Firefox: Security Vulnerability Analysis

Balachander Padmanabha*, Devyani Kulkarni†, Vishwesh Rege‡
*bpadmana@ucsd.edu, †dakulkar@ucsd.edu, ‡vrege@ucsd.edu

***Abstract*—Firefox is a popular browser used by millions of users and its security has been an active research topic over the years. Through this study we wish to explore the influence of legacy code and software development practices on the overall security of the browser. We primarily rely on CVE reports and Bugzilla entries for our dataset spanning 49 versions and over 11 years.**

We found that the total number of security vulnerabilities reported have increased as the product matured, but there has been a declining trend in the percentage of these vulnerabilities that are foundational. However, we could not draw a strong relationship between time and foundational vulnerabilities. We also found via analysis that there has been a decline in the mean lifetime and the patch duration of vulnerabilities after Firefox shifted to the rapid release model in 2011. This indicates that there has been greater attention to security after Firefox shifted to rapid releases of 6 weeks from its traditional model spanning approximately 1 year. We were also able to identify certain components of Firefox which are more prone to vulnerabilities than others, with JavaScript Engine being the most vulnerable component. Lastly, we built a classifier which could predict the CVSS scores of vulnerabilities based on certain attributes in the CVE entries with an accuracy of 97%.

I. INTRODUCTION

Computer users spend most of their time on the Internet nowadays and the web browser is by far the most highly used software on both desktops and mobiles. Most online activities such as reading news, email, social networking, information search and retrieval is done through the browser. Consequently web browsers are complex and dont just display web pages but run significant amount of active code. Websites are more like applications than documents. Browsers have to deal with pages consisting of lots of JavaScript and Flash as well as full-blown "web apps" like Gmail etc. Large parts of these applications run inside the browser due to which its role has changed into more of an operating system rather than a simple document renderer. Insecure websites or browsers can easily lead to loss of user privacy, compromise of confidential data or even the underlying system. Losses due to attacks on web security were estimated at 158 Billion in 2016 [18] which makes browser security ever more important due to the web browser being the primary mode of accessing information online.

In this study, we seek to understand the nature and trends of security vulnerabilities in the open-source Mozilla Firefox browser. We chose Firefox due to its popularity, large code base, and prominence as an attack target. Furthermore, we aim to answer several questions that have been previously raised regarding the insecurity of software product and developer inattention to security. Previous studies like the 'Milk or Wine' paper [1] by Ozment et. al. have looked at quantitative evidence over a period of several years to verify whether attention to security can improve a products security over time. In [1] they seek to understand whether efforts by the OpenBSD development team to secure their product decreased the rate at which vulnerabilities were reported. We propose a similar approach on Firefox. Firefox has some unique features about its development process which lead to interesting scenarios for analysis. In 2011, Mozilla switched to a rapid release cycle where new releases are planned to occur at six-week intervals in order to get new features to users faster. We explore the impact of this accelerated release cycle as well as the trade-offs in terms of vulnerability to attacks and security compatibility with add-ons.

We set out to answer the following major questions in the course of our project:

1) Does Firefox browser security improve with age?
2) Does software release life cycle affect software security?
3) Are certain components and files more vulnerable compared to others?

II. METHODOLOGY

Mozilla released its first web-browser named Phoenix in September 2002, following which the product went through some changes and the Firefox version 1.0 was released on November 9, 2004. After this release, the company released its major versions approximately once every year in the names of Firefox 2, Firefox 3.5, Firefox 3.6 and Firefox 4. With the intent to push new features faster, the release model was changed to rapid release cycles of 6 weeks from Firefox 5 onwards. So far, there have been 57 official releases of the browser.

For the purpose of our study, we consider Firefox 2 as the foundational version, as the security vulnerabilities and their corresponding bugs were not methodically documented prior to this version. The last version of our study is Firefox 49 as the bug related information beyond this version is not entirely available on CVE. Moreover, we only consider the major official releases and consider all the sub releases to be under the umbrella of its major release. For instance, data for version 17.0.1 is considered to belong to the umbrella version 17.

The Firefox vulnerability dataset was created using the following sources:

1) We studied a total of 1438 publicly known security vulnerabilities in the CVE database from the year 2006 to 2016. Each CVE-ID is associated with a single security vulnerability. For the purpose of our study we considered the following attributes associated with each CVE entry: 1) CVSS Score, 2) Publish date of the CVE entry, 3) Referenced Bugzilla pages, 4) Firefox versions

Origin Version	End Version	No. of Vulnerabilities	Origin Version	End Version	No. of Vulnerabilities	Origin Version	End Version	No. of Vulnerabilities
2	2	34	3.6	6	1	27	28	1
2	3	123	3.6	7	7	28	29	3
2	4	172	3.6	11	4	29	30	10
2	5	10	4	5	7	30	31	14
2	6	1	4	6	9	30	32	8
2	7	1	4	7	3	30	33	12
2	8	5	4	8	2	32	33	1
2	9	1	4	9	4	33	34	11
2	10	7	4	10	10	34	35	10
2	13	1	4	12	12	36	37	18
2	15	32	4	13	9	37	38	18
2	16	25	4	14	19	38	39	19
2	17	35	6	7	1	39	40	20
2	18	27	7	8	2	40	41	30
2	19	14	8	9	2	41	42	24
2	20	1	10	11	1	41	43	1
2	26	13	12	13	2	42	43	21
2	27	15	16	17	1	43	44	18
2	28	19	19	20	13	44	45	42
2	29	36	19	21	14	45	46	13
3	4	62	19	22	17	46	47	14
3.5	4	49	19	23	15	47	48	24
3.5	5	4	19	24	19	48	49	18
3.6	4	26	19	25	14			
3.6	5	1	19	26	1			

Fig. 1. Number of vulnerabilities between pairs of origin version and end version

affected by the vulnerability, 5) Type of vulnerability, 6) Confidentially impact, 7) Access strategy

To associate a vulnerability with the version in which it was *introduced*, we used the *Firefox versions affected by the vulnerability* data and selected the umbrella version of the earliest version for our purpose. We also tried to verify this information by manually inspecting every file affected in the patch fix for the vulnerability and associating it with the commit-id in which it was introduced using the mercurial CLI on the release branch of Firefox.

We chose 30 vulnerabilities at random from the CVE database, out of which the foundational version of 21 vulnerabilities in the CVE entry matched with that of the foundational version obtained using manual code inspection. For the remaining 9 vulnerabilities there was some discrepancy in our findings, which may be due to the fact that the lines of code that we checked for were not the root cause of the vulnerability. Our approach matched with that of [5] and we decided to obtain the foundational version from the CVE entry.

The version in which a vulnerability *died* was considered to be the version immediately after the last entry in the *Firefox versions affected by the vulnerability* data. This information was verified to be correct for all the 30 vulnerabilities that we manually inspected. Figure 1 shows the number of vulnerabilities between pairs of origin version and end version, which corresponds to the version in which a vulnerability was introduced and the version in which it died respectively. There were no vulnerabilities corresponding to version pairs not shown in the figure.

To obtain the date on which a vulnerability was born/introduced, we considered the release date of the version in which it was introduced. The reported date of a vulnerability was taken to be the minimum of the published date of the vulnerability on CVE and the reported dates of all the bugs associated with that CVE entry on Bugzilla. The date on which a vulnerability was fixed was the maximum of all the dates on which the patches were released for all the bugs for the CVE report.

2) For the analysis related to code size, we primarily referred to the OpenHub dataset [10] which has a record of the number of lines of code in each version. We validated this data by taking diffs of the commits corresponding to consecutive releases in the Firefox code-base. We thus obtained the number of lines added/removed between versions. This was found to be approximately equal in magnitude to the data obtained from [10].
3) To analyze the vulnerable components and files in the code-base we used a combination of CVE data, associated Bugzilla bugs and the OpenHub website to get file and component related information. This was collated over the entire dataset and we sorted the file and component frequencies to get the most frequently changed files and affected Firefox components.
4) To predict CVSS score for a given CVE entry, we split the randomized CVE dataset into train and test in a ratio of 30:70 and trained supervised learning models with the feature vector containing type of vulnerability, access level gained by attacker, access strategy, complexity of attack, confidentially and integrity impact.

III. Analysis

A. Are vulnerability reporting rates declining?

We analyzed the reporting trend of foundational vulnerabilities over the years from 2006 to 2016. Figure 2 depicts these trends in comparison with the total number of vulnerabilities reported each year.

We observe that there is a decrease in the number of foundational vulnerabilities reported over the years, except for the years 2006 and 2012. The considerable increase in reported vulnerabilities in 2012 correlates with the increase in number of contributors to Firefox [12].

The percentage of foundational vulnerabilities gives a better picture of this trend in Figure 2. Thus, we can say that the total number of vulnerabilities have increased over the years

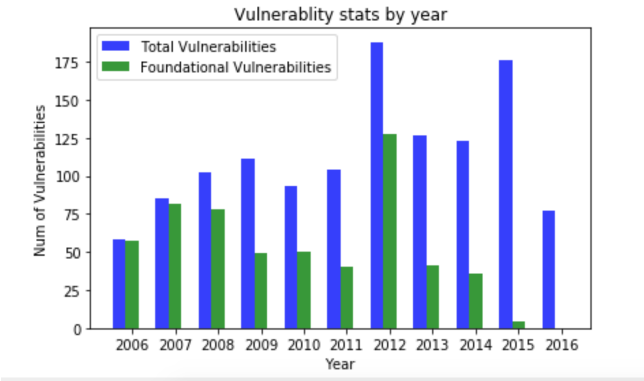

Fig. 2. Vulnerability Distribution by Year

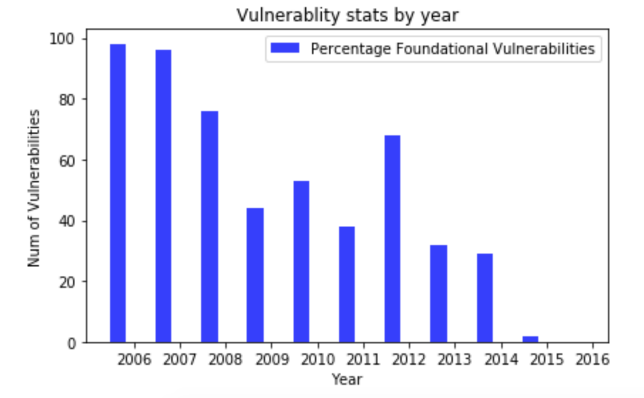

Fig. 3. Percentage of Foundational Vulnerability by Year

and there is a decreasing trend in the number of foundational vulnerabilities over the years. Also, the number of foundational vulnerabilities relative to the total number of vulnerabilities seem to be reducing over time.

B. Do larger code changes cause more vulnerabilities?

In Figure 4, the plot illustrates the relationship between number of vulnerabilities reported to the lines of code introduced across different versions. One anomaly observed is for version 2 which had about 2.97 million lines of code and was our foundational version for the study. The Pearson correlation coefficient for the data is 0.0. Thus, there is no relationship between the altered lines of code and number of vulnerabilities in Firefox.

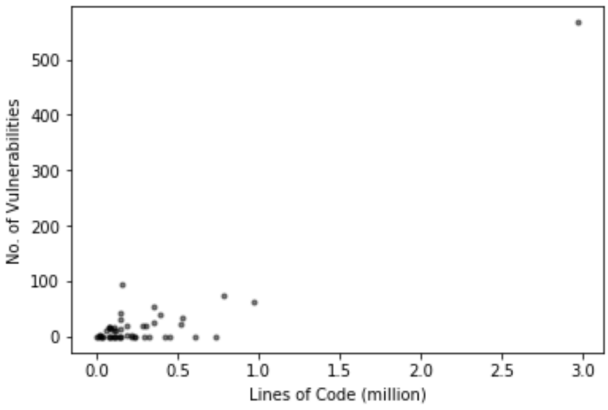

Fig. 4. Lines of Code Analysis

C. Do today's coders introduce fewer vulnerabilities per line of code?

The vulnerability density of a version is the ratio of number of vulnerabilities to lines of code (MLOC) as illustrated in figure 5. Version 19 (released in 2012) stands out for having the highest vulnerability density of 581.25 with 93 vulnerabilities and 0.16 million lines of code. The large ratio of reported vulnerabilities per line of code in this version correlates to the increase in number of contributors in 2012 [12]. This invariantly increased number of vulnerabilities, thus increasing the density. Overall, the trend for vulnerability density in figure 5 doesnt give conclusive proof regarding if todays coders are introducing fewer vulnerabilities per line of code.

Furthermore, we calculated vulnerability density per thousand lines of code. The density of all reported vulnerabilities ranged from 0 - 0.58125 and averaged at 0.0757. As stated in [1], software engineers estimate the defect density of well-written code to be 3-6 bugs per thousand lines of code. We observe that the defect density for Firefox is one order of magnitude lesser than this estimated value.

D. What is the mean lifetime of a vulnerability?

The OpenBSD "Milk or Wine" study [1] calculates the median lifetime of a vulnerability. However, since many vulnerabilities in Firefox have been reported in the same version in which they were born, we consider mean lifetime of a vulnerability as a better metric. Here, lifetime of a vulnerability is defined as the number of days between the date on which the vulnerability was reported and the release date of the version in which it was born. For vulnerabilities which were reported in the same version in which they were born, most likely in the nightly or beta releases of that version, the lifetime is considered to be 0 days. Figure 6 shows the mean vulnerability lifetime of the different versions of Firefox. For versions in which there were no reported vulnerabilities, the mean lifetime is plotted as 0.

Figure 6, clearly depicts that the lifetime of vulnerabilities was considerably higher before the rapid release cycle kicked in from version 5 onwards. The mean of the mean lifetime of a vulnerability for traditional releases is 676.6 days, whereas it is 3.6 days for the rapid releases. This could be because vulnerabilities were discovered much faster in versions with rapid releases due to frequent changes in code. It is also possible that the vulnerabilities were reported faster after Mozilla increased the bounty for vulnerability reporting to up to $3000 in 2010[16].

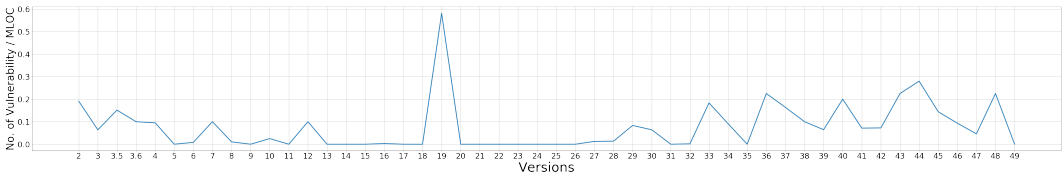

Fig. 5. Vulnerability Density Analysis

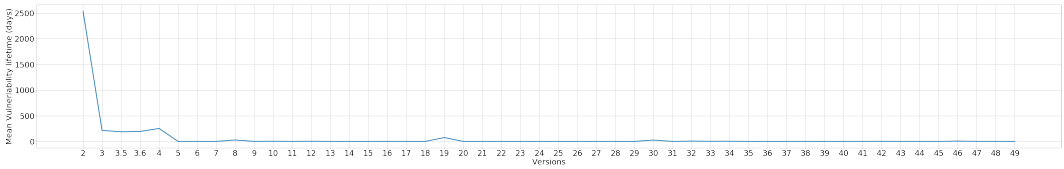

Fig. 6. Mean Vulnerability Lifetime

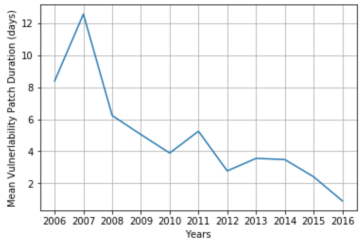

Fig. 7. Patch Duration Analysis over the years

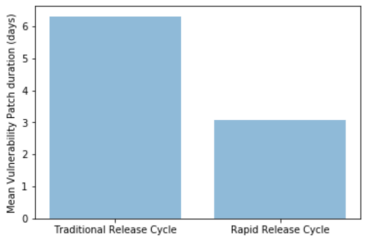

Fig. 8. Mean Patch Duration for traditional and rapid releases

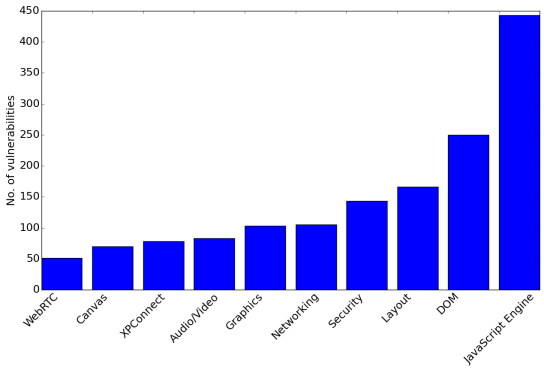

Fig. 9. Top 10 Vulnerable Components.

E. How fast are vulnerabilities fixed after being reported?

We analyzed the patch speed for the Firefox browser over the years as shown in Figure 7. The patch duration for a vulnerability is defined as the difference between the latest date on which the fix for the vulnerability was released and the earliest date on which the vulnerability was reported. We observe that the mean patch duration has declined over the years, indicating that the attention to fixing security vulnerabilities has increased over the years.

We computed the mean patch duration for vulnerabilities for the traditional releases and the rapid releases in Figure 8. We see that the patch duration for the rapid releases is approximately half when compared to the patch duration of traditional releases. This confirms the fact that rapid releases ensures faster fixes. We can also infer that the overall code is changing at a fast pace which requires the attackers to keep rethinking their strategy and tools, i.e. the adversarys ability to discover security defects is dominated less by the code quality in terms of security and more by the time required to familiarize themselves with it [4]. It is representative that the Firefox rapid-release cycles exposes the software to a shorter window of vulnerability thus providing dynamic security to possible attacks.

F. Which files and components are most vulnerable?

We analyze the frequently affected components and files in Firefox. We found that the files related to automated test suite like crashtests.list in different directories are very frequently modified while fixing vulnerabilities. Table 1 lists the most sensitive files. In Figure 8 we plot the top 10 vulnerable components since 2006, and in Figure 9 we plot the frequency of occurrence of these components over the years.

The graphs are very insightful in understanding the parts of the browser that are more vulnerable or prone to attack. According to a report by the security vendor Cenzic in March 2012, cross-site scripting is one of the top security vulnerabilities accounting for 37% of all attacks and is also one of the most common JavaScript and DOM based security vulnerabilities. This is validated in our analysis of most

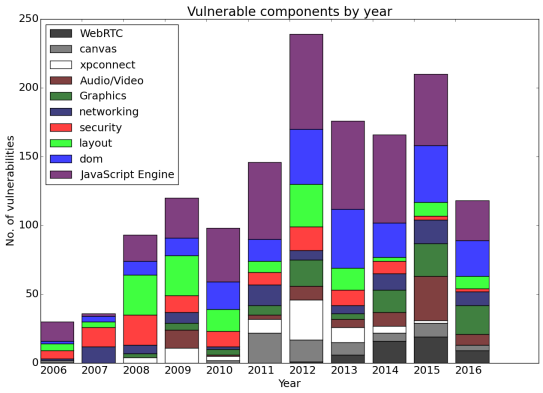

Fig. 10. Vulnerable Components by year

File name	Change frequency
content/base/src/nsContentUtils.cpp	28
content/canvas/src/WebGLContext.h	21
dom/base/nsGlobalWindow.h	16
js/src/vm/Stack.cpp	14
dom/canvas/CanvasRenderingContext2D.cpp	13
docshell/base/nsDocShell.cpp	13

TABLE I
MOST VULNERABLE FILES IN FIREFOX

vulnerable components in Firefox where we find that most security bugs are found in the JavaScript Engine followed by the DOM.

Further analysis of these components led us to the conclusion that most of the vulnerabilities are inherently tied to rendering and execution of JavaScript code. For example, all bugs marked under the umbrella of 'Canvas', 'Graphics', and 'Layout' are mainly vulnerabilities in the Rendering & Layout Engine (Gecko) and WebGL (a library for rendering interactive 3D and 2D graphics within web browsers). Manual analysis of several bug descriptions revealed that majority of these cases are due to possible Denial of Service (DoS) attacks caused by crashes due to improper memory management and quite a few remote code execution attacks due to buffer overflows.

DOM basically refers to the methods for allowing programmatic access to the web document 'tree' through programming languages such as JavaScript, and WebRTC is a JavaScript based communication protocol. XPCOM is an API for developing add-ons while XPConnect is a bridge between JavaScript and XPCOM. However XPCOM is currently being deprecated with a shift to the safer WebExtensions API [14]. This can also be observed in the decline in the number of XPConnect related vulnerabilities since 2015. The increase in the number of XPConnect related security bugs in 2011 and 2012 may be attributed to the transition to the rapid release cycle which is known to have significantly affected add-on compatibility [15].

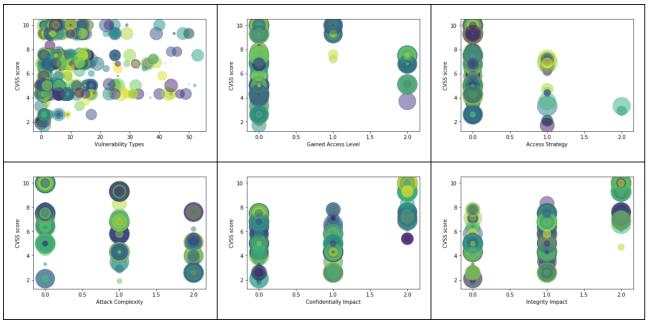

Fig. 11. CVSS score vs Parameter distribution

G. Can we predict the CVSS score of a CVE Entry?

The Common Vulnerability Scoring System (CVSS) [17] is an open framework for communicating the characteristics and impacts of IT vulnerabilities and a standard measurement system for industries, organizations, and governments that need accurate and consistent vulnerability impact scores. Its quantitative model ensures repeatable accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the scores. In this section we intend to predict the CVSS score using parameters associated with the CVE reports. This provides insight into which parameters influence the score thus influencing how the community prioritizes the fix for the vulnerability and how its severity is categorized internally in organizations.

In cases where the vendor has not released all the relevant details for a vulnerability, NVD issues a 10.0 score for the CVE entry which could skew our data and prediction capability.

Figure 8 illustrates the relationship of individual parameters (vulnerability types, gained access level, access strategy, confidentiality impact, attack strategy and integrity impact) with the CVSS score using different colors for each possible value of that parameter. When we examine the plots we observe strong statistical correlation between certain types of vulnerabilities (like Execution of Code, Buffer Overflow, Memory corruption, Denial of Service), gained access level of attack (like admin, user), access strategy (like Remote attacker) and confidentiality impact (like complete, partial) with relatively high CVSS scores. On the other hand, CVE entries with complex attack strategies and partial integrity impact contributed to lower scores. We performed supervised learning using the data and obtained accuracy around 88% for Linear Support Vector Classifier and 97% for Decision Tree classifier. This result provides a valid model to predict future CVSS scores based on various parameters in an automated and unbiased fashion.

IV. RELATED WORK

Since Firefox shifted to its rapid release model of software development, a number of studies have been conducted with regards to its effect on security [3], [4] and software quality in general [2]. Experts expect that moving from large-scale, infre-

quent releases of new versions to releases with new features at much shorter, regular intervals results in favoring adding new features over writing less vulnerable code. This is because of the traditional software engineering wisdom regarding code maturity, reliability and reuse, and its correlation with security. Shorter release cycles also result in shorter testing periods. However Clark et. al. [3], [4] found that during the active life cycle of a software product, the adversary's ability to discover security defects is dominated more by the time required to familiarize with the code than by the intrinsic quality of the code itself. As such, the rapid release cycle results in the lengthening of the attacker's learning curve as attackers have to re-learn and re-adapt their tools in response to a rapidly changing codebase. This is in line with our results. The findings also suggest that software reuse can actually be harmful.

Khomh et. al.[2] investigated the effect of rapid releases on software quality by comparing the crash rates, median uptime, and the proportion of post-release bugs of the versions that had a shorter release cycle with those having a traditional release cycle. They found that with shorter release cycles, users do not in fact experience significantly more post-release bugs and bugs are actually fixed faster (but proportionally less bugs are fixed compared to the traditional release model). However, users do experience these bugs earlier during software execution (the program crashes earlier).

Baysal et. al. [7] compare the initial long version cycles of Firefox and short version cycles of Chrome and examine the relationship between development cycles with popularity and reliability. Chromes popularity has a direct correlation with the number of defects found, which may mean more bugs are likely to be reported as more people adopt a software system. On the other hand, theres no direct correlation between Firefox's popularity and number of defects. Thus, one may also infer that rapid releases result in increased adoption but at the cost of software reliability, which is in contrast to the findings of Clark.

Zaman et. al [8] compare security bugs with performance bugs in Firefox and find that security bugs are fixed and triaged much faster, involve more developers and impact more files in a project, but are also reopened and tossed more frequently.

V. CONCLUSIONS

From our analysis we found that 565 out of the total 1438 vulnerabilities were introduced in or before Firefox 2. Thus, a large portion of the vulnerabilities are foundational. But we did not find a strong correlation of these foundational vulnerabilities with age of the product. This could be due to factors such as:

1) Higher scrutiny of the product after 2010/2011 due to, increase in contributors, change in code base every 6 weeks due to rapid release model, and increase in bug bounty for finding vulnerabilities
2) Incorrect information in the CVE database.

However, we found that the attention to security of Firefox has improved considerably after the shift to rapid releases. This is evident from the fact that the vulnerabilities are fixed twice as fast in rapid release model as compared to that in traditional releases. Also the mean life of vulnerabilities in the rapid release model is 3.6 days as opposed to that of 676.6 days in the traditional model. We also found that certain components and files in Firefox are more vulnerable than others. JavaScript and Rendering components are most vulnerable and contribute to the major chunk of risks associated with the browser. We also trained a decision tree classifier which predicts the CVSS score of a vulnerability with an accuracy of 97%. Source code for the project is available at [9].

REFERENCES

[1] Ozment, Andy, and Stuart E. Schechter. "Milk or wine: does software security improve with age?." USENIX Security Symposium. 2006.
[2] Khomh, Foutse, et al. "Do faster releases improve software quality?: an empirical case study of Mozilla Firefox." Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. IEEE Press, 2012.
[3] Clark, Sandy, et al. "Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities." Proceedings of the 26th annual computer security applications conference. ACM, 2010.
[4] Clark, Sandy, et al. "Moving targets: Security and rapid-release in firefox." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.
[5] Massacci, Fabio, Stephan Neuhaus, and Viet Hung Nguyen. "After-Life Vulnerabilities: A Study on Firefox Evolution, Its Vulnerabilities, and Fixes." ESSoS. 2011.
[6] Nguyen, Viet Hung, and Fabio Massacci. "The (un) reliability of NVD vulnerable versions data: An empirical experiment on Google Chrome vulnerabilities." Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM, 2013.
[7] Baysal, Olga, Ian Davis, and Michael W. Godfrey. "A tale of two browsers." Proceedings of the 8th Working Conference on Mining Software Repositories. ACM, 2011.
[8] Zaman, Shahed, Bram Adams, and Ahmed E. Hassan. "Security versus performance bugs: a case study on firefox." Proceedings of the 8th working conference on mining software repositories. ACM, 2011.
[9] Source Code. https://github.com/bala8/CSE-227
[10] OpenHub. https://www.openhub.net/p/firefox
[11] CVE database. https://www.cvedetails.com/product/3264/Mozilla-Firefox.html
[12] https://support.mozilla.org/en-US/kpi/dashboard
[13] Version History. https://en.wikipedia.org/wiki/Firefox_version_history
[14] https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/
[15] https://blog.mozilla.org/addons/2011/04/19/add-on-compatibility-rapid-releases/
[16] https://blog.mozilla.org/security/2010/07/15/refresh-of-the-mozilla-security-bug-bounty-program/
[17] https://nvd.nist.gov/vuln-metrics
[18] https://www.forbes.com/sites/stevemorgan/2016/01/24/how-consumers-lost-158-billion-to-cyber-crime-in-the-past-year-and-what-to-do-about-it/.