Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

The Evolution and Decay of Statically Detected Source Code Vulnerabilities

Massimiliano Di Penta, Luigi Cerulo, Lerina Aversano
RCOST – Dept. of Engineering, University of Sannio
Via Traiano, 82100 Benevento, Italy
dipenta@unisannio.it, lcerulo@unisannio.it, aversano@unisannio,it

Abstract

The presence of vulnerable statements in the source code is a crucial problem for maintainers: properly monitoring and, if necessary, removing them is highly desirable to ensure high security and reliability. To this aim, a number of static analysis tools have been developed to detect the presence of instructions that can be subject to vulnerability attacks, ranging from buffer overflow exploitations to command injection and cross-site scripting.

Based on the availability of existing tools and of data extracted from software repositories, this paper reports an empirical study on the evolution of vulnerable statements detected in three software systems with different static analysis tools. Specifically, the study investigates on vulnerability evolution trends and on the decay time exhibited by different kinds of vulnerabilities.

Keywords: software vulnerabilities, mining software repositories, empirical study.

1 Introduction

The presence of vulnerable instructions in the source code is often the cause of security attacks or, in other cases, of system failures or crashes. Example of vulnerable instructions are buffer accesses performed without a proper boundary checking, or the injection of malicious shell commands and SQL instructions through Web forms. In his PhD thesis [15] Krsul defined a *software vulnerability* as *"an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy"*.

Detecting the presence of such instructions is therefore crucial to ensure high security and reliability: for example, as claimed by CERT[1], statements vulnerable to buffer overflows are cause of 50% of software attacks. A number of automatic tools have been developed for the identification of such a kind of instructions. These tools performs static analysis of source code in different ways: some tools merely use pattern matching e.g., with the aim of identifying programming language functions that are known to be vulnerable, while others perform a more accurate analysis, including data-flow analysis. An overview of these tools is reported in Section 3. Although the effectiveness of these tools has been assessed in the past, up to now the literature lacks of studies aimed at analyzing how vulnerable statements are maintained, or whether during the time developers modify the system to protect it against vulnerability attacks.

Nowadays, the availability of several vulnerability detection tools, of data from source code and defect repositories for many open source systems, and techniques to perform software historical analyses [10], poses the basis for this kind of study. This paper performs a fine-grained analysis of the evolution of vulnerabilities statically detected in three network applications, namely a Web caching proxy (Squid), a file/printer sharing system (Samba), and a Web application framework (Horde), the first two developed in Java and the third one in PHP. The objective of this study is two-fold:

- analyze how the number of vulnerabilities in a system varies over the time, and check for the presence of trends and on vulnerability removal activities occurred in particular contexts, e.g., in proximity of a new release;
- focus on vulnerabilities that, after a given time interval, disappear from the system, compare their decay time for different vulnerability categories and investigate whether such a decay time follows any particular statistical distribution.

The paper is organized as follows. After a discussion of related work, Section 3 discusses the main features of static vulnerability detection tools, and presents a taxonomy of vulnerabilities that can be detected by the tools we considered. Section 4 details the vulnerability evolution analysis process. Section 5 describes the empirical study definition, the context, and outlines the research questions it aims to

[1] *http://www.cert.org/*

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.20

address. Results are reported and discussed in Section 6, while threats to validity discussed in Section 7. Finally, Section 8 concludes the paper and outlines directions for future work.

2 Related Work

The literature reports various approaches for identifying and protecting vulnerabilities. DaCosta *et al.* [5] proposed an approach to evaluate the security vulnerability likelihood of a function based on the assumption that a function near a source of input may have a high probability to being vulnerable. Buffer overflow detection has been addressed in a number of research works (e.g., [14, 20]). Genetic algorithms and simulated annealing have been used with the purpose of exercising exception handling and determining the cause-effect relationship between program inputs and variables used in likely dangerous statements [7, 22]. Program transformation languages, such as TXL, have been used by Dahn and Mancoridis [6] and by Wang *et al.* [23] to secure C programs against buffer overflows.

SQL-injection exploits weak validation of textual inputs used to build database queries. Merlo *et al.* [17] combined static analysis, dynamic analysis, and code reengineering to automatically protect applications written in PHP from SQL-injection attacks. Huang *et al.* [11] detected Web application vulnerabilities by using dynamic analysis approaches, while Scott and Sharp [21] have a proxy-based approach to prevent cross site scripting attacks.

Mockus *et. al.* [19] used a fine-grained analysis to predict defect correction effort and the time-interval for which such an effort is needed. We share with them the fine-grained analysis approach, although we focus on vulnerability decay rather than on defect removal effort. Previous works [12, 16, 25] found that Weibull and exponential distributions capture defect-occurrence behavior across a wide range of systems. In our case, we find that such distributions are suited to model vulnerability decay. Calzolari *et al.* [2] used the predator and prey model borrowed from ecological dynamic system to model maintenance and testing effort. They found that, when programmers start to correct code defects, the effort spent to find new defects has an initial increase, followed by a decrease when almost all defects are removed.

Kim and Ernst [13], proposed an algorithm to prioritize the fixing of warnings detected by tools such as *FindBugs*, *Jlint* and *PMD*. We share with them the idea of using historical information to analyze how warnings—and vulnerabilities in our specific case—are fixed. However, Kim and Ernst focus on warnings removed by bug-fixing directly affecting source code lines containing the warning itself. They found (and our data indicates a consistent behavior) that this represents a very small percentage of warning/vulnerability removal. In our paper, we extend the analysis to vulnerabilities that disappear since other source code lines are changed, e.g., to "protect" the vulnerability.

3 Statically Detecting Code Vulnerabilities

There is number of commercial or freely available tools able to statically detect source code vulnerabilities, among others *FlawFinder*[2], *Pixy*[3], *RATS*[4], *ITS4*[5], and *Splint*[6]. For our analyses, we selected three freely available tools, *Splint*, *Rats*, and *Pixy*, in order to increase the range of vulnerability categories we can detect and the set of programming languages we can analyze. In the following we report a short description of these three tools.

Splint, previously known as LCLint, is an open source static analysis tool for ANSI C programs. It uses a lightweight static analysis technique to identify code vulnerabilities, including data flow analysis and type checking/inference.

RATS (Rough Auditing Tool for Security) is an open source tool which is able to analyze source code written in C, C++, Perl, PHP and Python. The tool performs a rough source code analysis — mostly based on pattern matching — and provides a relative assessment of the potential severity of the detected problems.

Pixy analyzes PHP programs and is particular suitable for detecting cross site scripting (XSS) and SQL injection vulnerabilities.

Vulnerability detection tools are often able to detect a large number of vulnerabilities at a level of detail not necessary for the purpose of the study described in this paper. Table 1 shows the different vulnerability categories that can be detected by the three tools adopted for our study. In the Table we also provide a brief description of each category, organizing them into four groups: *Input validation*, *Memory safety*, *Race/Control flow* and *Other* vulnerabilities. The classification is largely inspired from what reported on tool user manuals and on the Krsul's PhD thesis [15]. A more detailed description of the vulnerabilities studied in this paper can be found on a technical report [8] containing further analyses performed for a related study.

4 Data Extraction

This section describes the process we follow to extract data necessary for the analysis of source code vulnerability evolution. The data extraction process consists in a sequence of four steps reported below.

[2] http://www.dwheeler.com/flawfinder
[3] http://pixybox.seclab.tuwien.ac.at/pixy/
[4] http://www.securesoft.com/rats.php
[5] http://www.cigital.com/its4
[6] http://www.splint.org

Table 1. Vulnerability categories detected by each tool

VULNERABILITY	DESCRIPTION	SPLINT	RATS	PIXY
INPUT VALIDATION: concerns the incorrect validation of input data, which may lead to data corruption and, consequently, to a security vulnerability.				
XSS unconditional (XSSU)	Cross-site scripting: malicious content not properly stripped. from the output sent back to a user			√
XSS conditional (XSSC)	Cross-site scripting with the PHP configuration flag *register_globals* active.			√
SQL injection (SQLI)	Execution of unauthorized queries due to incorrectly filtered input.			√
Command Injection (CI)	Allows to inject and execute unauthorized commands.		√	√
File System (FS)	A weakness of the system which allows a unauthorized file system access.		√	
Net Problem (NET)	A weakness of the system which allows to exploit functions used to transfer data through a network.		√	
MEMORY SAFETY: concerns vulnerabilities dealing with memory access and allocation.				
Input Allocation (I)	Execution of unauthorized code by using the memory allocated for input data.	√	√	
Buffer Overflow (BO)	Execution of unauthorized programs by accessing data beyond the boundaries of a fixed-length buffer.	√	√	
Type Mismatch (TM)	Inconsistent, implicit type conversions, where the actual type differs from the expected one.	√	√	
Memory Access(M)	Incorrect use of memory management instructions which may cause, e.g., dangling and wild pointers.	√		
RACE/CONTROL FLOW CONDITIONS: arise when separate processes or threads of execution depend on some shared state.				
Race Check (RC)	Race condition caused by changes in a conditional statement, e.g., a security credential.	√	√	
Control Flow (CF)	Control flow instances where an undetermined order of evaluation produces an undefined behavior.	√		
OTHERS: vulnerabilities not specific to any particular exploiting technique.				
Random generation (RND)	Predictability of random number generators.		√	
Dead Code (DC)	Code not reached and/or no longer used.	√		

Step I: snapshots extraction. Since we are interested to perform a fine-grained analysis of vulnerability evolution, we look at changes committed in the Concurrent Versions Systems (CVS)/SubVersioN (SVN) repository rather than looking at releases of the software system. In particular, we rely on an analysis technique [10] that considers the evolution of a software system as a sequence of *Snapshots* (S_1, S_2,. . .,S_m) generated by a sequence of *Change Sets*, representing the logical changes performed by a developer in terms of added, deleted, and changed source code lines. Change sets can be extracted from a CVS/SVN history log using various approaches. We adopt a time-windowing approach that considers a change set as sequence of file revisions that share the same author, branch, and commit notes, and such that the difference between the timestamps of two subsequent commits is less or equal than 200 seconds [27].

Step II: differences identification and line tracing. To analyze the evolution of vulnerable source code lines over snapshots, we need to identify whether a change committed in the CVS/SVN consists in the addition of new lines, in the removal of existing lines, and/or in the change of existing lines. To this aim, we use an approach which identifies the differences between two versions of a source file introduced in [3]. Such approach identifies the set of changed, added and deleted source code lines between two subsequent snapshots by using `ldiff`[7], an improved `diff` algorithm. The tool combines the Levenshtein string edit distance with vector space models cosine similarity to determine whether a line has been changed or whether, instead, a change consists in the removal of an old line and in the addition of a new one. With such an information we are able to trace the line evolution among subsequent versions of a source code file with a high (over 90%) precision.

Step III: identification of vulnerable source code lines. Once we have identified snapshots (*Step I*) and traced the evolution of source code lines (*Step II*), we need to identify, for each snapshot, the set of source code lines that, according to the tools described in Section 3, contain vulnerabilities. To this aim, we run the vulnerability detection tools on the set of files that, on each snapshot, have been changed. The output of this step is, for each snapshot and for each source code file modified in the snapshot, the list of vulnerable source code lines along with a vulnerability description as extracted by the tool, and a vulnerability classification according to the taxonomy of Section 3.

Step IV: determining vulnerability changes among snapshots. The last step of this process is to trace a vulnerability across snapshots. In particular, by combining the information extracted at *Step II* (source code line tracing across snapshots) with the information extracted at *Step III* (vulnerable source code lines for each snapshot), we are able to determine:

- when (in which snapshot) a vulnerability appears in a source code line for the first time;
- whether a vulnerability identified on line l_i of file f_j in snapshot s_{k+1} is the same vulnerability identified on line $l_{i'}$ in snapshot s_k, where $l_{i'}$ in s_k corresponds to l_i in s_{k+1} according to the analysis of *Step II*;
- when a vulnerability disappears from the system. This can happen for two reasons. First, a source code line containing a vulnerability can be removed. Second, although the line has not been removed, the vulnerability is not detected anymore, either because of a change occurred on the line itself, or because of a change occurred somewhere else.

5 Empirical Study Definition

This section defines the empirical study that was carried out, details its context, and finally formulates the research

[7] http://rcost.unisannio.it/cerulo/tools.html

Table 2. Case study history characteristics

SYSTEM	SNAPS	RELEASES	KNLOC	FILES
Squid	6215	1.0–3.0	13.2–179.5	21–876
Samba	10531	1.9.16–3.0.0	36.1–344.7	56–889
Horde	6204	1.0.0–3.2rc2	0.482–43.6	5–542

questions it will address. The study is defined according to the Goal Question Metric Paradigm [1] and described following the template suggested by Wohlin *et al.* [24], as well as the guidelines for case study research provided by Yin [26]. The *goal* of this study is to perform a fine-grained analysis on the evolution of vulnerable source code lines. The *purpose* is to determine whether different vulnerabilities exhibit a particular evolution trend over the time, and how long vulnerabilities tend to remain in the system. The *quality focus* is the software system reliability and security, which can be affected by these statements. The *perspective* is of researchers, aimed at understanding whether the number of vulnerabilities of a given category tends to increase if compared with the system size evolution, whether vulnerabilities tend to be removed in correspondence of particular events (e.g., new releases), to investigate on the "life-span" of a vulnerability and on the likelihood a vulnerability has to be removed in the future.

The *context* deals with analyzing the evolution of vulnerable source code lines in three open source systems, namely *Squid*, *Samba*, and *Horde*. *Squid*[8] is a Web caching proxy, written in ANSI C, supporting HTTP, HTTPS, and FTP. *Samba*[9] is a software suite, written in ANSI C, that provides file and print services and allows for interoperability between Linux/Unix servers and Windows-based clients. *Horde*[10] is a general-purpose Web application framework written in PHP, providing classes for dealing with preferences, compression, browser detection, connection tracking, MIME handling, and more. Main characteristics of the three projects, i.e., number of snapshots and range of analyzed releases, and minimum-maximum non-commented KLOC (KNLOC) and source code files, are reported in Table 2. The fraction of unique vulnerabilities detected over all the analyzed snapshots that after a given time disappeared is reported in Table 3.

5.1 Research Questions

The research questions this study aim at answering are the following:

- **RQ1:** *How does the number of vulnerabilities vary during the time?* In particular, this research question analyzes the evolution of the vulnerability density defined as number of vulnerabilities per NKLOC.

[8] *http://www.squid-cache.org*
[9] *http://www.samba.org*
[10] *http://www.horde.org/*

Table 3. Number of Disappeared/Detected vulnerabilities.

Vulnerability	Squid	Samba	Horde
SPLINT			
Buffer Overflow	85/106	18/49	–
Control Flow	10/15	9/9	–
Dead Code	35/38	37/48	–
Memory	314/408	850/1075	–
Type Mismatch	315/700	540/1175	–
RATS			
Buffer Overflow	805/1638	2026/3305	–
Command Injection	0/1	–	12/12
File System	365/462	408/463	–
Input	25/46	64/287	197/181
Memory	5/9	36/65	–
Network	38/55	9/18	–
Race Check	45/84	132/146	23/25
Random	23/42	72/133	–
PIXY			
XSS Conditioned	–	–	49/49
XSS Unconditioned	–	–	121/157

- **RQ2:** *How long vulnerabilities tend to remain in the system before disappearing?* Specifically, it investigates whether vulnerabilities belonging to different categories had a significantly different decay, i.e., the time it remained in the system from its introduction until it disappeared.

- **RQ3:** *How can vulnerability decay be modeled with a statistical distribution?* This research question investigates whether the decays of different vulnerability categories follow a specific distribution.

5.2 Analysis Method

For *RQ1*, we mainly perform a qualitative analysis, by using plots where the x-axis indicate the snapshot timestamp (relative to the first snapshot analyzed) and the y-axis indicate the vulnerability density, i.e. the ratio between the number of vulnerability and NKLOC. We also consider when a new system release was made available. In addition to the qualitative analysis, we test whether the vulnerability density time series for different categories are stationary or not, i.e., whether they exhibit a trend. To this aim, we used the Augmented Dickey-Fuller (ADF) test [9], which tests the null hypothesis H_{01}*: the time series is not stationary*.

For *RQ2*, to test whether different vulnerability categories exhibit significant differences in terms of decay time, we used the Kruskal-Wallis test, which is a non-parametric test for testing the difference among multiple medians. The null hypothesis tested is H_{02}*: all medians of vulnerability decay times are equal*. In addition, we used the non-parametric, two-tailed Mann-Whitney test to perform pairwise comparisons among vulnerability categories. Since we are performing multiple tests on data extracted from

the same data set, it is appropriate to correct the significance level (95% in all the tests) using Bonferroni correction [18], i.e., dividing the significance by the number of tests performed (the number of combinations across categories). Thus, a p-value indicates a significant difference if it is $< 0.05/number\_of\_comparisons$. Truly, in this case the application of the Bonferroni correction is a sort of over-protection against the internal validity threat of fishing rate, since the overall hypothesis is tested using a multiple-median test and the pairwise comparison is only performed for the purpose of comparing categories. Other than computing p-values, it is also important to evaluate the difference magnitude. To this aim we used the Cohen d effect size [4] which is defined, for independent samples, as the difference between the means (M_1 and M_2), divided by the pooled standard deviation ($\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$) of both groups: $d = \frac{M_1 - M_2}{\sigma}$. The effect size is considered *small* for $d \sim 0.2$, *medium* for $d \sim 0.5$ and *large* for $d \sim 0.8$ or higher.

Regarding *RQ3*, we attempted to fit distributions of decays for different vulnerability categories to various statistical distributions, namely normal, exponential, Weibull, Gamma, and lognormal. In the following we only focus on exponential and Weibull, since these were the only distributions that fitted to our data, and that were used in literature to model defect decay [16]. The probability density function of the Weibull distribution is defined as:

$$f(x; k, \beta) = \frac{k}{\beta} \left(\frac{x}{\beta}\right)^{k-1} e^{-(x/\beta)^k} \quad (1)$$

where $k > 0$ is the *shape* parameter and $\beta > 0$ is the *scale* parameter. The exponential distribution is a particular case of Weibull distribution with k=1. In the exponential distribution the *rate* parameter λ is often used instead of the *scale* β, where $\lambda = 1/\beta$. To check whether a distribution could fit our data, we first estimated the distribution parameters using the method of Maximum Likelihood, which maximizes the likelihood that the set of data used for the estimation can be obtained from the statistical distribution modeled with the estimated parameters. Once estimated the distribution parameters, we used a non-parametric test, the Kolmogorov-Smirnov (KS) test to check whether the distribution was able to actually fit the data H_0*: there is no significant difference between the theoretical distribution and the actual data distribution*. Thus, the data set fits the distribution for p-values> 0.05. We used a maximum likelihood estimator available in the *fitdistr* function included in the MASS package of the *R* statistical environment[11] and the *ks.test* again available in *R*.

[11] *http://www.r-project.org*

6 Empirical Study Results

This section reports results of the empirical study defined in Section 5.

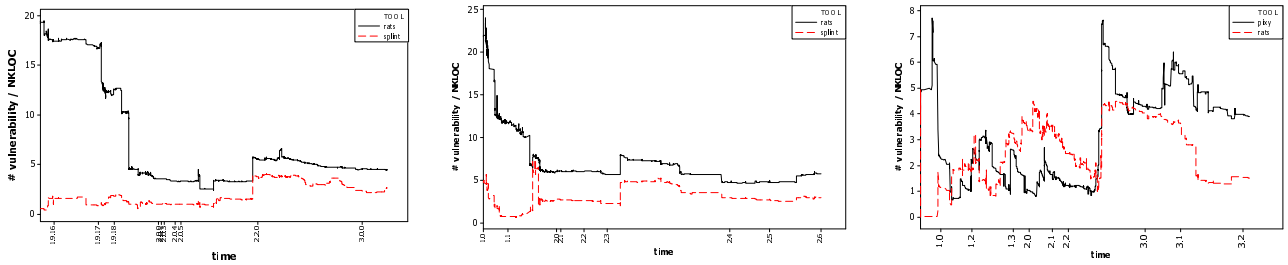
6.1 RQ1: How does the number of vulnerabilities vary during the time?

To answer *RQ1* we analyzed the evolution of vulnerability density over time. Due to space limitations, we only report the most interesting results; a complete set of vulnerability density evolution plots is included in a detailed technical report[12]. Figure 1 shows, for the three systems analyzed and for the tools used, the evolution of vulnerability density counting, for each snapshot, the whole set (i.e., all categories) of vulnerabilities detected. It can be noted that vulnerabilities detected with different tools exhibit almost the same consistent behavior. Vulnerabilities detected with *Rats* started with a high density, that tended to decrease over the time; vulnerabilities detected with *Splint* had a lower density: this can be explained by the thorough analysis *Splint* performs (data and control flow analysis) with respect to the simple pattern matching of *Rats*. We also noticed, in all systems we analyzed, patterns on vulnerability density increases followed by sudden decreases. This happened because, when a new pre-release is made available, it usually exhibits a high density of vulnerabilities, that tend to be removed with the subsequent release of security patches and updates.

Interesting behaviors can be noticed by analyzing vulnerability categories separately. Figure 2 shows the density evolution of vulnerability categories detected in *Horde* with *Pixy*. There is an increasing trend of *XSS Unconditioned* vulnerabilities, and a stationary behavior of *XSS Conditioned* vulnerabilities as confirmed by the ADF test reported in Table 4. *XSS Conditional* vulnerabilities, specifically related to the PHP *register_globals* variable, are partially reduced in correspondence of each major release, and completely removed in the last release candidate *3.2rc2*, when the *register_globals* variable was deprecated.

Figures 3 and 4 show some trends of *Buffer Overflows* in *Squid* and *Memory* problems in *Samba*, respectively, restricted to a short time window. Both exhibit — over the whole time frame analyzed — a stationary behavior as indicated in Table 4. It is interesting to note, in both cases, a periodic increment of vulnerability, due to system evolution and addition of new code, and a subsequent decrement due to security patches. In *Squid*, the partial increment of *Buffer Overflows* introduced in release *2.3STABLE3* has been removed with the subsequent security patches released in *2.4STABLE7* and *2.5STABLE7*, bringing the vulnerability density back to the level before *2.3STABLE3*. After a while,

[12] *http://www.rcost.unisannio.it/mdipenta/scam-tr.pdf*



(a) Samba (b) Squid (c) Horde

Figure 1. Evolution of vulnerability density

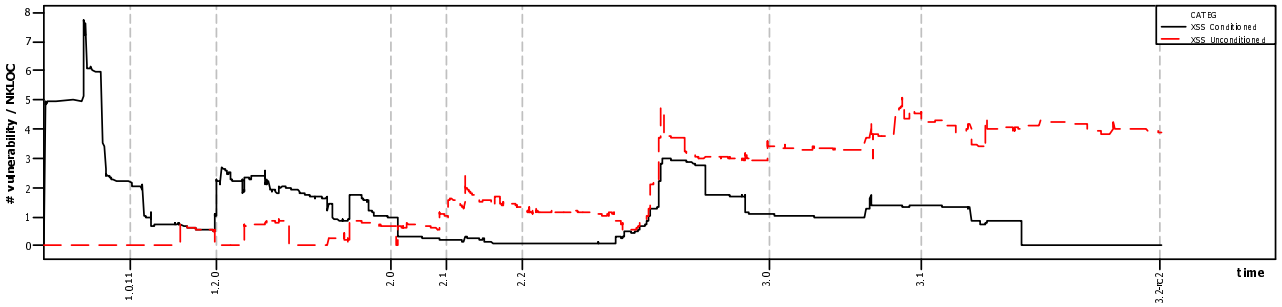


Figure 2. Horde – Vulnerability density detected with Pixy

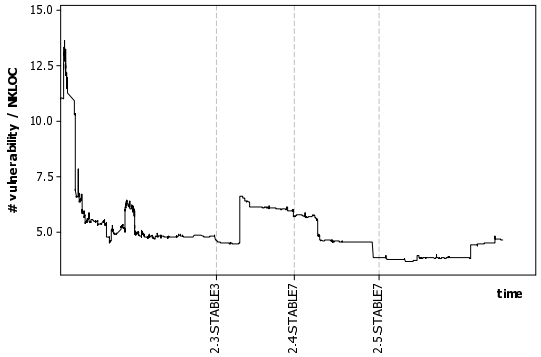


Figure 3. Squid – Buffer Overflow density (detected with Splint)

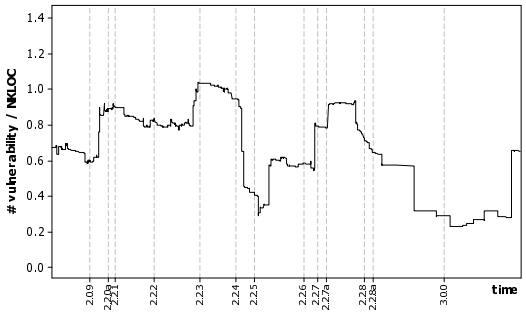


Figure 4. Samba – Memory problem density (detected with Splint)

a new increment occurs when new features are added. In *Samba*, a similar periodic behavior is exhibited between releases *2.0.9* and *3.0.0*. The alpha release *2.2.0a* contains a local maximum of *Memory* problems which is partially reduced by the subsequent (not security-related) patches from *2.2.1* to em 2.2.4, and highly reduced with the security patch *2.2.5*. This behavior occurs again with other patches, from *2.2.6* to *2.2.8a*, until in *3.0.0* a stable release is produced.

Summarizing, similar periodic trends can be observed in almost all cases whether the vulnerability density series is stationary (see Table 4), while in all other cases an increasing or decreasing trend can be observed, similarly to *XSS*

Table 4. Results of ADF stationarity test on vulnerability density time series (p-values in boldface indicate a stationary time series)

	Squid	Samba	Horde
SPLINT			
BO	**0.01**	0.77	–
CF	0.92	**0.01**	–
DC	0.24	0.5	–
M	**0.02**	**0.01**	–
TM	0.28	0.84	–
RATS			
BO	**0.01**	**0.01**	–
CI	0.93	–	0.78
FS	0.31	0.1	–
I	0.20	**0.03**	0.27
M	0.20	**0.01**	–
NET	**0.01**	**0.01**	–
RC	**0.01**	**0.04**	**0.01**
RND	0.28	0.63	–
PIXY			
XSSC	–	–	**0.01**
XSSU	–	–	0.56

Unconditioned vulnerabilities detected in Horde (see Figure 2). The first behavior is consistently motivated by the intrinsic periodicity of stable and unstable releases, while the second needs further investigations to explain.

6.2 RQ2: How long vulnerabilities tend to remain in the system before disappearing?

To answer *RQ2* we analyzed, for each vulnerability, its decay. Figure 5 shows boxplots of decays (expressed in days) for each vulnerability category detected in the three systems with the different tools.

The Kruskal-Wallis test indicates a significant differences among different vulnerabilities detected with *Splint*. In particular, for Samba there exist a significant difference among decays of different vulnerability categories (p-value $< 1.96 \cdot 10^{-12}$), and the same happens for Squid (p-value = 0.00029). For vulnerabilities detected with *Rats*, the Kruskal-Wallis indicates a significant difference for Samba (p-value $< 2.2 \cdot 10^{-16}$) and Squid (p-value$= 7.65 \cdot 10^{-11}$), while not for Horde (p-value = 0.1919). Finally, for vulnerabilities detected with *Pixy* on Horde, no significant difference was found between decays of XSS Conditioned and XSS Unconditioned (p-value = 0.06923).

To perform a deeper comparison, we compared all pairs of vulnerability categories using a Mann-Whitney two-tailed test. Although for the purpose of such a comparison it is not strictly necessary, we applied the Bonferroni correction that, considering the number of tests performed on each data sets, decreases the p-value significance threshold to 0.005 for *Splint* vulnerabilities and 0.0024 for *Rats* vulnerabilities.

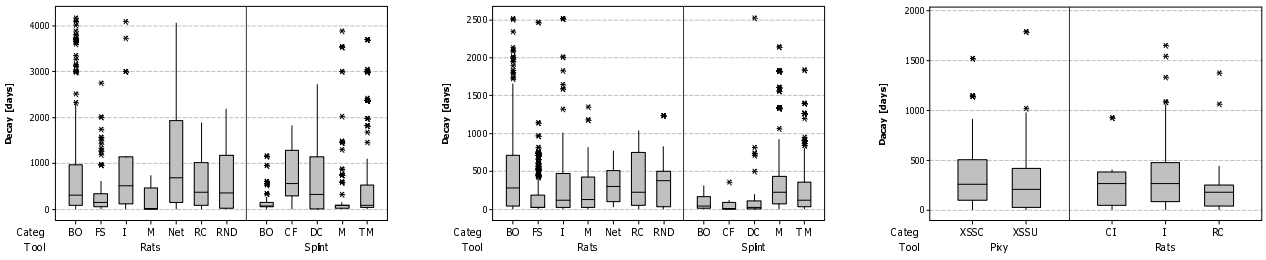
Results are shown in Table 5 and Table 6 for *Rats* and *Splint* respectively. The significant values (without Bonferroni correction) are shown in boldface; where the value is significant even after the correction the value is followed by a star (*) symbol. The tables also reports the Cohen d effect size (negative values indicate that the row-vulnerability mean decay is smaller than the column-vulnerability mean decay).

For Samba-*Splint* we found that Buffer Overflows disappeared significantly faster than Control Flow, Memory and Type Mismatch vulnerabilities, suggesting the particular attention paid by developers to this kind of vulnerability, while others such as Memory and Type Mismatch can be considered a kind of physiological and not necessarily harmful vulnerability for C programs. Also, Dead Code disappeared faster than Memory and Type Mismatch vulnerabilities, indicating the execution of activities such as refactoring aimed at improving the code quality (confirming what found in *RQ1* related to periodic patches that decrease vulnerabilities. For Samba-*Rats*, it emerges that FS problems disappeared significantly faster than all other vulnerabilities; this can be due to the specific characteristics of the application (distributed file sharing). Results for Buffer Overflows obtained with *Splint* were not confirmed with *Rats*, since the latter detects a larger number of Buffer Overflow vulnerabilities, many of which not necessarily harmful.

For Squid-*Splint* vulnerabilities, results obtained for Samba are partially confirmed, i.e., Buffer Overflows disappeared significantly faster than Control Flow, Dead Code and Type Mismatch vulnerabilities. For Squid-*Rats* vulnerabilities, Buffer Overflows decay significantly faster than FS, Memory, Net and Random Generation problems. FS problems disappeared faster than Input, Net, and Race Check problems. Buffer Overflow represents the kind of vulnerability developers tend to remove faster, because of its harmfulness; this is particularly true for a Web proxy like *Squid* where it can be a major cause of attacks.

6.3 RQ3: How can vulnerability decay be modeled with a statistical distribution?

Finally, we investigated, according to the method described in Section 5.2, whether the vulnerability decay can be modeled using any particular statistical distribution. Table 7 reports distribution parameters and KS test p-values for vulnerability categories where a fitting distribution was found. As it can be noticed, in most cases the vulnerability decay fits an exponential distribution, but in two cases — Control Flow Problems and Dead Code for Samba-Splint — where data fitted a Weibull distribution. Comparing this result with existing literature modeling defects decay with



(a) Squid (b) Samba (c) Horde

Figure 5. Boxplot of decays for different vulnerabilities

Table 5. Comparing decay of Rats vulnerabilities: p-value (Cohen d effect size)

SAMBA						
	FS	I	M	NET	RND	RC
BO	<**0.001*** (0.60)	<**0.001*** (0.10)	0.08 (0.31)	0.97 (0.21)	0.20 (0.11)	0.71 (0.18)
FS	–	<**0.001*** (-0.39)	<**0.001*** (-0.28)	<**0.001*** (-0.51)	<**0.001*** (-0.52)	<**0.001*** (-0.46)
I	–	–	0.74 (0.15)	0.27 (0.05)	0.67 (-0.01)	0.15 (0.03)
M	–	–	–	0.17 (-0.15)	0.34 (-0.21)	0.23 (-0.15)
NET	–	–	–	–	0.88 (-0.09)	0.65 (-0.01)
RND	–	–	–	–	–	0.80 (0.07)

SQUID						
	FS	I	M	NET	RND	RC
BO	<**0.001*** (0.63)	0.14 (-0.37)	0.05 (0.75)	<**0.001*** (-0.50)	0.91(0.03)	0.65 (0.07)
FS	–	<**0.001*** (-0.88)	0.14 (0.22)	<**0.001*** (-1.06)	0.13 (-0.66)	<**0.001*** (-0.73)
I	–	–	**0.03** (0.96)	0.59 (-0.10)	0.33 (0.42)	0.31 (0.46)
M	–	–	–	**0.01** (-1.14)	0.09 (-0.80)	**0.03** (-0.89)
NET	–	–	–	–	0.06 (0.56)	0.06 (0.62)
RND	–	–	–	–	–	0.84 (0.62)

Table 6. Comparing decay of Splint vulnerabilities: p-value (Cohen d effect size)

SAMBA				
	CF	DC	M	TM
BO	**0.04** (0.25)	0.83 (-0.31)	<**0.001*** (-0.86)	< **0.001*** (-0.75)
CF	–	0.69 (-0.39)	0.49 (-0.94)	0.49 (-0.84)
DC	–	–	**0.01** (-0.31)	**0.03** (-0.15)
M	–	–	–	0.60 (-0.21)

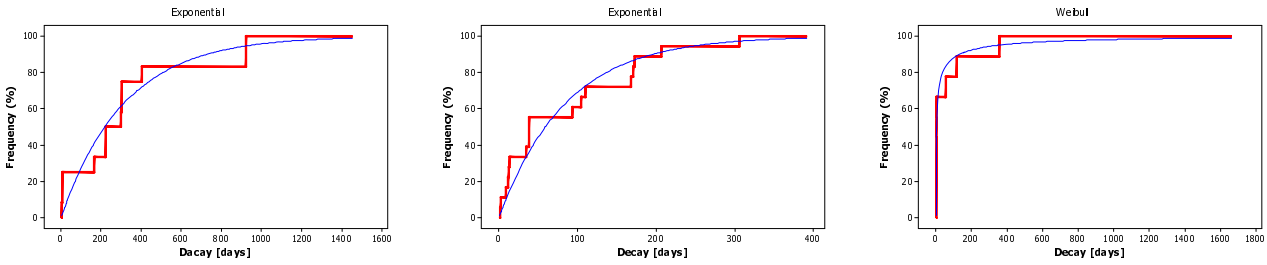
SQUID				
	CF	DC	M	TM
BO	**0.012** (-1.05)	**0.04** (-0.77)	0.14(0.00)	**0.02** -(0.48)
CF	–	0.53(0.11)	< **0.001*** (0.87)	0.11 (0.28)
DC	–	–	< **0.001*** (0.66)	0.35 (0.16)
M	–	–	–	< **0.001*** (0.43)

Table 7. Results of distribution fitting

SYSTEM	TOOL	VULN. CATEG.	FITTED DISTR.	PARAM		KS TEST P-VALUE
				λ ($1/\beta$)	k	
Squid	Splint	CF	Exp	0.0014	–	0.77
Squid	Rats	I	Exp	0.00093	–	0.23
		M	Exp	0.0054	–	0.08
		NET	Exp	0.00083	–	0.21
		RC	Exp	0.0016	–	0.31
		RND	Exp	0.0016	–	0.16
Samba	Splint	BO	Exp	0.012	–	0.53
		CF	Weibull	0.16	0.27	0.65
		DC	Weibull	0.024	0.35	0.32
Samba	Rats	M	Exp	0.0037	–	0.054
		NET	Exp	0.0031	–	0.55
Horde	Rats	CI	Exp	0.0032	–	0.54
		RC	Exp	0.004	–	0.51

statistical distributions [12, 16, 25], it can be noted that vulnerabilities — i.e., possible causes of reliability or security problems — decay following laws similar to defects.

Examples of Cumulative Distribution Functions (CDFs) are shown in Figure 6, where the actual CDF (thick line) is compared with the the theoretical CDF (thin line). The fitting with both exponential and Weibull distribution can be explained as follows: vulnerabilities have a high likelihood to disappear — i.e.., developers remove or protect them — shortly after vulnerabilities have been introduced in the system. As time increases, the likelihood a vulnerability has to be removed decreases following an exponential or Weibull probability density function. This can suggest that developers do not care about vulnerabilities which remain in the system for a long period of time because they believe that such vulnerabilities are not particularly harmful, i.e., would not cause serious security or reliability prob-



(a) Squid-Rats: Input Problem (b) Samba-Splint: Buffer Overflow (c) Samba-Splint: Control Flow Problem

Figure 6. Examples of theoretical (thin line) and empirical (thick line) CDF

lems. Indeed, further investigation is needed to confirm this interpretation.

7 Threats to Validity

This section discusses threats to validity that can affect the results reported in Section 6, following guidelines provided for case study research [26].

Construct validity threats concern the relationship between theory and observation; in this context they are mainly due to errors introduced in measurements. In particular, our results can be affected by performances of the adopted static vulnerability detection tools. Although tools can detect false positives — i.e., vulnerabilities that do not really cause problems — most of our study (*RQ2* and *RQ3*) focuses on vulnerabilities that disappear, thus vulnerabilities that actually required preventive or corrective maintenance activities aimed at removing them. Also, there may be false negatives — i.e., vulnerabilities not detected — although we limited this problem by performing the analysis with two different tools (*Rats* and *Splint* for C code, *Rats* and *Pixy* for PHP code) working in different ways and detecting different kinds of vulnerabilities. Also, although some false negatives can still be present, the number of detected ones is enough to make some considerations about their evolution. Another measurement problem can be due to the fact that we considered a vulnerability as removed either when its source code line was maintained, or when the line was removed from the system. Especially in the second case, it is likely that the vulnerable instruction was removed for purposes different from security improvement (e.g., system evolution or refactoring). However, in this paper we prefer to make a comprehensive analysis considering all causes of removal, and leaving further analyses for future work.

Threats to *internal validity* did not affect this particular kind of study, being it an explorative study [26].

Reliability validity threats concern the possibility of replicating this study. Analysis tools are available for downloading, as well as code and bug repositories of the analyzed systems. The data extraction process is detailed in Section 4.

Threats to *external validity* concern the possibility of generalize our findings. The study was performed on three different systems representative of different kinds of network applications. Nevertheless, analyses on further systems are desirable, as well as the use of vulnerability detection tools different from those we adopted.

8 Conclusions

This paper reported a fine-grained analysis, performed on three different software systems, of the evolution of statically detected source code vulnerabilities. We found that the vulnerability density is often stationary, with increases occurring in correspondence of system pre-releases, followed by decreases made possible by security patches or other maintenance tasks aimed at improving the system quality. Some vulnerability categories (e.g., buffer overflows) tend to have, in some cases, a shorten decay time due to their potential harmfulness. Finally, the decay of several vulnerability categories can be modeled with exponential or Weibull distributions, used in the past to model defect decay. These distributions indicate that the likelihood a vulnerability has to disappear from the system exponentially decreases with the time.

Future work aims at performing further investigations about modeling vulnerability decays with statistical distributions, at relating the vulnerability decay with the occurrence of bugs, and in general at performing a qualitative analysis of vulnerability evolution by using data from source code repositories and bug tracking systems.

9 Acknowledgments

The authors are partially supported by the project METAMORPHOS (MEthods and Tools for migrAting software systeMs towards web and service Oriented aRchitectures: exPerimental evaluation, usability, and tecHnOlogy tranSfer), funded by MiUR (Ministero dell'Università e della Ricerca) under grant PRIN2006-2006098097.

References

[1] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[2] F. Calzolari, P. Tonella, and G. Antoniol. Maintenance and testing effort modeled by linear and nonlinear dynamic systems. *Information & Software Technology*, 43(8):477–486, 2001.

[3] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14. IEEE CS, 2007.

[4] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1988.

[5] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 266, Washington, DC, USA, 2003. IEEE Computer Society.

[6] C. Dahn and S. Mancoridis. Using program transformation to secure C programs against buffer overflows. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 323, Washington, DC, USA, 2003. IEEE Computer Society.

[7] C. Del Grosso, G. Antoniol, M. Di Penta, P. Galinier, and E. Merlo. Improving network applications security: a new heuristic to generate stress testing data. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1037–1043, New York, NY, USA, 2005. ACM.

[8] M. Di Penta, L. Cerulo, and L. Aversano. An empirical study on the evolution of vulnerable code. Technical report, Dept. of Engineering, Univ. of Sannio, Italy, 2008. *http://www.rcost.unisannio.it/vuln-tr.pdf*.

[9] D. A. Dickey and W. A. Fuller. Likelihood ratio statistics for autoregressive time series with a unit root. *Econometrica*, 49(4):1057–72, June 1981.

[10] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13. IEEE Computer Society, 2003.

[11] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2003. ACM.

[12] W. Jones. Reliability models for very large software systems in industry. In *International Symposium on Software Reliability Engineering*, pages 35–42, 1991.

[13] S. Kim and M. D. Ernst. Which warnings should i fix first? In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 45–54, 2007.

[14] B. Korel and A. M. Al-Yami. Assertion-oriented automated test data generation. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 71–80, Washington, DC, USA, 1996. IEEE CS.

[15] I. V. Krsul. *Software vulnerability analysis*. PhD thesis, West Lafayette, IN, USA, 1998. Major Professor-Eugene H. Spafford.

[16] P. L. Li, M. Shaw, J. Herbsleb, B. Ray, and P. Santhanam. Empirical evaluation of defect projection models for widely-deployed production software systems. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2004. ACM.

[17] E. Merlo, D. Letarte, and G. Antoniol. Automated protection of php applications against SQL-injection attacks. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 191–202, Washington, DC, USA, 2007. IEEE Computer Society.

[18] R. G. Miller. *Simultaneous statistical inference. 2nd*. Springer Verlag, 1981.

[19] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 274–284, Washington, DC, USA, 2003. IEEE CS.

[20] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[21] D. Scott and R. Sharp. Abstracting application-level web security. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 396–407, New York, NY, USA, 2002. ACM.

[22] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Softw. Pract. Exper.*, 30(1):61–79, 2000.

[23] L. Wang, J. R. Cordy, and T. R. Dean. Enhancing security using legality assertions. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 35–44, Washington, DC, USA, 2005. IEEE CS.

[24] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[25] A. Wood. Predicting software reliability. *IEEE Computer*, 9:69–77, 1999.

[26] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.

[27] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE CS, 2004.