Moving Targets: Security and Rapid-Release in Firefox

Sandy Clark
saender@cis.upenn.edu
University of Pennsylvania

Michael Collis
mcollis@cis.upenn.edu
University of Pennsylvania *

Matt Blaze
mab@crypto.com
University of Pennsylvania

Jonathan M. Smith
jms@cis.upenn.edu
University of Pennsylvania

ABSTRACT

Software engineering practices strongly affect the security of the code produced. The increasingly popular Rapid Release Cycle (RRC) development methodology and easy network software distribution have enabled rapid feature introduction. RRC's defining characteristic of frequent software revisions would seem to conflict with traditional software engineering wisdom regarding code maturity, reliability and reuse, as well as security.

Our investigation of the consequences of rapid release comprises a quantitative, data-driven study of the impact of rapid-release methodology on the security of the Mozilla Firefox browser. We correlate reported vulnerabilities in multiple rapid release versions of Firefox code against those in corresponding extended release versions of the same system; using a common software base with different release cycles eliminates many causes other than RRC for the observables. Surprisingly, the resulting data show that Firefox RRC does not result in higher vulnerability rates and, further, that it is exactly the unfamiliar, newly released software (the "moving targets") that requires time to exploit. These provocative results suggest that a rethinking of the consequences of software engineering practices for security may be warranted.

1. INTRODUCTION

The root cause of many of today's computer and network security threats is errors in software. As software is an engineered artifact, the discipline of software engineering has emerged to model and manage such factors as cost [9] and time [11] estimates, feature selection [21] and code maturity [8]. A maturity model based on developer bug fix rates [40, 41] might be used in combination with other factors to determine release readiness, with the goal of shipping bug-free software systems. When the software ships with *bugs*, some may be exploitable *vulnerabilities*, and a subset of exploitable vulnerabilities will be discovered and further engineered into *exploits* which are then used, sold or saved for later use.

Mainstream software engineering practice has development models intended to produce secure systems. Examples include Process Improvement Models, used in the ISO/IEC 21827 Secure Systems Engineering-Capability Maturity Model SSE-CMM [30], originated by the U.S. National Security Agency, but now an international standard), Microsoft's Secure Development Lifecycle (SDL) [28], Oracle's Software Security Assurance Process [26] and the Comprehensive, Lightweight Application Security Process (CLASP) [50]. The goal [37] of these models is:

> "*To design, build, and deploy secure applications, [...] integrate security into your application development life cycle and adapt your current software engineering practices and methodologies to include specific security-related activities*".

In contrast, *Agile* approaches to software development such as Extreme Programming (XP) [16], Adaptive Software Development (ASD) [27], and Feature Driven Development (FDD) [14] are primarily intended to ensure customer satisfaction via rapid feature delivery [5] rather than to produce secure code [7]. The U.S. Department of Homeland Security [44] assessed each of the 14 core principles of the Agile Manifesto [5] and found 6 to have negative implications for security, with only 2 having possible positive implications. Attempts to reconcile security with Agile development [48, 51, 32] have noted that many of the practices recommended for security undermine the rapid iterations espoused by the Agile Manifesto [5] (see Section 4.1).

Security experts chastise software developers [36] for favoring adding new features over writing less vulnerable code. However, the survival of a product in competitive software markets *requires* frequent introduction of new features particularly for user-facing software systems such as web browsers embroiled in features arms races.

As a consequence, two major web browser developers, Google (Chrome) and Mozilla (Firefox), have overhauled their development lifecycle, moving from large-scale, infrequent releases of new versions with later patches as needed, to releases with new features at much shorter, regular intervals. Microsoft is moving Windows development to a RRC [19].

*Now at Google; work done while at Penn

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
ACM 978-1-4503-2957-6/14/11.
http://dx.doi.org/10.1145/2660267.2660320.

New releases of Chrome and Firefox versions occur every six weeks. The primary intent of each RRC iteration is to get new features to users as rapidly as possible [42, 34]; RRCs may also include bug fixes in the release.

This change in software development and release strategy raises three security research questions that we address in detail in Section 3:

1. Does a switch to Agile RRC development introduce large numbers of new vulnerabilities into software, given that the focus is on new features, rather than on improving existing code?
2. Where in the code base are vulnerabilities being discovered? (*i.e.*, are they in code written prior to the switch to RRC, in code introduced in previous iterations of RRC or in code added in the current version?)
3. Are vulnerabilities being discovered more quickly since the switch to RRC?

Contributions: Our main contributions are:

1. A new dataset of Firefox vulnerabilities constructed by correlating a number of publicly available data sources;
2. Quantitative evidence that:
   - The rate of vulnerability disclosure has not increased substantially since the start of Firefox RRC
   - The overwhelming majority of vulnerabilities discovered and disclosed are *not* in the new code
   - Vulnerabilities originating in Firefox RRC versions are almost all not disclosed until that version has been obsoleted by newer versions
   - Firefox RRC does *not* appear to produce demonstrably more vulnerable software
3. A data-inspired observation that frequent releases of high volumes of new code, due its relative unfamiliarity to attackers, may provide some protection for frequently targeted software; and
4. Further supporting evidence for an exploit-free "honeymoon" or "grace period"[13] provided by the attacker's learning curve.

2. OUR DATASET AND METHODOLOGY

In this research, we seek to understand the dynamics of what a switch to rapid-release cycles might mean for the numbers of vulnerabilities discovered and disclosed in a hitherto traditionally developed system.

We make three assumptions, which we believe are reasonable for the purposes of our analysis: (1) that with each addition of new code, a number of new software defects are also added; (2) that (to the extent that security vulnerabilities are a consequence of software defects), that new vulnerabilities are also introduced and will be discovered and disclosed; and (3) that attackers are analyzing code bases searching for weaknesses in both old and new code.

2.1 Vulnerability Taxonomy

For the purposes of this paper, we differentiate among three types of vulnerabilities:

1. *Baseline* vulnerabilities - vulnerabilities that affect the original codebase on which RRC was based.
2. *Regressive* vulnerabilities - vulnerabilities discovered and disclosed in code *after* the version in which it was introduced has been obsoleted by a more recent version. For example, a vulnerability disclosed in version 13 that also affects versions 10, 11 and 12 would be classified as regressive.
3. *New* vulnerabilities - vulnerabilities that affect the current version of code at the time of disclosure but that do not affect previous versions.

We also differentiate between two *states* of vulnerabilities: *Active* vulnerabilities are vulnerabilities that affect a given version of software while that version is the most current available, while vulnerabilities become *Inactive* once the most recent version which it affects has been obsoleted by a more recent version, fixing the vulnerability. For example, a regressive vulnerability disclosed in version 20, but introduced in version 18, while version 20 was the most recent is said to be *active*.

Finally, there are *unknown* vulnerabilities - vulnerabilities in a given version of software that have not yet been publicly found or disclosed.

2.2 Why Firefox?

Desiderata for a system to study include: (1) open source, (2) a frequent target of attack, (3) a broad user base, and (4) a statistically significant population of publicly disclosed vulnerabilities. For such software, the new features arms race appears to conflict with the standard secure development process. To test if this is indeed the case, we sought to compare vulnerability discovery and disclosure between software developed using Agile practices and that developed using traditional methods. At the time of writing, the two most widely deployed applications that make use of RRC are Google's Chrome and Mozilla's Firefox web browsers.

The Firefox web browser proved to be an ideal system for analysis, for four primary reasons.

First, since its initial release in 2004, all Firefox source code has been open source and freely available. Pre-RRC source code is available in a CVS repository. To prepare for the switch to RRC, Mozilla moved what was then the current source 3.6.2 and 4.0a to be a foundation for the first RRC version (5.0) into a new Mercurial repository. Since then all changes for each subsequent new RRC version have been added to this repository. For the purpose of our analysis, we use Firefox version 4.0 as the 'baseline' version of all subsequent RRC versions, and cover versions 5-20. Version 20 had just been released when the data collection was complete and the analysis begun; version 23 was available by the time the analysis was complete.

Second, Firefox has a well maintained and freely available bug database, Bugzilla [38], which contains detailed information on all bugs, including patched vulnerabilities. Mozilla does not openly list the details of the most recent, unpatched security vulnerabilities in Bugzilla, but they do publish timely and somewhat detailed references to the latest security bugs on the Mozilla Foundation Security Advisory (MFSA) site [39] and the relevant details are made public in Bugzilla sometime thereafter.

It is important to note that all acknowledged bugs (defects and vulnerabilities) reported in Firefox are given a Bug ID before being assigned to be patched, so all known vulnerabilites are associated with some Bugzilla Bug ID. In addition, the MFSAs link to relevant references in the NIST

National Vulnerability Database (NVD) [43] which contains an entry for each known vulnerability, including versions affected, criticality, date released. For our study, we scraped the NVD database for all Firefox vulnerabilities, and cross-referenced each vulnerability disclosed with its corresponding MFSA to find each Bug ID issued. There is some overlap, as a single NVD (CVE [22]) entry may contain several Firefox Bugzilla Bug IDs, and a single Bug ID may link to multiple NVD (CVE) entries.

Third, since Firefox is a frequent target of attackers, Mozilla has had a 'Bug Bounty' program in place since 2004, and purchases vulnerability information from researchers. Recent research [23] on the efficacy of Bug Bounty programs suggests that 25% of Firefox's vulnerabilities are discovered through its bug bounty program. Mozilla does not announce each purchase, but Coates [15] showed that, on average, Mozilla purchases six new vulnerabilities per month. While we recognize that it is impossible to know anything about the number of private or undisclosed vulnerabilities that may have been discovered in Firefox, and that this is a potential source of error, we found the number of Bug Bounty purchases consistent with the dataset we gathered from MFSAs, Bugzilla, and the CVE database, and believe our dataset is accurate with regard to publicly available information. Also, while it may be the case that some RRC code lacks critical vulnerabilities, Firefox synchronizes the two development tracks annually, so new features added to the RRC versions become part of the next ESR version. Thus RRC does modify some of the core code base. Further, our data set is cross referenced with the MFSA [39], vulnerabilities Mozilla considers important enough to issue an advisory for. Using this standard as a measurement for severity avoids any risk of bias in our results due to a bespoke metric for severity.

Fourth, Mozilla's developers [4] acknowledged the switch to RRC "*involved changing a number of our processes. It's also raised some new issues.*". The midstream introduction of RRC gives a basis for a "before and after" comparison of security properties in light of a significant change in software development practices. Firefox has documented history using *both* development models *at the same time*. The concurrent release processes for RRC and ESR (discussed above and displayed in Table 1), effectively provide two versions of the same software differing in a single variable (the release cycle). Thus, this dual-track Firefox release strategy provides a unique analytic framework for a data-driven examination of RRC.

2.3 Firefox RRC

Mozilla began the Firefox RRC in June of 2011 with the release of version 5.0 as the first rapid release version. The Firefox release cycle is structured such that new code actually goes through three 6-week phases before being released. In between major version releases, Mozilla introduces point releases only if a critical vulnerability has been found to affect it. In practice, there are only one or two of these between each version. The code spends 6 weeks in development, 6 weeks being stabilized (called the Aurora phase) and 6 weeks being beta-tested. The code is freely available at any of these phases. Thus, at the time of release of version n, versions n+1 through n+3 are in the Beta, Aurora and development phases, respectively. This schedule allows Mozilla to release a new version regularly every 6 weeks. Prior to the inception of RRC, a version of Firefox would spend as long as a year in an *alpha* phase, and a further year in *beta*, undergoing several revisions. Meanwhile, the current release would be patched as needed.

At the start of RRC, the current stable traditionally developed code (version 3.6.2) was cloned to become the base of the new RRC code. That same code became the Extended Support Release (ESR). The ESR is intended for mass deployment in organizations. Releases are maintained for a year, with point releases containing security updates coinciding with RRC Firefox releases. A new ESR version is released (essentially) by rolling the features of the current RRC version into the new ESR version. At the time of writing, Mozilla has done this twice: for versions 10 and 17, both of which were released at the same time as the corresponding RRC versions. In our analysis, we look at RRC versions 5 through 20, but when we compare RRC to ESR, we were careful to compare only concurrent versions: RRC versions 10.0-20.0 to ESR versions 10.0-10.0.12 and 17.0-17.0.6 (See Table 1).

2.4 Data collection

We are concerned specifically with vulnerabilities disclosed in software developed and released under a 6-week Rapid Release Cycle (RRC).

From the inception of RRC up to the time of writing, 617 new Bug IDs were issued, corresponding to new vulnerabilities reported in the MFSAs [39] and CVE [22] database, providing sufficient volume of data for empirical study.

Line of code (LOC) and file counts in this paper are derived from the Mercurial repositories hosted by Mozilla and are filtered to account for a subset of filetypes that account for almost all of the code relevant to this study [2]. Specifically, we include files with the extensions: `.c`, `.C`, `.cc`, `.cpp`, `.css`, `.cxx`, `.h`, `.H`, `.hpp`, `.htm`, `.html`, `.hxx`, `.inl`, `.js`, `.jsm`, `.py`, `.s`, and `.xml`; test cases and harness code have been excluded, as well as code comments and whitespace. Our LOC counting is conservative and may understate changes to the Firefox codebase between versions.

We also look at Firefox's Extended Support Releases (ESR). These long-term support releases still follow essentially the release-and-patch cycle that preceded the transition to RRC, with the same code base as RRC versions 10 and 17 covered by our study; the next ESR release was version 24. ESR is an effective point of comparison when examining the impact of RRC on security.

2.5 Limitations

As noted earlier, *unknown* vulnerabilities exist, and this makes the date that any given vulnerability was initially discovered hard to obtain. We can only know with when the vulnerability was first *reported*. For the purposes of our analysis we use the disclosure date as an approximation for the discovery date. The disclosure date, while later than the discovery date, is workable for our purposes, since we are concerned with large-scale phenomena and inter-arrival times for vulnerability discoveries.

Notably, as Firefox is a frequent attack target and Mozilla responds quickly, by issuing inter-cycle point releases for critical and severe vulnerabilities error is as small as it can be without omniscience of undisclosed vulnerabilities attackers might have "on the shelf".

3. SECURITY PROPERTIES OF RRC

Software engineering, as exemplified by the still-vibrant *Mythical Man Month* [12], presumes new software inevitably introduces new defects and that over time, those bugs get found and subsequently either removed or repaired. Thus, software engineers have understood that software quality improves with age and that adhering to models such as those listed in Section 1 saves money [49, 17] while improving software reliability and functionality. In addition, empirical studies [29] demonstrate that the majority of software defects are discovered early in the product's lifecycle, and that while the rate of defect discovery is initially high, after the first 3 months the rate drops substantially and remains consistently low for the remainder of the product's lifetime.

Secure Software Engineering models, such as SSE-CMM [30] and Microsoft's SDL [28], also presume that heavy investment in preventing vulnerabilities early in the software lifecycle is more cost effective, and that finding and removing vulnerabilities early in the development cycle produces more secure code over its lifetime. In 2010, Aberdeen Group published research [10] confirming that the total annual cost of application security initiatives is far outweighed by the accrued benefits organizations implementing structured programs for security development and found that they realized a 4x return on their annual investments in applications security.

'Agile' programming models, on the other hand, with their focus on frequent change and rapid delivery of software, cannot spend the extensive time required to do risk analysis, threat modeling and external review [52] in the development phase. Therefore, with Chrome, Firefox and now Microsoft [19] releasing new features at a much faster rate, we might expect to see increases, both in the number of vulnerabilities and the rate at which they are discovered and disclosed. We observe that each new Firefox RRC version to date has added, on average, 290K new lines of code (LOC), and that the average LOC removed in each RRC version is 160K.

3.1 Code Bases: RRC versus ESR

Mozilla's RRC reflects the principles of Agile programming. For example, Nightingale [42] states:

> "*Rapid release advances our mission in important ways. We get features and improvements to users faster. We get new APIs and standards out to web developers faster.*"

While security patches are also included in each new release, the focus of the program is to deliver new features, not patch vulnerabilities. In contrast, ESR versions [24] are intended to remain stable and unchanged after release, except for required security patches:

> "*Maintenance of each ESR, through point releases, is limited to high-risk/high-impact security vulnerabilities and in rare cases may also include off-schedule releases that address live security vulnerabilities.*"

Table 1 lists the release dates of the RRC versions and their corresponding ESR point releases. While both RRC and ESR start from the same codebase, they soon differ substantially.

New features and new APIs mean new code, which can affect functionality and maintainability, as well as security. How many lines of new code are pushed out in Firefox's RRC? Table 1 shows the number of lines of code added, and removed, the total numbers of files changed between versions, and the total number of LOC per version since RRC was instituted. Since the start of RRC, Firefox has added a minimum of 100k LOC per version, and averages 290k LOC added, 160k LOC removed, and 3,475 files changed per version. There is a wide variance, but the median LOC added is 249k. This amounts to an average of 10% of the code-base changing in some way every 42 days.

These changes are not isolated, but rather appear to have wide-reaching effects. In a study to determine the maintainability of the Firefox codebase since RRC, Almossawi [2], found that 12% of files in Firefox are highly interconnected. Almossawi also found that making any change to a randomly selected file can, on average, directly impact eight files and indirectly impact over 1,500 files. This means that on average each new RRC version could potentially impact as many as 30,000 files.

The difference between this and the ESR versions is substantial. Only two of the point releases add more than 10k LOC and only changes to version 17.0.5 reach anywhere near the average of the RRC versions (see Table 1).

Does the modification of such large amounts of new code result in a less secure product? If so, there are three things we would expect to see:

1. An *increase* in the number of vulnerabilities affecting each new release (active vulnerabilities);
2. The *scope* of vulnerabilities should change. That is, the vulnerabilities discovered should be primarily new and should affect only the current (and possibly subsequent) versions; and
3. The regular introduction of such code should *increase* the rate of vulnerability discovery and disclosure.

In other words, if the current models for general software defects apply here, the new code should be more vulnerable than the old code.

3.2 Rapid Release and Software Quality

In this section, we address the three questions on software quality raised in Section 1.

3.2.1 Does the addition of 250K+ lines of code every 42 days markedly increase the number of vulnerabilities discovered and disclosed?

Almossawi's research [2] indicated that the defect density remains constant for releases 5-9 and then rises by a factor of two in release 12. This finding is consistent with the current defect discovery models: the new code does indeed result in more defects, but not overwhelmingly more. Certainly this means that the quality of the code is not getting worse. But what about vulnerabilities? A plot of the cumulative totals from version to version shows that the numbers of active vulnerabilities being discovered and disclosed remains fairly constant (see Figure 1) for both RRC and ESR. Looking at the ratio of total vulnerabilities between versions (see Figure 2) for RRC we see that much of the graph is nearly flat and it is only going up by less than a factor of two at its maximum. Overall, the total number of active vulnerabilities disclosed per LOC in each Firefox version since the

RRC							ESR					
Version	Release Date	LOC Added	LOC Removed	LOC Δ	Total LOC	Files Δ	Version	Release Date	LOC Added	LOC Removed	LOC Δ	Total LOC
4	-	157.4k	710k	230k	362.1k	5300	-	-	-	-	-	-
5	-	164k	161k	325k	362.4k	1700	-	-	-	-	-	-
6	-	142k	164k	306k	360.6k	2100	-	-	-	-	-	-
7	-	124k	120k	243k	361.0k	2000	-	-	-	-	-	-
8	-	109k	90k	199k	363k	1700	-	-	-	-	-	-
9	-	159k	90k	250k	368.7k	2100	-	-	-	-	-	-
10	1/31/12	491k	282k	773k	386k	4000	10	1/31/12	-	-	-	386k
10.0.1	2/10/12	-	-	-	-	-	10.0.1	2/10/12	29	7	36	386k
10.0.2	2/16/12	-	-	-	-	-	10.0.2	2/16/12	7	3	10	386k
11	3/13/12	254k	203k	457k	390.2k	2000	10.0.3	3/13/12	2,510	1,782	4,292	386k
12	4/24/12	245k	190k	436k	395k	2500	10.0.4	4/24/12	12,314	7,066	19,380	386.4k
13	6/5/12	133k	85k	218k	399.1k	2300	10.0.5	6/5/12	1,070	528	1,598	386.4k
13.0.1	6/15/12	-	-	-	-	-	-	-	-	-	-	-
14	7/17/12	265k	88k	354k	414.6k	2200	10.0.6	7/17/12	1,182	514	1,696	386.5k
14.0.1	7/17/12	-	-	-	-	-	-	-	-	-	-	-
15	8/28/12	383k	280k	664k	422.6k	9000	10.0.7	8/28/12	605	216	821	386.5k
15.0.1	9/6/12	-	-	-	-	-	-	-	-	-	-	-
16	10/9/12	608k	85k	693k	467.5k	2800	10.0.8	10/9/12	535	165	700	386.6k
16.0.1	10/11/12	-	-	-	-	-	10.0.9	10/12/12	23	10	33	386.6k
16.0.2	10/26/12	-	-	-	-	-	10.0.10	10/26/12	124	20	144	386.6k
-	-	-	-	-	-	-	10.0.11	11/20/12	1,151	316	1,467	386.7k
17	11/20/12	271k	177k	448k	475.3k	5400	17.0	11/20/12	-	-	-	475.3k
17.0.1	11/30/12	-	-	-	-	-	17.0.1	11/30/12	126	27	153	475.4k
-	-	-	-	-	-	-	10.0.12	1/8/13	2,585	260	2,845	386.8k
18	1/8/13	820k	385k	120.4k	512k	7700	17.0.2	1/8/13	2,092	1,076	3,168	475.4k
18.0.1	1/18/13	-	-	-	-	-	-	-	-	-	-	-
18.0.2	2/5/13	-	-	-	-	-	-	-	-	-	-	-
19	2/19/13	193k	146k	339k	515.6k	3700	17.0.3	2/19/13	1,204	440	1,644	475.5k
19.0.1	2/27/13	-	-	-	-	-	-	-	-	-	-	-
19.0.2	3/7/13	-	-	-	-	-	17.0.4	3/7/13	4	4	8	475.5k
20	4/2/13	252k	163k	415k	523.2k	2700	17.0.5	4/2/13	67,142	61,198	128,340	475.7k

Table 1: RRC changes from the previous version, and total LOC per version

advent of rapid release mirrors the defect discovery. Similar to defects, there is no significant jump in the number of vulnerabilities disclosed. This lack of a significant rise in the vulnerability density becomes even more apparent when we compare the number of active vulnerabilities found affecting RRC (465) and ESR (420) during the same period. It is surprising to note how close the totals are, because the magnitude of code changes in RRC is so much greater than in ESR in the same time frame.

This means that, contrary to expectations, the large volume of code added does not appear to contain more than its share of vulnerabilities.

3.2.2 *Is the scope of disclosed vulnerabilities confined to RRC?*

If the vulnerabilities found in the current RRC version result from new code added, we ought to find that most of these are *new* vulnerabilies and therefore ones that do not affect code shared with ESR versions. However, this is not the case. As stated above, during the active lifetimes of ESR versions 10 and 17, only a few new RRC vulnerabilities were disclosed, but, more importantly, if we compare the 465 total RRC vulnerabilities to the 420 in ESR, all but 45 have the same BugID affecting both RRC and ESR. In other words, the overwhelming majority of active vulnerabilities disclosed in Firefox RRC *also* affect ESR. This means that the 24 ESR point releases, which average 4,700 LOC code added per release, are affected by nearly *90%* of the vulnerabilities that affect the concurrent RRC versions *which average more than 290K LOC added per release!*

This does not mean that the new code in RRC does not contain new vulnerabilities, but rather, that 90% of the vulnerabilities disclosed in the RRC versions released during the lifetime of each ESR version must be in the older, *shared* code. As we can see in Table 2 very few vulnerabilities affecting each RRC version actually originate in those RRC versions. Of the 617 vulnerabilities disclosed in Firefox since the inception of RRC, 32 of them do not affect *any* of the RRC versions. These vulnerabilities *only* affect the baseline code originating in or before version 4, but were not found until after RRC was adopted.

The implications of this for ESR, and software engineering more generally, are substantial. The effective lifetime of the ESR versions is four times longer than for RRC. No new features are added after its initial release. It is only changed to patch critical security bugs. Yet, it is still vulnerable to 90% of the same vulnerabilities that affect the RRC versions. This raises concerns about the security of code over time, and the impact on security of code reuse.

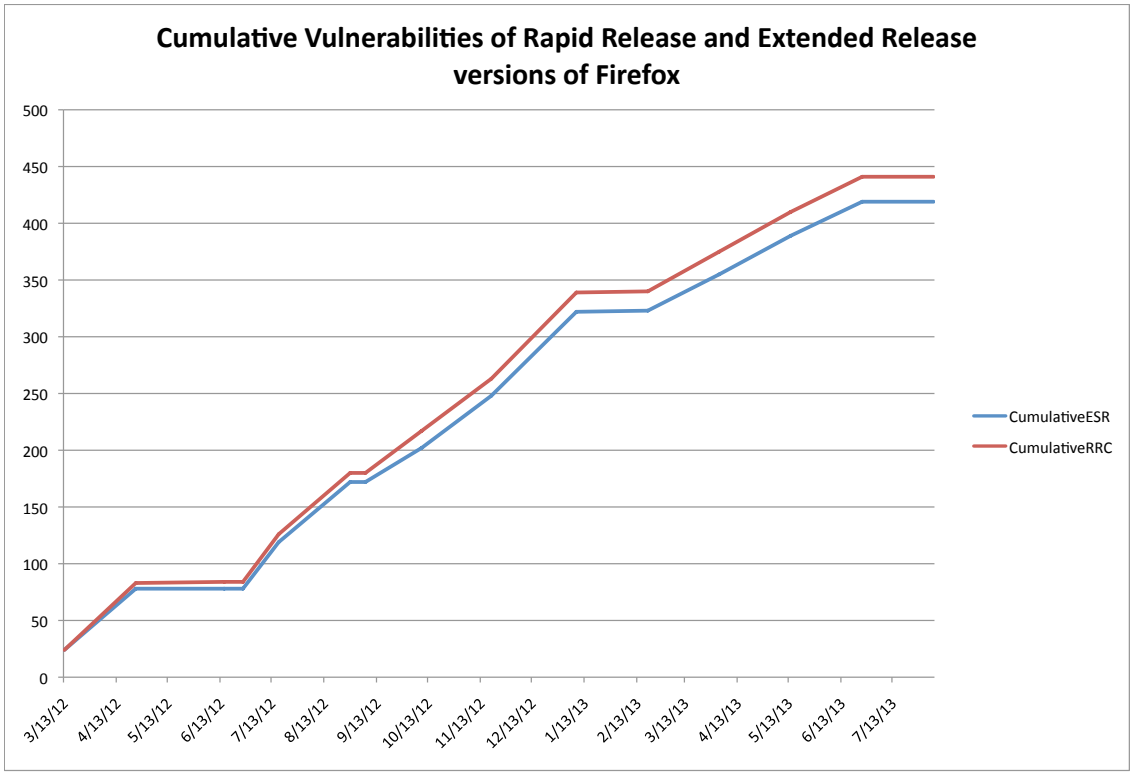

Figure 1: Cumulative total vulnerabilities affecting RRC and corresponding ESR versions during the same 6-week period

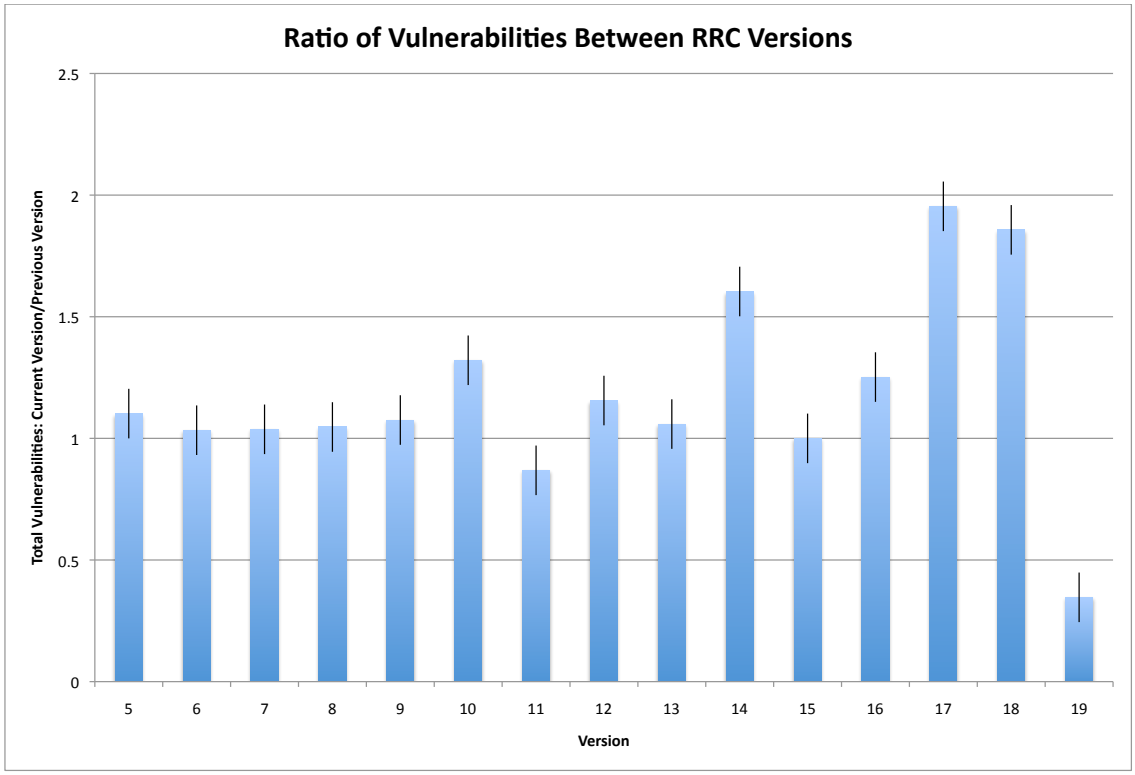

Figure 2: The ratio of vulnerabilities from version to version

Version	Total	Baseline	Regressive	New
4	476	292	184	-
5	432	261	171	-
6	418	255	149	14
7	403	246	146	11
8	385	240	144	1
9	358	240	118	-
10	271	223	47	1
11	312	223	89	-
12	270	213	55	2
13	255	213	42	-
14	159	159	-	-
15	159	159	-	-
16	127	126	-	1
17	65	65	-	-
18	35	35	-	-
19	101	1	65	35
20	65	-	65	-

Table 2: Counts of vulnerabilities by type affecting RRC (correspondence between RRC versions 10+ and ESR versions 10 and 17 is given in Table 3). Totals are not unique, as a single vulnerability may affect multiple versions

Version	Total	Baseline	Regressive	New
10	351	286	64	1
10.0.1	318	318	-	-
10.0.2	360	360	-	-
10.0.3	335	335	-	-
10.0.4	294	294	-	-
10.0.5	268	268	-	-
10.0.6	237	237	-	-
10.0.7	188	188	-	-
10.0.8	163	163	-	-
10.0.9	162	162	-	-
10.0.10	139	139	-	-
10.0.11	98	98	-	-
10.0.12	-	-	-	-
17	170	135	35	-
17.0.1	145	110	35	-
17.0.2	114	79	35	-
17.0.3	96	61	35	-
17.0.4	81	47	34	-
17.0.5	46	30	16	-
17.0.6	28	28	-	-

Table 3: Counts of vulnerabilities by type affecting ESR. Totals are not unique, as a single vulnerability may affect multiple versions

3.2.3 Are the RRC vulnerabilities easier to find?

With traditional defect and vulnerability discovery models, the expectation is that the 'low-hanging fruit' vulnerabilities in new code are found and patched quickly [45, 1].

Looking at traditional non-RRC software, Clark, *et al* [13] suggested that these models do not accurately represent the early lifecycle of vulnerability disclosure. Instead, there appears to be a relatively long period before the first vulnerability in new software is disclosed, after which the rate of vulnerability disclosure in that version of code increases. The authors speculated that this period corresponds to the attacker's learning curve.

If new code, released without a traditionally long code review process (as in RRC) is bad for security, then the vulnerabilities disclosed should not only be new, but found quickly (*i.e.*, they should be low-hanging fruit.) We should also see the rate of vulnerability disclosure for vulnerabilities introduced in the new code increase roughly in proportion to the LOC added. If, on the other hand, it takes time for an attacker to become familiar with the new code, then the vulnerabilities should take longer to find and those disclosed will be primarily from code introduced in older versions.

Pre-RRC Release Date	Total	Post-RRC Release Date	Total
2/3/09	26	5/1/11	29
3/18/09	30	6/21/11	38
4/30/09	41	8/16/11	35
6/12/09	33	9/27/11	32
7/25/09	9	11/8/11	20
9/6/09	20	12/20/11	29
10/19/09	32	1/31/12	24
12/1/09	32	3/13/12	24
1/13/10	14	4/24/12	59
2/25/10	29	6/15/12	1
4/9/10	1	6/26/12	-
5/22/10	20	7/17/12	42
7/4/10	26	8/28/12	54
8/16/10	25	9/6/12	-
9/28/10	27	10/9/12	37
11/10/10	22	11/19/12	46
12/23/10	-	1/8/13	76
2/4/11	24	2/19/13	1
3/19/11	2	4/2/13	35

Table 4: Counts of vulnerabilities disclosed every 6 weeks, in the 18 6-week periods preceding the switch to RRC and in the 18 periods following.

Table 2 lists the total number of vulnerabilities disclosed that affect each rapid release version. For each version, we list the total number of those that affect the baseline version, the total number that are regressive, (also affect earlier versions), and the total number newly introduced. Table 3 lists the corresponding data for the ESR versions.

On average, across all the RRC versions (5-20), approximately 75% of the vulnerabilities affecting each RRC version are baseline. A further 22% of them are regressive. This means that for any version, on average, 97% of the vulnerabilities disclosed since RRC was implemented *were not found in the new code while it was the current version.*

Accounting for the fact that RRC versions 5.0-9.0 were released before the release of the first ESR version, from the release of version 5.0 up to 20.0, a total of 617 active vulnerabilities were disclosed. Of those, only 16% are new vulnerabilities (ones which originate in RRC source code) *across all 15 versions studied.* Looking only at these new vulnerabilities, we see that fewer than half (41.5%) of them were disclosed during the current lifetime of the originating version. In other words, while vulnerabilities *are* found in the code that is introduced each RRC iteration, the overwhelming majority of them *are not* found during the 6-week period following their initial release. As this is written, even in the worst case, version 19, with 35 original vulnerabilities, 70% of were not disclosed until version 21, released 12 weeks later.

Table 4 lists the number of vulnerabilities disclosed in Firefox in the 6 week period preceding and following the change to the RRC; these values are calculated by time period in which they were disclosed, without regard to version affected. On average, there were 22 vulnerabilities disclosed

every 6 weeks in the two years preceding the change over to RRC and 30 vulnerabilities disclosed every 6 weeks after. However, there is a high variance, with no vulnerabilities found in some periods, and as many as 70 in others. At first blush, this increase in the rate of disclosure might seem to indicate that RRC vulnerabilities are low-hanging fruit. However, from Table 2, most of the RRC vulnerabilities are either baseline or regressive, and therefore in code that had been available for a longer period of time (rather than relatively unfamiliar new code), further supporting the model of an attacker learning curve [31, 25, 13].

Version	Total	Baseline	Regressive	New
5	38	19	19	-
5.0.1	38	19	19	-
6	38	19	19	-
6.0.1	37	18	19	-
6.0.2	37	18	19	-
7	37	18	19	-
7.0.1	15	12	3	-
8	37	18	19	-
8.0.1	36	18	18	-
9	36	18	18	-
9.0.1	14	12	2	-
10	13	12	1	-
10.0.1	13	12	1	-
10.0.2	13	12	1	-
11	14	12	2	-
12	13	10	1	2
13	11	10	1	-
13.0.1	8	8	-	-
14	9	9	-	-
14.0.1	8	8	-	-
15	9	9	-	-
15.0.1	4	4	-	-
16	2	2	-	-
16.0.1	2	2	-	-
16.0.2	2	2	-	-
17	2	2	-	-
17.0.1	1	1	-	-
18	1	1	-	-
18.0.1	1	1	-	-
18.0.2	-	-	-	-
19	4	-	1	3
19.0.1	4	-	4	-
19.0.2	1	-	1	-

Table 5: Count by type of RRC vulnerabilities that do not affect ESR (correspondence between RRC versions 10 or greater and ESR versions 10 and 17 is given in Tables 2 and 3). Totals are not unique, as a single vulnerability may affect multiple versions

Lastly, we look at those vulnerabilities that were introduced by new code in RRC, that *do not* affect the corresponding ESR version. How quickly were they discovered and disclosed? As we see in Table 5, of the 45 vulnerabilities that affect only RRC and do not affect ESR code, only 5 (2 in version 12.0 and 3 in version 19.0) actually originate in the newly released versions. More importantly, *only these 5* were disclosed during the 6 weeks that that version was current. While the new code surely contains vulnerabilities *they are not being found and disclosed while the version in which they originate is current.*

We have shown that Firefox's RRC strategy did not increase the rate of vulnerability discovery and disclosure and, that the vulnerabilities disclosed while a particular RRC version was current were *not* low-hanging fruit. It does indeed appear that during the RRC lifecycle, the time to find vulnerabilities and learn how to exploit them in new code compensates for the presumed increase in the density of vulnerabilities in immature code.

4. RELATED WORK

Our work involves the security impacts of software development practices, analysis of these practices in the context of the Firefox web browser, and lifecycle issues for software security. We discuss related work in each of these areas.

4.1 Software development

Previous work [12, 46, 3, 13] has addressed the effects of development practices on the quality and security of software.

Woody [52] surveyed Agile developers about the impact of security engineering activities on software development within an Agile approach. Notably, the survey found that many industry-standard frameworks, including:

1. the design requirements, threat modeling, and code review recommended practices in Microsoft's SDL's [18],
2. the risk analyses and external review recommended in Cigital Touchpoints [35], and
3. the risk analyses, critical assets and UMLSec in the Common Criteria for Information Technology Security [20],

are at least partially incompatible with the rapid delivery approach.

Seacord [47] notes that the traditional model of patch-and-install is problematic as "patches themselves contain security defects. The strategy of responding to security defects is not working. There is a need for a prevention and early security defect removal strategy." Indeed, this paper directly addresses this concern by acknowledging that the changes to the Firefox codebase with each release contain vulnerabilities that, while introduced by this model of rapid releases, may be effectively mitigated by these same large changes in subsequent releases.

Bessey *et al.* [6] discuss the prevailing attitudes towards software upgrades in terms of the number of bugs generated by each release. They assert that users want

> "different input (modified code base) + different function (tool version) = same result",

highlighting the delicate balancing act in traditional models of software development between the users' desire for new features and the impulse to squash as many bugs as possible in existing code. We have presented our hypothesis on this tradeoff by showing that a rapidly churning codebase (the "moving targets" of the paper title) may serve to mitigate security flaws, quickly deprecating code that introduced them, while providing new features to users to maintain market share.

Ozment and Schechter's [46] study of vulnerabilities in OpenBSD focused on the effect of slow, monolithic code changes on the introduction of new vulnerabilities. In this paper we have sought to analyze the impact of a rapidly shifting codebase and Agile programming methodology on vulnerability discovery. Ozment and Schechter did find that *baseline* (they use "foundational") vulnerabilities made up the vast majority of all security flaws. Our analysis indicates that Firefox RRC shares this same property.

4.2 Firefox Software Engineering

Firefox has been the focus of prior research concerning software quality on account of its (relatively) long history, large, open source codebase, active developer community and massive user base. Khomh, *et al.* [33] examined the effect of the rapid release model on the abstract 'quality' of Firefox, using concrete metrics such as the number of bugs reported after release, median daily crash count and median uptime. Their study seeks to tackle entirely different questions than our approach; we are concerned with the effects of changing a significant proportion of the codebase every six weeks on vulnerability discovery.

Almossawi's [2] work deals with the maintainability of Firefox since the start of the RRC model from the point of view of metrics such as cyclomatic complexity [30], intermodule dependencies, and defect density. Our work serves to compliment Almossawi's analysis by addressing the impact of this model of development on security flaws.

4.3 Lifecycle issues

Arbaugh, *et al.*'s [3] foundational work on vulnerability lifecycles demonstrated that the usable lifetime of vulnerabilities is far longer than expected. They concluded that "Windows of Vulnerability" exist, during which software is more likely to be compromised. We have looked at an earlier point in the lifecycle, and our data regarding when vulnerabilities are discovered and the existence of a learning curve are consistent with Arbaugh, *et al.*'s [3] data and conclusions.

Jonsson and Olovsson [31] tested the effect an attacker's knowledge and experience had on successfully compromising a system. Assigning students to attack a University computer system, they measured number of successful breaches, rate of breach and experience level. They concluded that there appears to be a learning curve that disadvantages the less experienced attacker.

Gopalakrishna and Spafford [25] presented a trend analysis of vulnerabilities reported on Bugtraq, CVE and ICAT. They speculated that the increased rate of discovery of vulnerabilities of the same type in a piece of software was the result of a learning period. They reasoned that this 'learning' was the period of time required for a given piece of software to gain a "critical-mass" of users before bugs are discovered.

However, as Ozment [45] points out, this incorrectly assumes that some fixed proportion of the total user population are looking for vulnerabilities. Ozment conjectured scenarios in which an attacker discovers a vulnerability or reads about the details of one, and applies these "lessons learned" to a similar domain by attempting an attack of a similar type. This observation is the contrapositive to the benefits of rapid releases we have proposed in this paper: the usefulness of these "lessons learned" is minimized as the section of the codebase relevant to the type of vulnerability in question may have already been deprecated by the time the attacker applies this learning. Indeed, this is further supported by the Bug Bounty findings presented by Coates [15], wherein the vast majority of flaws reported fall into a small set of classes (*e.g.*, CSRF and XSS bugs account for 70% of those reported).

Clark *et al.* [13] posited the existence of a grace period (a "honeymoon") enjoyed by software immediately following its release, before attackers have time to adapt their tools and methodologies to the new target. They suggested that this "Honeymoon Effect" was due to a learning curve faced by potential adversaries after a new release that has to be overcome before vulnerabilities are discovered, independent of any security practices undertaken during the development process.

5. CONCLUSION

Intuition suggests a tension between the rapid deployment of new software features and the avoidance of software defects, particularly those affecting security. The rapid release strategy of Firefox, in which new software releases, with new features, are rolled out on an aggressive schedule, seems as if it could only come at the expense of security. Users concerned with security, we might assume, would be better off eschewing the latest features in favor of more mature, stable releases. "Agile programming", particularly in an application as exposed as a web browser, should be a security disaster.

At least with respect to vulnerabilities disclosed during each Firefox release's lifecycle, based on our data-driven study, this intuition appears to be wrong. Vulnerabilities are disclosed in the older code at least as often as they are in the newer code. This is both surprising and encouraging news. It suggests that during the active lifecycle, the adversary's ability to discover security defects is *dominated less by the intrinsic quality of the code and more by the time required to familiarize themselves with it.* It suggests that the Firefox rapid-release cycles expose the software to a shorter *window of vulnerability* [3]. Frequent releases of new features appear to have provided the Firefox developers with new grace periods or *second honeymoons* (using the terminology of Clark, *et al.* [13]). While there may also be other factors affecting vulnerability discovery which are changing over the duration of software evolution we studied, it is clear that the net effect, seen in our data, has been the attenuation of the attacker.

We chose the Firefox browser as the basis for our study as it was originally architected using the traditional development model and switched to rapid-release midstream. It will be interesting to see if other software systems, including those that have been designed and developed using *only* Agile methods share the same properties. It will also be interesting to see what effect the switch to silent auto-updates has had on the vulnerability life-cycle. However, the dataset that we integrated for Firefox with its large code-base, and large user-base, coupled with its prominence as an attack target is strongly suggestive that the rapid release strategy has significant and unexpected security advantages in real world systems.

Even while generalization remains an open question, in Firefox, the unexpected benefit of frequent large code releases is a lengthening of the attacker's learning curve. The findings reported here further support the ideas that familiarity with a codebase is a useful heuristic for determining how quickly vulnerabilities will be discovered and, consequently, that software reuse (exactly because it is already familiar to attackers) can be more harmful to software security than beneficial.

Our data and analysis suggest that the pattern exhibited by vulnerability disclsure in Firefox is the result of would-be attackers having to re-learn and re-adapt their tools in response to a rapidly changing codebase and is consistent

with a "Honeymoon Effect" [13]. These results should lead software developers to question conventional software engineering wisdom when security is the goal.

6. ACKNOWLEDGMENTS

This work is partially supported by MURI grant FA9550-12-1-0400 "Science of Cyber Security: Modeling, Composition, and Measurement", administered by the U.S. Air Force under Grant FA9550-08-1-0352. This work is also partially supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4020. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Project Agency and Space and Naval Warfare Systems Center Pacific.

7. REFERENCES

[1] O.H. Alhamzi and Y.K. Malaiya. Application of vulnerability discovery models to major operating systems. *IEEE Transactions on Reliability*, 57:14–22, 2008.

[2] Ali Almossawi. How maintainable is the Firefox codebase?, May 2013. `http://almossawi.com/firefox/prose/`.

[3] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–59, 2000.

[4] Baker, Mitchell. Mozilla Blog. `http://blog.lizardwrangler.com/2011/08/25/rapid-release-process/`.

[5] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001. `http://www.agilemanifesto.org/`.

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[7] Konstantin Beznosov and Philippe Kruchten. Towards agile security assurance. In *Proceedings of the 2004 Workshop on New Security Paradigms*, pages 47–54. ACM, 2004.

[8] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 20(5):43–57, May 1985.

[9] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1:57–94, 1995.

[10] Brink, DerekA. Security and the Software Development Lifecycle: Secure at the Source. `download.microsoft.com/download/9/D/4/9D403333-C4F6-4770-A330-89661BE545CF/Aberdeen_SecureSource.pdf`.

[11] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.

[12] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995. `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0201835959`.

[13] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 251–260, New York, NY, USA, 2010. ACM.

[14] Peter Coad, Eric LeFebrve, and Jeff De Luca. Feature-driven development. *Java Modeling in Color with UML*, pages 182–203, 1999.

[15] Michael Coates. Security Evolution - Bug Bounty Programs for Web Applications, September 2011. `http://www.slideshare.net/michael_coates/bug-bounty-programs-for-the-web`.

[16] Kieran Conboy. Toward a conceptual framework of agile methods: a study of agility in different disciplines. In *Extreme Programming And Agile Methods - XP/ Agile Universe 2004, Proceedings*, pages 37–44. ACM Press, 2004.

[17] Forrester Consulting. State of Application Security: Immature Practices Fuel Inefficiencies, but Positive ROI Is Attainable - A Forrester Consulting Thought Leadership Paper Commissioned by Microsoft. 2011. `http://www.microsoft.com/en-us/download/details.aspx?id=2629`.

[18] Microsoft Corporation. Microsoft Security Development Lifecycle for Agile. 2009. `http://www.microsoft.com/security/sdl/discover/sdlagile-onetime.aspx`.

[19] Microsoft Corporation. `http://www.microsoft.com/en-us/news/speeches/2013/06-26build2013.aspx`, 2013.

[20] Common Criteria. Common Criteria for Information Technology Security Evaluation. Technical report, September 2012.

[21] Michael A. Cusumano and Richard W. Selby. How Microsoft builds software. *Communications of the ACM*, 40:53–61, June 1997.

[22] CVE. Common vulnerabilities and exposures. `http://cve.mitre.org`, 2008.

[23] M. Finifter, D. Akhawe, and D. Wagner. An Empirical Study of Vulnerability Reward Programs. In *22nd USENIX Security Symposium*, 2013.

[24] Mozilla Foundation. Mozilla firefox esr overview, 2014. `https://www.mozilla.org/en-US/firefox/organizations/faq/`.

[25] Rajeev Gopalakrishna and Eugene H. Spafford. A trend analysis of vulnerabilities. *CERIAS Tech Report 2005-05*, May 2005.

[26] Duncan Harris. Oracle Software Security Assurance. Technical report, 2014. `http://www.oracle.com/us/support/assurance/overview/index.html`.

[27] Jim Highsmith. *Adaptive software development: a collaborative approach to managing complex systems*. Addison-Wesley, 2013.

[28] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, May 2006.

[29] Pankaj Jalote, Brendan Murphy, and Vibhu Saujanya Sharma. Post-release reliability growth in software products. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–20, 2008.

[30] George Jelen. Sse-cmm security metrics. In *NIST and CSSPAB Workshop*, 2000.

[31] E. Jonsson and T. Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, Apr 1997.

[32] Hossein Keramati and S-H Mirian-Hosseinabadi. Integrating software development security activities with agile methodologies. In *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, pages 749–754. IEEE, 2008.

[33] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox. In *Mining Software Repositories, 2012 9th Working Conference*, Kingston, Ontario, Canada, June 2012.

[34] Anthony Laforge. Release Early, Release Often, July 2010. `http://blog.chromium.org/2010/07/release-early-release-often.html`.

[35] Gary McGraw. Software Security Touchpoint: Architectural Risk Analysis. Technical report, 2010. `http://www.cigital.com/presentations/ARA10.pdf`.

[36] Gary McGraw and Brian Chess. The building security in maturity model(bsimm). In *Proceedings of the 18th USENIX Security Symposium (USENIX Security '09)*, Montreal, Canada, August 2009.

[37] J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Andy Wigley, and Kishore Gopalan. Security Guidelines for .NET Framework Version 2.0. Technical report, October 2005. `http://msdn.microsoft.com/en-us/library/aa480477.aspx`.

[38] Mozilla. Bugzilla@Mozilla. `https://bugzilla.mozilla.org/`, September 2013.

[39] Mozilla. Mozilla Foundation Security Advisories. `https://www.mozilla.org/security/announce/`, September 2013.

[40] John D. Musa. A theory of software reliability and its application. *IEEE Transactions on Security Engineering*, SE-1:312–327, September 1975.

[41] John D. Musa, Anthony Iannino, and Kasuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[42] Johnathan Nightingale. Mozilla blog post future releases, 2011. `https://blog.mozilla.org/futurereleases/2011/07/19/every-six-weeks/`.

[43] NIST. National Vulnerability Database. `http://nvd.nist.gov`, 2008.

[44] Department of Homeland Security. *SECURITY IN THE SOFTWARE LIFECYCLE: Making Software Development Processes– and Software Produced by Them– More Secure*. 2006. `http://resources.sei.cmu.edu/asset_files/WhitePaper/2006_019_001_52113.pdf`.

[45] Andy Ozment. Improving vulnerability discovery models. In *QoP '07: Proceedings of the 2007 ACM Workshop on Quality of Protection*, pages 6–11, New York, NY, USA, 2007. ACM.

[46] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[47] Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, June 2008.

[48] Mikko Siponen, Richard Baskerville, and Tapio Kuivalainen. Integrating security into agile development methods. In *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*, pages 185a–185a. IEEE, 2005.

[49] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. 2002.

[50] John Viega. Building Security Requirements with CLASP. In *Proc. ACM SESS*, pages 1–7, 2005.

[51] Jaana Wäyrynen, Marine Bodén, and Gustav Boström. Security engineering and extreme programming: An impossible marriage? In *Extreme programming and agile methods-XP/Agile Universe 2004*, pages 117–128. Springer, 2004.

[52] Carol Woody. Agile security review of current research and pilot usages. *SEI Library White Paper*, 2013. `http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=70232`.