessed: January 21, 2013. August 2010. [Online]. Available: http://www.crn.com/news/security/226500302/hps-zero-day-initiative-gives-vendors-patching-deadline.htm

[40] S. Ragan. The Tech Herald. “The new era of vulnerability disclosure – a brief chat with HD Moore.” Accessed: January 21, 2013. August 2010. [Online]. Available: http://www.thetechherald.com/articles/The-new-era-of-vulnerability-disclosure-a-brief-chat-with-HD-Moore/11047/

[41] D. Brumley, P. Poosankam, D. Song, and J. Zheng, “Automatic patch-based exploit generation is possible: Techniques and implications,” in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2008, pp. 143–157.

[42] D. Veditz. “Bug 484320 - (CVE-2009-1044) XUL <tree> _moveToEdgeShift garbage-collection exploit (zdi-can-465).” Accessed: January 21, 2013. March 2009. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=484320

[43] J. L. Wright, M. McQueen, and L. Wellman, “Analyses of two end-user software vulnerability exposure metrics,” in *7th International Conference on Availability, Reliability, and Security (ARES)*. Prague, Czech Republic: IEEE, August 2012.

[44] J. L. Wright, M. A. McQueen, and L. Wellman, “Analyses of two end-user vulnerability metrics (extended version),” *Information Security Technical Report*, vol. 17, no. 4, pp. 173–184, May 2013, Elsevier.

[45] DHS National Cyber Security Division. US-CERT. “National Vulnerability Database Home.” Accessed: January 21, 2013. [Online]. Available: http://nvd.nist.gov/

[46] iDefense Vulnerability Contributor Program. “iDefense cyber intelligence, threat intelligence and security – verisign.” Accessed: January 21, 2013. [Online]. Available: http://www.verisigninc.com/en_US/products-and-services/network-intelligence-availability/idefense/public-vulnerability-reports/index.xhtml

[47] Zero Day Initiative. TippingPoint. “TippingPoint Zero Day Initiative.” Accessed: January 21, 2013. [Online]. Available: http://www.zerodayinitiative.com

[48] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *SIGCOMM Workshop on Large-Scale Attack Defense (LSAD)*. ACM, September 2006, pp. 131–138.

[49] Cenzic, Inc. "Cenzic web application security trends report q3/q4 2010." Accessed: January 25, 2012. 2010. [Online]. Available: http://www.cenzic.com/resources/reg-not-required/trends/

[50] IBM X-Force. "2010 trend and risk report." Accessed: January 21, 2013. February 2010. [Online]. Available: http://www-935.ibm.com/services/us/iss/xforce/trendreports/

[51] J. L. Wright, J. W. Larsen, and M. A. McQueen, "Estimating software vulnerabilities: A case study based on the misclassification of bugs in MySQL server," in *International Conference on Availability, Reliability, and Security (ARES)*. Regensburg, Germany: IEEE, September 2013, pp. 72–81.

[52] D. Wijayasekara, M. Manic, J. L. Wright, and M. A. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *International Conference on Human System Interactions (HSI)*, Perth, Austrailia, June 2012.

[53] MITRE Corporation. "Common Vulnerabilities and Exposures (CVE)." Accessed: January 21, 2013. November 2011. [Online]. Available: http://cve.mitre.org

[54] H. Shahriar and M. Zulkernine, "Classification of static analysis-based buffer overflow detectors," in *Proc. of the 4th Int. Conf. on Secure Software Integration and Reliability Improvement Companion*, June 2010, pp. 94–101.

[55] S. M. Kerner. Database Journal. "Oracle commits to MySQL with InnoDB." Accessed: January 21, 2013. April 2010. [Online]. Available: http://www.databasejournal.com/features/mysql/article.php/3876206/Oracle-Commits-to-MySQL-with-InnoDB.htm

[56] Microsoft. "Microsoft exploitability index." Accessed: January 21, 2013. October 2008. [Online]. Available: http://technet.microsoft.com/en-us/security/cc998259

[57] K. Johnson and M. Miller, "Exploiting the otherwise non-exploitable on Windows," Leviathan Security Group, Tech. Rep., May 2006, Accessed: January 21, 2013. [Online]. Available: http://uninformed.org/?v=4&a=5&t=sumry

[58] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap taichi: Exploiting memory allocation granularity in heap-spraying attacks," in *Annual Computer Security Applications Conference (ASSAC '10)*. ACM, 2010, pp. 327–336.

[59] O. H. Alhazmi and Y. K. Malaiya, "Quantitative vulnerability assessment of systems software," in *Reliability and Maintainability Symposium*, January 2005, pp. 615–620.

[60] A. Ozment, "Software security growth modeling: Examining vulnerabilities with reliability growth models," in *Quality of Protection: Security Measurements and Metrics*, ser. Advances in Information Security, D. Gollmann, F. Massacci, and A. Yautsiukhin, Eds. Springer, 2006, vol. 23.

APPENDIX A

Identifiers of Sampled Bugs for Each Scoring Group

APPENDIX A

Identifiers of Sampled Bugs for Each Scoring Group

For each of the four MySQL server bug scoring groups, we sampled a subset of the bug reports to undergo detailed analysis in order to determine if they were also vulnerabilities. In this Appendix we provide, by scoring group, the bug identifiers of each sampled bug (Tables A.1 through A.4). The bug identifier is used by the MySQL software team to uniquely identify each bug.

We encourage other research teams to carefully evaluate each of these bug samples and validate, or invalidate, our results. We further suggest that other research teams perform the same scoring process we developed, do their own sampling, followed by vulnerability identification. We will of course be interested in the results of these other efforts, and would hope to find mechanisms for collaboration and sharing of intermediate data in addition to results.

Table A.1. Group 0 (score ≤ 0) bugs examined.

bugid	score	bugid	score	bugid	score	bugid	score	bugid	score
249	0	429	0	708	0	785	0	856	0
952	0	1010	0	1073	0	1076	0	1484	0
1879	0	1977	0	1993	0	2539	0	3089	0
3218	0	3446	0	3667	0	3852	0	3863	0
3925	0	4451	-20	4609	0	4752	0	4801	0
5059	0	5145	0	5248	0	5712	0	5811	0
5926	0	5985	0	6557	0	6600	0	6690	0
6932	0	7047	0	7170	-20	7223	0	7329	0
7443	0	7687	0	7873	0	7957	0	7996	0
8134	0	8357	0	8435	0	9650	0	9965	0
10083	0	10313	0	10537	0	10818	-50	10935	0
11017	0	11072	0	11549	0	11731	0	11774	0
11953	0	11997	0	12457	0	12501	0	12657	0
12702	0	12702	0	12778	0	12902	0	13059	0
13202	0	13370	-50	13499	0	13642	0	13845	0
14216	0	14543	0	14900	0	14906	-50	14953	-50
15068	0	15072	-50	15471	0	15599	0	15716	0
15745	0	16172	0	16179	0	16205	0	16215	0
16331	0	17020	0	17319	0	17894	0	18276	-20
18289	0	18347	0	18729	0	18859	0	19326	0
19421	0	19747	0	19861	0	19893	0	20074	0
20140	0	20214	0	20381	-50	21020	0	21088	0
21332	0	21487	0	21566	0	22457	0	23208	0
24002	0	24062	0	24302	0	24429	0	24532	-50
24663	0	24988	0	26287	0	26602	0	26613	0
27600	0	28166	0	28194	0	28571	0	28578	0
28788	0	28872	0	28940	0	29033	-50	29214	0
29579	0	29822	0	33613	0	34454	0	35146	0
35343	0	37133	0	40263	0	40737	0	41854	0
44923	-50	47318	0	47582	0	47902	0	48661	0
48691	0	49450	0	50412	-70	51468	0	52207	0
54645	0	54699	0	55686	0	56708	0	57564	0

Table A.2. Group 1 ($0 < \text{score} \leq 100$) bugs examined.

bugid	score	bugid	score	bugid	score	bugid	score	bugid	score
146	100	2490	100	2674	40	5435	70	6885	20
7981	100	9916	20	9949	100	10058	20	11226	20
13626	100	13673	100	15338	50	16470	20	16738	20
16739	20	17123	20	18037	100	19311	100	19626	50
20076	50	20823	100	23165	100	24482	100	25332	20
25908	100	28073	50	28421	100	29801	100	31214	50
31242	100	31916	20	34534	20	36156	20	36583	100
37408	20	38848	100	46321	20	46592	50	46782	20
47144	100	47280	20	47883	100	50457	20	50632	100
56281	100								

Table A.3. Group 2 ($0 < \text{score} \leq 400$) bugs examined.

bugid	score	bugid	score	bugid	score	bugid	score	bugid	score
13090	200	13627	200	14086	270	15268	200	15924	200
16218	270	16298	180	16805	270	17001	270	17419	270
17535	270	17672	200	18649	180	19155	200	19210	180
19382	180	19727	270	20151	200	20258	270	20595	200
20979	200	21135	270	21651	170	21709	170	21851	200
21927	170	22244	170	22413	170	22440	170	22879	170
23075	170	23368	170	23506	170	23542	170	23944	170
24199	170	24211	170	24480	170	24502	170	27296	180
30562	180	31569	240	32431	240	33844	180	35272	180
36656	180	41733	240	42438	240	43827	320	44040	320
44886	320	51136	190	52711	190	55627	190	59111	190

Table A.4. Group 3 ($\text{score} > 400$) bugs examined.

bugid	score	bugid	score	bugid	score	bugid	score	bugid	score
639	470	7289	500	10918	420	16550	500	19885	420
21658	420	21913	440	27752	470	28975	520	48319	420

APPENDIX B

Published Works

APPENDIX B

Published Works

What follows is a list of works published by this author starting from the most recent and going back to the first. The focus is on academically published work.

Wright, J. L., Larsen, J. W., and McQueen, M. A., "Estimating software vulnerabilities: A case study based on the misclassification of bugs in MySQL server," in *International Conference on Availability, Reliability, and Security (ARES)*. Regensburg, Germany: IEEE, September 2013, pp. 72–81.

> Software vulnerabilities are an important part of the modern software economy. Being able to accurately classify software defects as a vulnerability, or not, allows developers and end users to expend appropriately more effort on fixing those defects which have security implications. However, we demonstrate in this paper that the expected number of misclassified bugs (those not marked as also being vulnerabilities) may be quite high and thus human efforts to classify bug reports as vulnerabilities appears to be quite ineffective.

Wright, J. L., McQueen, M., and Wellman, L., "Analyses of two end-user software vulnerability exposure metrics," in *7th International Conference on Availability, Reliability, and Security (ARES)*. Prague, Czech Republic: IEEE, August 2012.

> Two metrics focused on end-user consumption are defined: Median Active Vulnerabilities (MAV) and Vulnerability Free Days (VFD). The calculation, use, and implications are explored in a case study involving the four most popular web browsers (Apple Safari, Google Chrome, Microsoft Internet Explorer, and Mozilla Firefox) and two non-browser products (Apple QuickTime and Microsoft Office).

Wright, J. L., McQueen, M. A., and Wellman, L., "Analyses of two end-user vulnerability metrics (extended version)," *Information Security Technical Report*, vol. 17, no. 4, pp. 173–184, May 2013, Elsevier.

> This paper expands on the paper above and adds various analyses. Of particular note is the demonstration that the severity of vulnerabilities as measured by CVSS score does not correlate to the lifetime (time between vendor notification and release of patch). Further, it is demonstrated that for web browsers, vulnerabilities bought by iDefense/ZDI are more severe than vulnerabilities found only in the National Vulnerability Database

Wijayasekara, D., Manic, M., Wright, J. L., and McQueen, M. A., "Mining bug databases for unidentified software vulnerabilities," in *International Conference on Human System Interactions (HSI)*, Perth, Austrailia, June 2012.

The importance of hidden impact bugs, bugs which are reported and only later discovered to be vulnerabilities, is explored in this paper. This work expands upon previous work and demonstrates that hidden impact bugs exist within both the Linux kernel and the MySQL database server.

McQueen, M. A., Wright, J. L., and Wellman, L., "Are vulnerability disclosure deadlines justified?" in *Security Measurements and Metrics (METRISEC)*, Banff, Alberta, Canada, September 2011.

Vulnerability research organizations Rapid7, Google Security team, and Zero Day Initiative imposed grace periods for public disclosure of vulnerabilities. The grace periods ranged from 45 to 182 days, after which disclosure might occur with or without an effective mitigation from the affected software vendor. At this time there is indirect evidence that the shorter grace periods of 45 and 60 days may not be practical. However, there is strong evidence that the recently announced Zero Day Initiative grace period of 182 days yields benefit in speeding up the patch creation process, and may be practical for many software products.

McJunkin, T., Boring, R., McQueen, M. A., Shunn, L., Wright, J. L., Gertman, D., Linda, O., McCarty, K., and Manic, M., "Concept of operations for data fusion visualization," in *Advances in Safety, Reliability and Risk Management*, ser. ESREL 2011, Berenguer, C., Grall, A., and Soares, C. G., Eds. CRC Press, September 2011, pp. 634–640.

Data fusion for process control involves the presentation of synthesized sensor data in a manner that highlights the most important system states to an operator. The design of a data fusion interface must strike a balance between providing a process overview to the operator while still helping the operator pinpoint anomalies as needed. With the inclusion of a predictor system in the process control interface, additional design requirements must be considered, including the need to convey uncertainty regarding the prediction and to minimize nuisance alarms. This paper reviews these issues and establishes a design process for data fusion interfaces centered on creating a concept of operations as the basis for a design style guide.

Linda, O., Manic, M., Vollmer, T., and Wright, J. L., "Fuzzy logic based anomaly detection for embedded network security cyber sensor," in *IEEE Symposium Series on Computational Intelligence.* Paris, France: IEEE, April 2011, pp. 202–209.

Resiliency and security in critical infrastructure control systems in the modern world of cyber terrorism constitute a relevant concern. Developing a network security system specifically tailored to the requirements of such critical assets is of a primary importance. This paper proposes a novel learning algorithm for anomaly based network security cyber sensor together with its hardware implementation. The presented learning algorithm constructs a fuzzy logic

rule base modeling the normal network behavior. Individual fuzzy rules are extracted directly from the stream of incoming packets using an online clustering algorithm. This learning algorithm was specifically developed to comply with the constrained computational requirements of low-cost embedded network security cyber sensors. The performance of the system was evaluated on a set of network data recorded from an experimental test-bed mimicking the environment of a critical infrastructure control system.

Wright, J. L. and Manic, M., "Neural network architecture selection analysis with application to cryptography location," in *International Joint Conference on Neural Networks (IJCNN).* Barcelona, Spain: IEEE, July 2010.

When training a neural network it is tempting to experiment with architectures until a low total error is achieved. The danger in doing so is the creation of a network that loses generality by over-learning the training data; lower total error does not necessarily translate into a low total error in validation. The resulting network may keenly detect the samples used to train it, without being able to detect subtle variations in new data. In this paper, a method is presented for choosing the best neural network architecture for a given data set based on observation of its accuracy, precision, and mean square error. The method relies on k-fold cross validation to evaluate each network architecture k times to improve the reliability of the choice of the optimal architecture. The need for four separate divisions of the data set is demonstrated (testing, training, and validation, as normal, and an comparison set). Instead of measuring simply the total error the resulting discrete measures of accuracy, precision, false positive, and false negative are used. This method is then applied to the problem of locating cryptographic algorithms in compiled object code for two different CPU architectures to demonstrate the suitability of the method.

Wright, J. L. and Manic, M., "The analysis of dimensionality reduction techniques in cryptographic object code classification," in *Conference on Human Systems Interactions (HSI).* Rzeszow, Poland: IEEE, May 2010, pp. 157–162.

This paper compares the application of three different dimension reduction techniques to the problem of classifying functions in object code form as being cryptographic in nature or not. A simple classier is used to compare dimensionality reduction via sorted covariance, principal component analysis, and correlation-based feature subset selection. The analysis concentrates on the classification accuracy as the number of dimensions is increased. It is demonstrated that when discarding 90% of the measured dimensions, accuracy only suffers by 1% for this problem. By discarding dimensions, computational intelligence techniques can be applied with a drastic reduction in algorithmic complexity. The primary focus is on Intel IA32 instruction set, but analysis shows consistent results on the Sun SPARC instruction set.

Wright, J. L. and Manic, M., "Neural network approach to locating cryptography in object code," in *Internation Conference on Emerging Technologies and Factory Automation (ETFA).* Palma de Mallorca, Spain: IEEE, September 2009, pp. 1–4.

Finding and identifying cryptography is a growing concern in the malware analysis community. In this paper, artificial neural networks are used to classify functional blocks from a disassembled program as being either cryptography related or not. The resulting system, referred to as NNLC (Neural Net for Locating Cryptography) is presented and results of applying this system to various libraries are described.

Wright, J. L. and Manic, M., "Time synchonization in heirarchical TESLA wireless sensor networks," in *Internation Symposium on Resilient Control Systems (ISRCS).* Idaho Falls, ID, USA: IEEE, 2009, pp. 36–39.

Time synchronization and event time correlation are important in wireless sensor networks. In particular, time is used to create a sequence events or time line to answer questions of cause and effect. Time is also used as a basis for determining the freshness of received packets and the validity of cryptographic certificates. This paper presents secure method of time synchronization and event time correlation for TESLA-based hierarchical wireless sensor networks. The method demonstrates that events in a TESLA network can be accurately timestamped by adding only a few pieces of data to the existing protocol.

Keromytis, A. D., de Raadt, T., Wright, J. L., and Burnside, M., "Cryptography as an operating system service: A case study," *Transactions on Computer Systems (ToCS)*, vol. 24, no. 1, pp. 1–38, February 2006, ACM.

This is a revised and extended version of "The Design of the OpenBSD cryptographic framework" conference paper (below). It adds several experiments and analyses omitted from conference version.

Smith, J. M., Greenwald, M. B., Ioannidis, S., Keromytis, A. D., Laurie, B., Maughan, D., Rahn, D., and Wright, J. L., "Experiences enhancing open source security in the POSSE project," in *Global Information Technologies: Concepts, Methodologies, Tools, and Applications*, Tan, F. B., Ed. Idea Group Publishing, 2007, pp. 1587–1598.

This is a reprint of the article below.

Smith, J. M., Greenwald, M. B., Ioannidis, S., Keromytis, A. D., Laurie, B., Maughan, D., Rahn, D., and Wright, J. L., "Experiences enhancing open source security in the POSSE project," in *Free/Open Source Development*, Koch, S., Ed. Idea Group Publishing, 2004, pp. 242–257.

This chapter reports on our experiences with POSSE, a project studying "Portable Open Source Security Elements" as part of the larger DARPA effort on Composable High Assurance Trusted Systems. We describe the organization created to manage POSSE and the significant acceleration in producing widely used secure software that has resulted. POSSEs two main goals were, first, to increase security in open source systems and, second, to more broadly disseminate security knowledge, "best practices," and working code that reflects these practices. POSSE achieved these goals through careful study of systems ("audit") and starting from a well-positioned technology base (OpenBSD). We hope to illustrate the advantages of applying OpenBSD-style methodology to secure, open-source projects, and the pitfalls of melding multiple open-source efforts in a single project.

Keromytis, A. D., Wright, J. L., and de Raadt, T., "The design of the OpenBSD cryptographic framework," in *USENIX Annual Technical Conference*. San Antonio, TX, USA: USENIX, June 2003, pp. 181–196.

Cryptographic transformations are a fundamental building block in many security applications and protocols. To improve performance, several vendors market hardware accelerator cards. However, until now no operating system provided a mechanism that allowed both uniform and efficient use of this new type of resource. We present the OpenBSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the kernel, that provides uniform access to accelerator functionality by hiding card-specific details behind a carefully-designed API. We evaluate the impact of the OCF in a variety of benchmarks, measuring overall system performance, application throughput and latency, and aggregate throughput when multiple applications make use of it.

Keromytis, A. D. and Wright, J. L., "Transparent network security policy enforcement," in *USENIX Annual Technical Conference, Freenix track*. San Diego, CA, USA: USENIX, June 2000, pp. 215–226.

Recent work in the area of network security, such as IPsec, provides mechanisms for securing the traffic between any two interconnected hosts. However, it is not always possible, economical, or even practi- cal from an administration and operational point of view to upgrade the software and configuration of all the nodes in a network to support such security protocols. This paper describes the architecture and implementation of a Layer-2 (link layer) bridge with extensions for offering Layer-3 security services. We extend the OpenBSD ethernet bridge to perform simple IP packet filtering and IPsec processing for incoming and outgoing packets on behalf of a protected node, completely transparently to both the protected and the remote communication endpoint. The same mechanism may be used to construct "virtual local area networks," by establishing IPsec tunnels between OpenBSD bridges connected geographically separated LANs. As our system operates in the link

layer, there is no need for software or configuration changes in the protected nodes.

APPENDIX C

Copyright Information

APPENDIX C

Copyright Information

Most of the papers used in the creation of this thesis are copyright © IEEE. The use of papers copyright information for each paper is included on the following pages.

Title: Are Vulnerability Disclosure Deadlines Justified?
Conference Proceedings: 2011 Third International Workshop on Security Measurements and Metrics (Metrisec)
Author: McQueen, M.; Wright, J.L.; Wellman, L.
Publisher: IEEE
Date: 21-21 Sept. 2011
Copyright © 2011, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2. In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website:

> In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2013 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement.

Title: Analyses of Two End-User Software Vulnerability Exposure Metrics
Conference Proceedings: 2012 Seventh International Conference on Availability, Reliability and Security (ARES)
Author: Wright, J.L.; McQueen, M.; Wellman, L.
Publisher: IEEE
Date: 20-24 Aug. 2012
Copyright © 2012, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2. In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website:

> In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2013 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement.

Title: Analyses of Two End-User Software Vulnerability Exposure Metrics (extended version)
Information Security Technical Report, May 2013, Volume 17, Number 4.
Author: Wright, J.L.; McQueen, M.; Wellman, L.
Publisher: Elsevier
Date: May 2013
Copyright © 2013, Elsevier Ltd.

Thesis / Dissertation Reuse

According to Elsevier's Author Rights and Responsibilities[1], Elsevier does not require individuals working on a thesis to obtain a formal reuse license as long as full acknowledgement (above, and also in each relevant chapter) of the final version is made.

[1] http://www.elsevier.com/journal-authors/author-rights-and-responsibilities#rights, Last Accessed: July 14, 2013.

Title: Mining Bug Databases for Unidentified Software Vulnerabilities
Conference Proceedings: 2012 5th International Conference on Human System Interactions (HSI)
Author: Wijayasekara, Dumidu; Manic, Milos; Wright, Jason L.; McQueen, Miles
Publisher: IEEE
Date: 6-8 June 2012
Copyright © 2012, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.

2. In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.

3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior authors approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]

2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.

3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website:

> In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2013 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement.

Title: Estimating Software Vulnerabilities: A Case Study Based on the Misclassification of Bugs in MySQL Server
Conference Proceedings: 2013 Eighth International Conference on Availability, Reliability and Security (ARES)
Author: Wright, J.L.; Larsen, J.W.; McQueen, M.
Publisher: IEEE
Date: 2-6 Sep. 2013
Copyright © 2013, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.

2. In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.

3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]

2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.

3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website:

> In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/`

publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2013 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement.