After-Life Vulnerabilities: A Study on Firefox Evolution, its Vulnerabilities, and Fixes*

Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen

Università degli Studi di Trento, I-38100 Trento, Italy
{massacci,neuhaus,vhnguyen}@disi.unitn.it

Abstract. We study the interplay in the evolution of Firefox source code and known vulnerabilities in Firefox over six major versions (v1.0, v1.5, v2.0, v3.0, v3.5, and v3.6) spanning almost ten years of development, and integrating a numbers of sources (NVD, CVE, MFSA, Firefox CVS). We conclude that a large fraction of vulnerabilities apply to code that is no longer maintained in older versions. We call these *after-life vulnerabilities*. This complements the Milk-or-Wine study of Ozment and Schechter—which we also partly confirm—as we look at vulnerabilities in the reference frame of the source code, revealing a vulnerabilitiy's future, while they looked at its past history. Through an analysis of that code's market share, we also conclude that vulnerable code is still very much in use both in terms of instances and as global codebase: CVS evidence suggests that Firefox evolves relatively slowly.
This is empirical evidence that the software-evolution-as-security solution—patching software and automatic updates—might not work, and that vulnerabilities will have to be mitigated by other means.

1 Introduction

The last decade has seen a significant push towards security-aware software development processes in industry, such as Microsoft's SDL [1], Cigital's BSIMM [2], and many other processes that are specific to other software vendors.

In spite of these efforts, software is still plagued by many vulnerabilities and the current trend among software vendors is to *counter the risk of exploits by software evolution*: security patches are automatically pushed to end customers, support for old versions is terminated, and customers are pressured to move to new versions. The idea is that, as new software instances replace the old vulnerable instance, the eco-system as a whole progresses to a more secure state.

Beside the social good (improvement of the security of the ecosystem) this model also has some significant economic advantages for software vendors: the simultaneous maintenance of many old versions is simply too costly to continue.

* This work is supported by the European Commission under projects EU-FET-IP-SECURECHANGE and EU-FP7-IST-IP-MASTER.

We call the time after which a software product is no longer supported that product's *end-of-life*[1]. Of course, having reached the end-of-life doesn't mean that the product is no longer in use: entire conferences are devoted to the issue of maintaining and wrapping legacy code. This product existence in *after-life* can have interesting security implications for the security of the ecosystem because a product should only reach end-of-life when

- the number of unpatched vulnerabilities in that product is small; or
- the number of active users of after-life code is small.

The key question that we try to address in this paper is whether there is some empirical evidence that software-evolution-as-a-security-solution is actually a solution, i.e., leads to less vulnerable software over time.

In this paper, we report our empirical findings on a major empirical study on the evolution of Mozilla Firefox. After studying 899 vulnerabilities in Mozilla Firefox from versions v1.0 to v3.6, we find:

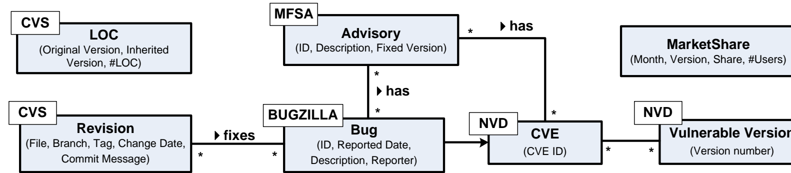
1. Many vulnerabilities for Firefox versions are discovered when these versions are well in their after-life. These *after-life vulnerabilities* account for at least 30% for Firefox v1.0.
2. Most disturbingly, there are still *many after-life instances* of Firefox.
3. There exists a statistically significant difference between *local* vulnerabilities (*i.e.*, those discovered and fixed in a same version) and *inherited* vulnerabilities (*i.e.*, those discovered in version x, but applicable also to some version $y < x$) or *foundational* (*i.e.*, inherited from the first version). By pure statistics change, foundational vulnerabilities are significant more than they should be, meanwhile, inherited on are significant less than they should be.
4. A possible explanation of this phenomenon is *slow code evolution*: a lot of code is retained between released versions. Code originating in Firefox v1.0 is over 40% of the code in v3.6.

These findings show that Ozment and Schechter's notion of foundational vulnerabilities [3] can be generalized to the more general case of *inherited* vulnerabilities. Ozment and Schechter's limitation of to foundational vulnerabilities (instead of the more general class of inherited ones) might be due to their particular case study, or might be due to a methodological specialty, which we analyze in §5. However we are not able to make sharp conclusions yet: foundational vulnerabilities are significantly more than they should be but inherited ones are significantly less than they should be. None of them is the majority.

These results could be explained by the quality of the slow evolution where code of v1.0 (*i.e.*, foundational code) largely dominates the inherited part: the good fraction of the code is the one that is retained.

The existence of many after-life vulnerabilities (contribution 1) and many after-life survivors (contribution 2), in spite of producers' efforts to eradicate

[1] Some companies provide security patches but not regular bugfixes for out-of-date products. In this case end-of-life is the point when all maintenance stops.



Rectangles denote tables; icons at the top left corner of tables denote the source database. Two additional tables (not shown) are used to trace back the evolution of individual lines of code from CVS and Mercurial, and Firefox's market share.

Fig. 1. Simplified database schema

them, is an interesting phenomenon by itself. It shows that security pushes during deployment and the active life of the project cannot eliminate vulnerabilities.

Further, it shows that software-evolution-as-security-solution is not really a solution: the fraction of vulnerable software instances that is globally present may be too high to ever attain herd immunity [4].

The remainder of this paper is structured as follows. First, we describe our methods of data acquisition and experiment setup (§2), after which we introduce important concepts such as foundational or inherited vulnerability (§3). Next, we report our detailed findings on after-life vulnerabilities (§4), revisit and challenge the "Milk or Wine" findings (§5), and present the data on software evolution (§6). Finally, we analyze the threats to validity (§7) and conclude by discussing our findings (§9).

2 Data Acquisition and Experiment Setup

Here we briefly describe how to acquire aggregated vulnerability data by the integration of a number of data sources; a more complete description is contained in a previous work [5].

Fig. 1 presents the abstract schema of our database and shows our data sources. The *Advisory* table holds a list of Mozilla Firefox-related Security Advisories (MFSA).[2] Rows in this table contain data extracted from MFSA (*e.g.*, vulnerability title, announced date, fixed versions); it also maintains links to Bugzilla and the CVE, captured in tables *Bug* and *CVE*, respectively. The *Bug* table holds Bugzilla-related information, including bugID, reported date, and description. Some Bugzilla entries are not publicly accessible, such as reported but unfixed vulnerabilities. In these cases, the reported date is set to the corresponding advisory's announced date. The *VulnerableVersion* table keeps the National Vulnerability Database (NVD) reported vulnerable version list for each CVE. The *Revision* table holds information of those code check-ins that fix bugs.

There are two additional tables that help to study the evolution of Firefox in term of code base, table *LOC*, and in term of global usage, table *MarketShare*. In

[2] Not all MFSAs relate to Firefox; some relate to other Mozilla products.

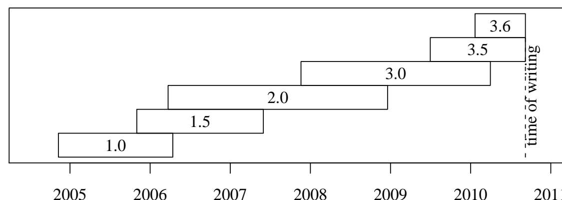


Fig. 2. Release and retirement dates for Firefox versions.

order to estimate the market share and hence the number of users of a particular Firefox version, we collect data on the global evolution of Firefox usage since November 2007 from Net MarketShare[3]. This website collects browser data from visitors of about 40,000 websites. It provides monthly market share data free of charge, but does not show the absolute number of users. For this, we use Internet World Stats[4], which gives numbers of internet users since December 1995. This website does not provide data for every month from 2007, hence we use linear interpolation to calculate the missing values.

3 Versions and Vulnerabilities

We looked at six major Firefox versions, namely v1.0, v1.5, v2.0, v3.0, v3.5 and v3.6. Their lifetimes[5] can be seen from Fig. 2. At the time of writing, the versions of Firefox that were actively maintained were v3.5 and v3.6. Therefore, the rectangles representing version v3.5 and v3.6 actually extend to the right. There is very little overlap between two versions that are one release apart, *i.e.*, between v1.0 and v2.0, v1.5 and v3.0, v2.0 and v3.5, or v3.0 and v3.6. This is evidence of a conscious effort by the Mozilla developers to maintain only up to two versions simultaneously.

This picture seems to show a quick pace of software evolution as in the time span of slightly more than an year, we have a new version replacing an old one. As we shall see, this is not the case.

In order to find out to which version a vulnerability applies, we look at the NVD entry for a particular vulnerability and take the earliest version for which the vulnerability is relevant. For example, MFSA 2008-60 describes a crash with memory corruption. This MFSA links to CVE 2008-5502, for which the NVD asserts that versions v3.0 and v2.0 are vulnerable. The intuitive expectation (confirmed by the tests) is that the vulnerability was present already in v2.0 and that v3.0 inherited it from there.

[3] `http://www.netmarketshare.com/`
[4] `http://www.internetworldstats.com/emarketing.htm`
[5] `https://wiki.mozilla.org/Releases`

Table 1. Breakdown of Cumulative Vulnerabilities in Firefox

The Total row shows the cumulative vulnerabilities for each version. Other rows display the vulnerabilities that a version inherits from retrospective ones.

Inherit	Affected Version					
from	v1.0	v1.5	v2.0	v3.0	v3.5	v3.6
v1.0	334	255	184	142	45	13
v1.5	—	227	119	15	0	0
v2.0	—	—	149	23	1	1
v3.0	—	—	—	102	35	5
v3.5	—	—	—	—	73	41
v3.6	—	—	—	—	—	14
Total	**334**	**482**	**452**	**282**	**154**	**74**

Table 1 shows the breakdown of cumulative vulnerabilities affecting to each version of Firefox. An entry at column x and row y indicates the number of vulnerabilities applied to version x inherits from version y. Obviously there are vulnerabilities applying for more than one version. Thus, they are counted several times in this table. It is a mistake if we sum all numbers and conclude about the total vulnerabilities of Firefox.

Based on versions that a vulnerability affects to, it can be classified into following sets:

- *Inherited vulnerabilities* are ones affect to a series of consecutive versions.
- *Foundation vulnerabilities* are inherited ones, but apply also to v1.0.
- *Local vulnerabilities* are known to apply to only one version.

Our definition of foundational vulnerability is weaker (and thus more general) than the one used by Ozment and Schechter [3]. We do not claim that there exists some code in version v1.0 that is also present in, say, v1.5 and v2.0 when a vulnerability is foundational. For us it is enough that the vulnerability applies to v1.0, v1.5 and v2.0. This is necessary because many vulnerabilities (on the order of 20–30%) are not fixed. For those vulnerabilities it is impossible, by looking at the CVS and Mercurial sources without extensive and bias-prone manual analysis, to identify the code fragment from which they originated.

When we tabulate which vulnerabilities affect which versions, we can in theory get $N = 2^6 - 1$ different results, depending on which of the six versions is affected.[6] If all N combinations were equally likely, vulnerable versions separated by not vulnerable versions would be commonplace: there are $2^n - 1 - \sum_{k=0}^{n}(n-k) = 2^n - n(n+1)/2 - 1$ such arrangements, which we call *regressions*. For $n = 6$, that is 42 out of 63.

Once we exclude regressions, there are 6 combinations in which a vulnerability only applies to a single version, 5 combinations with foundational vulnerabilities and 10 inherited but not foundational vulnerabilities.

[6] The only combination that is not allowed is when no version is affected. If no version is affected, why is it a vulnerability?

Table 2. Examples of inherited and foundational vulnerabilities.

A vulnerability is *inherited* when it appears first in some version and affects a nonempty string of consecutive versions (and no others); it is also *foundational* if it applies to the first version, v1.0. Non-consecutive affected versions point to a regression, which we did not find in the data.

Affected Version						
v1.0	v1.5	v2.0	v3.0	v3.5	v3.6	Remark
		×	×			Inherited (from v2.0)
		×	×	×		Inherited (from v2.0)
	×	×	×	×	×	Inherited (from v1.5)
×	×	×	×			Foundational
×						Local
			×			Local
	×		×			Regression (rare)

Another definition that we will use in this paper is the notion of *After-life Vulnerability*: a vulnerability which applies to a version which is no longer maintained. Since a vulnerability can apply to many versions, the same vulnerability can be an after-life vulnerability for v1.0 and a current vulnerability for v3.6.

4 After-Life Vulnerabilities and the Security Ecosystem

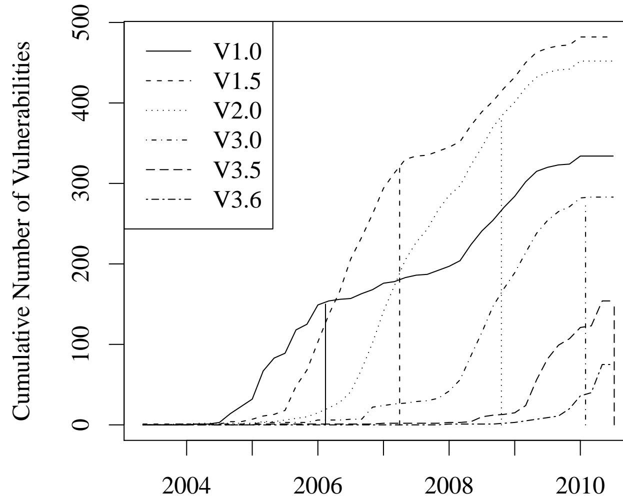
Our first research question was whether many vulnerabilities were discovered that affected after-life versions.

As Fig. 3 shows, this is indeed the case. This figure shows the cumulative number of vulnerabilities for each of the six Firefox versions versus time. We also marked the respective version's end of life. If there were no or just a few vulnerabilities discovered after end-of-life, the slopes of the curves should be zero, or close to zero, after that point. Since this is clearly not the case, after-life vulnerabilities do in fact exist.

In order to evaluate the impact of after-life vulnerabilities we consider the market share of the various versions and the attack surface of code that is around. The intuition is to calculate the LOC on each version that are currently available to attackers, either by directly attacking that version or by attacking the fraction of that version that was inherited in later versions.

Let $users(v,t)$ be the number of users of Firefox version v at time t, and let $loc(p,c)$ be the number of lines of code that the $c$urrent version c has inherited from the $p$revious version p. Then the total number of lines of code in version c is $\sum_{1 \le p \le c} loc(p,c)$. In order to get an idea how much inherited code is used, we define a measure *LOC-users* as

$$LOC\text{-}users(v,t) = \sum_{1 \le p \le v} users(p,t) \cdot loc(p,v). \tag{1}$$



Cumulative number of vulnerabilities for the various Firefox versions. End-of-life for a version is marked with vertical lines. As is apparent, the number of vulnerabilities continues to rise even past a version's end-of-life.

Fig. 3. Vulnerabilities discovered in Firefox versions

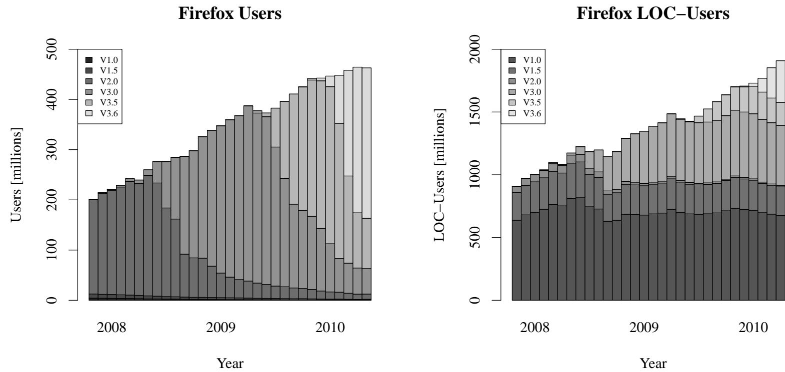
This is an approximation because the amount of code inherited into version v varies with time, therefore, $loc(p, v)$ is time-dependent. In this way, we eliminate transient phenomena for this high-level analysis.

Fig. 4 shows the development of the number of users and of LOC-users over time. It is striking to see the number of Firefox v1.0 go down to a small fraction, while the LOC-users for version v1.0 stays almost constant.[7]

An important observation is that even the "small" fraction of users of older versions (full of after-life vulnerabilities that will never be fixed, as we have just shown) accounts for hundreds of thousands of users. You can imagine wandering in Florence and each and every person that you meet in the city still uses old Firefox v1.0.

This might have major implications in terms of achieving herd immunity as the number of vulnerable hosts would be always sufficient to allow propagation of infections [4].

[7] A generalised linear model in which the residuals are modeled as an auto-regressive time series gives a statistically significant slope of -0.27 million LOC-users per month for version v1.5 ($p < 0.001$), and comparable numbers for the other versions.



Number of Firefox users (left) and LOC-users (right). While the number of users of Firefox v1.0 is very small, the amount of Firefox v1.0 *code* used by individual users is still substantial.

Fig. 4. Firefox Users vs LOC Users

5 "Milk or Wine" Revisited

Ozment and Schechter in their "Milk or Wine" study [3] look at vulnerabilities in OpenBSD and find evidence that most vulnerabilities are foundational, which they define as having been in the software in release 2.3, the earliest release for their study.

To verify this finding in Firefox, we need the number of foundational and inherited vulnerabilities, which are not be able to calculate from Table 1. We obtain these values by counting a vulnerability for once based on the version that a vulnerability is reported, and the earliest version that it applies to.

Table 3 shows the number of vulnerability entries that were created during the lifetime of a version x ("entered for"), where the earliest version to which it applies is y ("applies to"). In the terms of Table 2, an entry in the table with a particular value for x and y means that the leftmost '$\times$' symbol is in the slot corresponding to y, and the rightmost '$\times$' symbol is in the slot for x. For example, there were 15 vulnerabilities that were discovered during the lifetime of v3.0, which also applied to v1.5, but not to any earlier version. Since we have no regressions (see above), these 15 vulnerabilities also apply to the intermediate v2.0. Therefore, the total vulnerabilities applies to version x is sum of all entries which column is greater than x and row is lesser than x. For instance, total vulnerabilities for v2.0 $= 42 + 97 + 32 + 13 + 104 + 15 + 126 + 22 + 1 = 452$.

We can now easily categorise the vulnerabilities in the table according to the categories that interest us: *inherited* vulnerabilities are the numbers above the diagonal (they are carries over from some previous version); *foundational* vulnerabilities are those in the first row, excluding the first element (they are carries over from version v1.0); and *local* vulnerabilities are those on the diagonal (they are fixed before the next major release and are not carried over).

Table 3. Vulnerabilities in Firefox

Number of vulnerabilities entered for a specific Firefox version (columns) and versions in which that vulnerability originated (rows). Since we look only at vulnerabilities that have been fixed fixed, the entries below the diagonal are all zero.

first known	entered for					
to apply to	v1.0	v1.5	v2.0	v3.0	v3.5	v3.6
v1.0	79	71	42	97	32	13
v1.5	—	108	104	15	0	0
v2.0	—	—	126	22	0	1
v3.0	—	—	—	67	30	5
v3.5	—	—	—	—	32	41
v3.6	—	—	—	—	—	14

The most important claim by Ozment and Schechter was that most vulnerabilities are foundational. If there were no difference between foundational vulnerabilities and others, all numbers in the table would be equal to $899/21 = 42.8$. Out of the 21 matrix entries, 5 would be foundational and 16 nonfoundational, so there would be $5 \cdot 899/21 = 214$ foundational and $16 \cdot 899/21 = 685$ nonfoundational vulnerabilities. We have 255 actual foundational and 565 actual nonfoundational vulnerabilities. A χ^2 test on this data rejects this null hypothesis ($p = 1.3 \cdot 10^{-3}$).

We can say that the vulnerabilities are not equally distributed, yet we cannot conclude that foundational vulnerabilities are the majority of vulnerabilities as argued by Ozment and Schechter, because they are not. They are actually less than a third of the total number of vulnerabilities. This is even more striking when compared with the actual fraction of the codebase that is still foundational.

What about a weaker claim? maybe most vulnerabilities are inherited (but perhaps not necessarily foundational)? Using the same logic as above, we should now see $6 \cdot 899/21 = 257$ local and $15 \cdot 899/21 = 642$ inherited vulnerabilities; the actual counts are 426 and 473, respectively. This is strongly rejected ($p < 10^{-6}$).

Inherited vulnerabilities are far less than they should be, under the assumption of uniform distribution. These seem to show that Mozilla Firefox developers are doing a good job of vetting out vulnerabilities. In contrast, while we find that foundational vulnerabilities are more than they should be, we find no evidence that they are the majority.

We believe that the difference between our results and [3] can be explained by two factors. As we mentioned, Ozment and Schechter use a slightly different definition of "foundational": for them, a vulnerability is foundational if it was in the code when they started their study. For us, a vulnerability is foundational when it is inherited from version v1.0. However, they start their study at version v2.3, after a significant amount of development has already happened. Applied to our data, this would mean truncating Table 3, where we start at a later version, truncating all columns before, and adding all rows above that version.

More formally, if we had started our study at version v, where $1 < v < 6$ we would get a new matrix $m'(v)$ with $6 - v + 1$ rows and columns where

$$m'_{jk}(v) = \begin{cases} \sum_{1 \le k \le v} m_{vk} & \text{if } j = 1, \\ m_{j+v-1,k+v-1} & \text{otherwise.} \end{cases} \tag{2}$$

In this aggregated matrix, much more weight is now given to the new first row, where foundational vulnerabilities originate, and the p-values for the corresponding χ^2 test get less and less. This is therefore grounds to suspect that finer resolution (more intermediate versions) and truncation (not going back to v1.0, but to some later version) would in our study produce exactly the sort of foundational vulnerabilities that Ozment and Schechter found. In other words, the foundational vulnerabilities in their study might well be an artifact of the study design, which might disappear had more accurate data been available for releases earlier then 2.3.[8]

Other than that, when we repeat our tests with the data presented in their study (their Table 1 in [3]), we get high confidence that vulnerabilities are inherited there also, and also that they are not foundational under the uniform distribution assumption ($p < 10^{-6}$).

6 The Slow Pace of Software Evolution

While Fig. 2 seems to give a quick turn-around of versions, the actual picture in term of the code-base is substantially different. As Fig. 5 shows, the pace of evolution is relatively slow. Every version is largely composed by code inherited from old versions and the fraction of new code is a relatively small one.

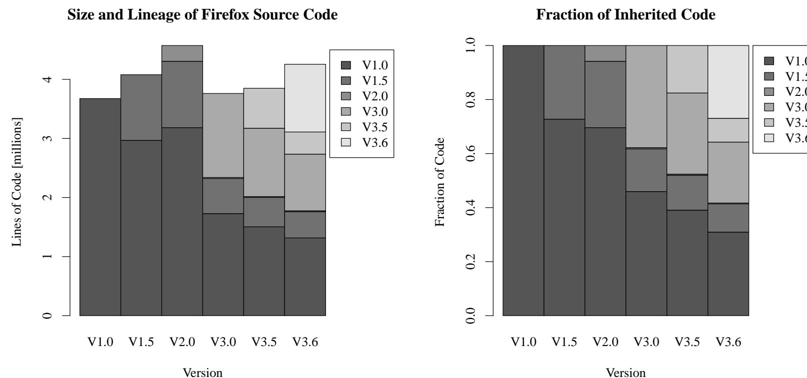
Looking at this picture it is somewhat a surprise that our statistical tests reject the idea that foundational and inherited vulnerabilities are not the majority. At the same time the presence of after-life vulnerabilities, and even the frequency of zero-day attacks is no longer a big surprise.

The large fraction of code re-use reconciliates the seemingly contradictory information that vulnerability discovery is a hard process with the existence of zero-day attacks: when a new release gets out, the 40% of old code has been already targeted for over 6 months. Therefore the zero-day attack of version v could well be in reality a six month mounting attack on version $v - 1$.

Observing this phenomenon from a different angle, if we assume that vulnerabilities are approximately uniformly distributed, then it is clear that most surviving vulnerabilities of version $v - 1$ will also be present in version v.

There is a curious dip in the lines of code development for code inherited from v1.0. For v1.5, the number goes down, as expected, but for v2.0, it goes up again. So far, we do not have a good explanation of this phenomenon. A preliminary explanation is that it might be due to the way in which our algorithm calculates versions for merged branches.

[8] They did of course not choose release 2.3 arbitrarily, but rather because it was the first release where vulnerability data was consistently and reliably documented. Therefore, they had no real choice for their initial version.



Evolution of Firefox codebase in absolute terms (left) and fraction of codebase (right). The left diagram shows the size of the Firefox releases, and the right diagram normalises all columns to a common height.

Fig. 5. Size and lineage of Firefox source code

7 Threats to Validity

Errors in NVD. we determine the Firefox version from which a vulnerability originated by looking at the earliest Firefox version to which the vulnerability applies, and we take that information from the "vulnerable versions" list in the NVD entry. If these entries are not reliable, we may have bias in our analysis. We have manually confirmed accuracy for few NVD entries, and an automatic large-scale calibration is part of our future work.

Bias in CVS. We only look at those vulnerabilities for which we can find fixes in the CVS. Fixes are identified by their commit messages, which must contain the Bugzilla identifier; therefore, fixes that do not have that form will escape us. This might introduce bias in our fixes (See [6]).

Bias in data collection. In addition to threats on parsing data collected from MFSA and CVS as described in previous work [5], we also have to extract lifetime of each source line. The extraction might bias our analysis if the extraction tool contains bug. We also apply the same strategy discussed in [5] to mitigate this issue.

Ignoring the severity. We deliberately ignore the severity of vulnerabilities in our study. Because current severity scoring system, Common Vulnerability Scoring System (CVSS), adopted by NVD and other ones (*e.g.*, qualitative assessment such such as critical, moderate used in Microsoft Security Bulletin) have shown their limitation. In accordance to [7], these systems are "inherently ad-hoc in nature and lacking of documentation for some magic numbers in use". Moreover, in that work, Bozorgi *et al.* showed that there is no correlation between severity and exploitability of vulnerabilities.

Generality. The combination of multi-vendor databases (*e.g.*, NVD, `Bugtraq`) and software vendor's databases (*e.g.*, MFSA, Bugzilla) only works for products for which the vendor maintains a vulnerability database and is willing

to publish it. Also, the source control log mining approach only works if the vendor grant community access to the source control, and developers commit changes that fix vulnerabilities in a consistent, meaningful fashion *i.e.*, independent vulnerabilities are fixed in different commits, each associated with a message that refers to a vulnerability identifier. These constraints eventually limit the application of the proposed approach.
Another facet of generality, which is also our limitation, is that whether our findings are valid to other browsers, or other categories of software such as operating system? We plan to overcome this limitation by extending our study to different software in future.

8 Related Work

Fact Finding papers describe the state of practice in the field [8–10]. They provide data and aggregate statistics but not models for prediction. Some research questions picked from prior studies are *"What is the median lifetime of a vulnerability?"*[9], *"Are reporting rate declining?"* [9, 10].

Modeling papers propose mathematical models for vulnerabilities properties [11–14, 10]. Here researchers provide mathematical descriptions of the vulnerability evolution such a thermodynamic model [14], or a logistics model [12]. Good papers in the group will provide experimental evidences that support the model, *e.g.*, [11–13]. Studies on this topic aim to increase the goodness-of-fit of their models *i.e.*, answer the question *"How well does our model fit the fact?"*.

Prediction papers try to predict defected/vulnerable component [15–19, 8, 20–23]. The main concern of these papers is to find a metric or a set of metrics that correlate with vulnerabilities in order to predict vulnerable components.

Our paper can be classified in the fact finding group and the novelty of our approach and our findings is mostly due to our ability to dig into a database integrating multiple sources (albeit specialized to Mozilla Firefox).

9 Discussion and Conclusions

First, for the *individual*, we have the obvious consequence that running after-life software exposes the user to significant risk, which should therefore be avoided. Also, we seem to discover vulnerabilities late, and this, together with low software evolution speeds, means that we will have to live with vulnerable software and exploits and will have to solve the problem on another level. Vendors shipping patches faster will not solve the problem.

Second, for the *software ecosystem*, the finding that there are still significant numbers of people using after-life versions of Firefox means that old attacks will continue to work. This means that the penetrate-and-patch model of software security does not work, and that systemic measures, such as multi-layer defenses, need to be considered to mitigate the problem.

These phenomena revel that the problem of inherent vulnerabilities is merely a small part of the problem, and that the lack of maintenance of older versions

leave software (Firefox) widely open to attacks. Security patch is not made available because it is not being deployed and because users are slow at moving to newer version of software.

In terms of understanding the interplay of the evolution of vulnerabilities with the evolution of software we think that the jury is still out. As we mentioned, foundational vulnerabilities are significantly more than they should be but are less than a third of the total number of vulnerabilities. On the other side inherited vulnerabilities are almost half of the existing vulnerabilities but are significantly less than they should be according a uniform distribution model.

So we cannot in any way affirm that most vulnerabilities are due to foundational or anyhow past errors. We need to refine these findings by a careful analysis of the fraction of the codebase for each version.

These results have been possible by looking at vulnerabilities in a different way. Other studies have studied a vulnerability's *past*, i.e., once a vulnerability is known, we look at where it was introduced, who introduced it etc. In our study, we look at a vulnerability's *future*, i.e., we look at what happens to a vulnerability after it is introduced, and find that it survives in after-live versions even when it is fixed in the current release.

We plan to look also at fixes. In particular, CVS commit messages of the form "Fixes bug n", "#n", "Bug n", "bug=n", or related forms, where n is a Bugzilla identifier make it possible to find the file(s) in which the vulnerability existed, the code that fixed it, and when it was fixed. This information is already present in our integrated database [5] and will be the subject of future studies.

We also plan to look at other software (either open-source or commercial products) to see whether our finding is applicable to other software. And therefore, should it have any significant influence on practise and give advice on how to move towards more secure software.

References

1. Howard, M., Lipner, S.: The Security Development Lifecycle. Secure software development. Microsoft Press (May 2006)
2. McGraw, G., Chess, B., Migues, S.: Building Security In Maturity Model v 1.5 (Europe Edition). Fortify, Inc., and Cigital, Inc. (2009)
3. Ozment, A., Schechter, S.E.: Milk or wine: Does software security improve with age? In: Proceedings of the 15th Usenix Security Symposium, Berkeley, CA, USA, USENIX Association, USENIX Association (August 2006)
4. Hethcote, H.W.: The mathematics of infectious diseases. SIAM Review **42**(4) (2000) 599–653
5. Massacci, F., Nguyen, V.H.: Which is the right source for vulnerabilities studies? an empirical analysis on mozilla firefox. In: Proc. of MetriSec'10. (2010)
6. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York, NY, USA, Association for Computing Machinery, ACM Press (August 2009) 121–130

7. Bozorgi, M., Saul, L.K., Savage, S., Voelker, G.M.: Beyond heuristics: Learning to classify vulnerabilities and predict exploits. (July 2010)
8. Ozment, A.: The likelihood of vulnerability rediscovery and the social utility of vulnerability hunting. In: Proceedings of 2nd Annual Workshop on Economics and Information Security (WEIS'05). (2005)
9. Ozment, A., Schechter, S.E.: Milk or wine: Does software security improve with age? In: Proceedings of the 15th Usenix Security Symposium (USENIZ'06). (2006)
10. Rescorla, E.: Is finding security holes a good idea? IEEE Security and Privacy **3**(1) (2005) 14–19
11. Alhazmi, O., Malaiya, Y., Ray, I.: Measuring, analyzing and predicting security vulnerabilities in software systems. Computers & Security **26**(3) (2007) 219–228
12. Alhazmi, O., Malaiya, Y.: Modeling the vulnerability discovery process. In: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering. (2005) 129–138
13. Alhazmi, O., Malaiya, Y.: Application of vulnerability discovery models to major operating systems. IEEE Trans. on Reliab. **57**(1) (2008) 14–22
14. Anderson, R.: Security in open versus closed systems - the dance of Boltzmann, Coase and Moore. In: Proceedings of Open Source Software: Economics, Law and Policy. (2002)
15. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early predictors of vul. Journal of Software Architecture (2010)
16. Gegick, M., Rotella, P., Williams, L.A.: Predicting attack-prone components. In: Proc. of the 2nd Internat. Conf. on Software Testing Verification and Validation (ICST'09). (2009) 181–190
17. Jiang, Y., Cuki, B., Menzies, T., Bartlow, N.: Comparing design and code metrics for software quality prediction. In: Proceedings of the 4th International Workshop on Predictor models in Software Engineering (PROMISE'08), ACM (2008) 11–18
18. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering **33**(9) (2007) 2–13
19. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Communications and Computer Security (CCS'07). (October 2007) 529–540
20. Shin, Y., Williams, L.: An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08). (2008)
21. Shin, Y., Williams, L.: Is complexity really the enemy of software security? In: Proceedings of the 4th Workshop on Quality of Protection (QoP'08). (2008) 47–50
22. Zimmermann, T., Nagappan, N.: Predicting defects with program dependencies. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09). (2009)
23. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of the 3th International Workshop on Predictor models in Software Engineering (PROMISE'07), IEEE Computer Society (2007)