Familiarity Breeds Contempt

The Honeymoon Effect and the Role of Legacy Code in Zero-Day Vulnerabilities

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

Stefan Frei
Secunia
sfrei@secunia.com

Matt Blaze
University of Pennsylvania
blaze@cis.upenn.edu

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

ABSTRACT

Work on security vulnerabilities in software has primarily focused on three points in the software life-cycle: (1) finding and removing software defects, (2) patching or hardening software after vulnerabilities have been discovered, and (3) measuring the rate of vulnerability exploitation. This paper examines an earlier period in the software vulnerability life-cycle, starting from the release date of a version through to the disclosure of the fourth vulnerability, with a particular focus on the time from release until the very first disclosed vulnerability.

Analysis of software vulnerability data, including up to a decade of data for several versions of the most popular operating systems, server applications and user applications (both open and closed source), shows that *properties extrinsic to the software play a much greater role in the rate of vulnerability discovery than do intrinsic properties such as software quality.* This leads us to the observation that (at least in the first phase of a product's existence), software vulnerabilities have different properties from software defects.

We show that the length of the period after the release of a software product (or version) and before the discovery of the first vulnerability (the 'Honeymoon' period) is primarily a function of familiarity with the system. In addition, we demonstrate that legacy code resulting from code re-use is a major contributor to both the rate of vulnerability discovery and the numbers of vulnerabilities found; this has significant implications for software engineering principles and practice.

1. INTRODUCTION

Software vulnerabilities are the root cause of many security breaches, so understanding software systems is essential to developing models for how and when to invest effort in securing software. The most important software systems to understand are those of large scale and those in wide use. Since almost all software systems today are large and complex, we can focus our attention on those in wide use. Ranging from document preparation programs to web browsers and operating systems, such systems can each comprise millions of lines of source code, a very rough measure of software complexity. Given the importance of such systems, models for their creation, use, maintenance and upgrades - their "life-cycle" - are clearly necessary.

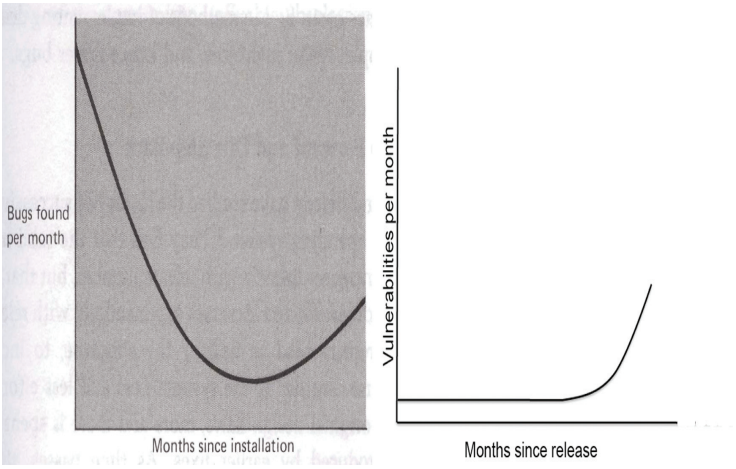

Figure 1: Bugs per month, Left:Figure 11.2 from "The Mythical Man Month", Right: Security vulnerabilities per month

Models are useful in estimating project costs and timing. For example, if a model predicts that the bug discovery rate drops rapidly after an initial flurry of discoveries, this fact can be used to determine when software is ready for release: once the rate has reached an acceptable level, the software can be shipped. Such estimation can have significant economic effects upon an enterprise: ship too early and pay a price in service calls; ship too late and potentially lose customers who might look elsewhere. A powerful predictive model can therefore be worth significant amounts of revenue, as it allows trading development costs and time against a combination of sales revenue and maintenance costs.

Software Reliability Models (SRMs) are primarily concerned with increasing the quality of the code by predicting and locating software defects. A major assumption made

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

by SRMs is that software is released with some number of defects that can be categorized based on how easy each is to find. A further assumption is made that the easy-to-find defects are discovered and fixed *early* in the software life-cycle, quickly leading to a state where only difficult-to-find vulnerabilities are left and the software can be considered reliable. Figure 11.2 from Brooks [5] is reproduced on the left of Figure 1 to illustrate this point.

Software Vulnerability Discovery Models (VDMs) resemble SRMs, but VDMs focus predominantly on predicting attacks against mature software systems. VDMs rely on the *intrinsic qualities* of the software for a measure of its initial security. For a VDM the expectation is that the low-hanging fruit vulnerabilities are found quickly and patched. The remaining vulnerabilities (which are increasingly difficult to find) are presumed to take much longer to discover, and the software is considered "secure". A VDM with those expectations would predict that vulnerabilities are found fastest shortly after the release of a product, and the rate of discovery decreases thereafter.

The implications of such a VDM are significant for software security. It would suggest, for example, that once the rate of vulnerability discovery was sufficiently small, that the software is "safe" and needs little attention. It also suggests that software modules or components that have stood this "test of time" are appropriate candidates for reuse in other software systems. If this VDM model is wrong, these implications will be false and may have undesirable consequences for software security.

Unlike much of the previous work [25, 2, 28] which focused on understanding time to exploit after a vulnerability has been discovered, this paper focuses on measuring time to vulnerability discovery.

The remainder of the paper is organized as follows. Section 2 describes our unique dataset of vulnerabilities, covering several versions of the most popular software products, operating systems, server applications and user applications.

In Section 3 we analyze this data, which show that the period between the release date of a product and its very first 0-day vulnerability is considerably longer than the mean time between the first vulnerability and second or between the second and the third. We call this unexpected grace period the *honeymoon effect* and believe it to be important, because these numbers challenge our expectations and intuition about the effect of software quality on security. The interval between software release and the discovery of it's first 0-day vulnerability also appears to be a strong predictor of the arrival rate of subsequent vulnerability discoveries.

The honeymoon effect also illustrates another incompatibility between current software engineering practices and security: the effect of code reuse. "Good programmers write code, Great programmers reuse " is a well-known aphorism, and the assumption made is that reusing code is not only more efficient, but since the code has already been deployed successfully, it is more reliable and therefore, by implication, also more secure. In Section 4 our data again show this is not the case.

We set our results in the context of prior work in Section 5 and conclude the paper by summarizing our claims and discussing the implications for engineering secure software systems in Section 6.

2. OUR DATASET

In this paper, we are concerned specifically with the early post-release vulnerability life-cycle for modern, mass market software, including operating systems, web clients and servers, text and graphics processors, server software, and so on.

Our analysis focuses on publicly distributed software released between 1999 and 2007. (2007 is the latest date for which complete vulnerability information was reliably available from various published data sources). We included both open and closed source software.

To encompass the most comprehensive possible range of relevant software releases, we collected data about all released versions of the major operating systems (Windows, OS X, Redhat Linux, Solaris, FreeBSD), all released versions of the major web browsers (Internet Explorer, Firefox, Safari), and all released versions of various server and end user applications, both open and closed source. The server and user applications were based on the top 25 downloaded / purchased / favorite application identified in lists published by ZDNet, CNet, and Amazon, excluding only those applications for which accurate release date information was unavailable or that were not included in the vulnerability data sources described below. In total, we were able to compile data about 38 of the most popular and important software packages.

For each software package and version during the period of our study, we examined public databases, product announcements, and published press releases to assign each version a release date. For the period of versions (1990-2007) and for the period of vulnerabilities (1999-2008), we identified 700 distinct released versions ('major' and 'minor') of the 38 different software packages.

We then compiled a dataset of more than 30,000 exploitable vulnerabilities that were disclosed during the period under analysis (January 1999 through January 2008). Our baseline sources were publicly available databases from the National Vulnerability Database (NVD) [23] and from the Common Vulnerabilities and Exposures (CVE) [9] initiative that feeds NVD. (For each vulnerability, NVD provides a publication date, a short description, a risk rating, references to original sources, and information on the vendor, version and name of the product affected.) We also downloaded, parsed, and correlated the information from over 200,000 individual security bulletins from several "Security Information Providers" (SIPs), choosing the set of SIPs based on criteria such as independence, accessibility, and available history of information. Ultimately, we processed all security advisories from the following seven SIPs: Secunia, US-CERT, SecurityFocus, IBM ISS X-Force, SecurityTracker, iDefense's (VPC), and TippingPoint(ZDI) [29, 33, 30, 14, 34, 31, 15, 32].

For this study, we selected from these bulletins and database entries bugs identified as *exploitable vulnerabilities* that render the software vulnerable to actual attack and for which a practical exploit has been demonstrated. We then calculated the *initial disclosure date* for each exploitable vulnerability to be the earliest calendar day on which information on a specific vulnerability is made freely available to the public in a consistent format by some recognized published source [11]. To help ensure accuracy, we manually checked and corrected over 3,000 instances of software version information for the specific product versions under analysis in this paper to normalize for inconsistencies in NVD's vulnerability to product mapping.

3. THE HONEYMOON EFFECT

Virtually all mass-market software systems undergo a lengthy period, after their release, during which end-users discover and report bugs and other deficiencies. Most software suppliers (whether closed-source or open-source) build into their life-cycle planning a mechanism for reacting to bug reports, repairing defects, and releasing patched versions at regular intervals. The number of latent bugs in a particular version of a given version of a given piece of software thus tends to decrease over time, with the initial, unpatched, release suffering from the largest number of defects. (This excludes, of course, defects introduced by patches, which are a minority in practice). In systems where bugs are fixed in response to user reports, the most serious and easily triggered bugs would be expected to be reported early, with increasingly esoteric defects accounting for a greater fraction of bug reports as time goes on.

Empirical studies in both the classic [5] and the current [16] software engineering literature have shown that, indeed, this intuition reflects the software life-cycle well (see Figure 2). Invariably, these and other software engineering studies have shown that the rate of bug discovery is at its highest immediately after software release, with the rate (measured either as inter-arrival time of bug reports or as number of bugs per interval) slowing over time.

Note that some (but not all) of the bugs discovered and repaired in this process represent *security vulnerabilities;* in security parlance a vulnerability that allows an attacker to exploit a newly discovered, previously unknown bug is called a *0-day* vulnerability. Virtually all software vendors give high priority to repairing defects once a 0-day exploit is discovered.

It seems reasonable, then, to presume that users of software are at their most vulnerable, with software suffering from the most serious latent vulnerabilities, immediately after a new release. That is, we would expect attackers (and legitimate security researchers) who are looking for bugs to exploit to have the easiest time of it early in the life cycle. This, after all, is when the software is most intrinsically weak, with the highest density of "low hanging fruit" bugs still unpatched and vulnerable to attack. As time goes on, after all, the number of undiscovered bugs will only go down, and those that remain will presumably require increasing effort to find and exploit.

In other words, to the extent that security vulnerabilities are a consequence of software bugs, conventional software engineering wisdom tells us to expect the discovery of 0-day exploits to follow the same pattern as other reported bugs. The pace of exploit discovery should be at its most rapid early on, and slowing down as the software quality improves and the "easiest" vulnerabilities are repaired.

But our analysis of the rate of the discovery of exploitable bugs in widely-used commercial and open-source software, tells a very different story than what the conventional software engineering wisdom leads us to expect. In fact, new software overwhelmingly enjoys a *honeymoon* from attack for a period after it is released. The time between release and the first 0-day vulnerability in a given software release tends to be markedly longer than the interval between the first and the second vulnerability discovered, which in turn tends to be longer than the time between the second and the third. That is, when the software it at its *weakest*, with the "easiest" exploitable vulnerabilities still unpatched, there is a *lower* risk that this will be discovered by an actual attacker on a given day than there will be *after the vulnerability is fixed!*

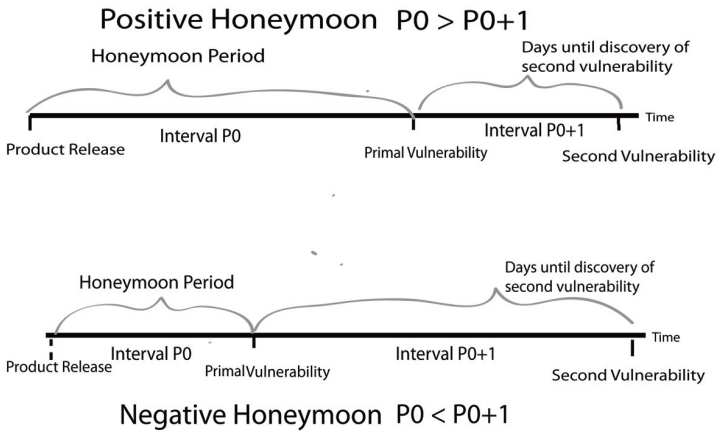

Figure 3: The Honeymoon Period, both Positive and Negative time-lines

3.1 The Honeymoon Effect and Mass-Market Software

For the purposes of this paper, we define the first (publicly reported) exploitable vulnerability as the *primal* vulnerability, we define a software release as experiencing a *positive honeymoon* if the interval p_0 between the (public) release of the software and the primal vulnerability in the software is greater than the interval p_{0+1} between the primal vulnerability and the second(publicly reported) vulnerability.(see Figure 3) We will refer here to the interval p_0 as the *honeymoon period* and the ratio p_0/p_{0+1} as the *honeymoon ratio*. In other words, a software release has experienced a positive honeymoon when its honeymoon ratio > 1.

We examined 700 software releases of the most popular recent mass-market software packages for which release dates and vulnerability reports were available (see Section 2). In 431 of 700 (62%) of releases, the honeymoon effect was positive. Most notably, the median overall honeymoon ratio (including both positive and negative honeymoons) p_0/p_{0+1} was 1.54. That is, the median time from initial release and the primal vulnerability is 1 1/2 times greater than the time from primal to the discovery of the second. The honeymoon effect is not only present, it is quite pronounced, and the effect is even more pronounced when we exclude minor version updates and focus on major releases. For major releases, the honeymoon ratio(including both positive and negative honeymoons) rises to 1.8.

Remarkably, positive honeymoons occur across our entire dataset for all classes of software and across the entire period under analysis. The honeymoon effect is strong whether the software is open- or closed- source, whether it is an OS, web client, server, text processor, or something else, and regardless of the year in which the release occurred.(see Table 1)

Although the honeymoon effect is pervasive across the entire dataset, one factor appears to influence its length more than any other: the re-use of code from previous releases, which, counter-intuitively, *shortens* the honeymoon. Soft-

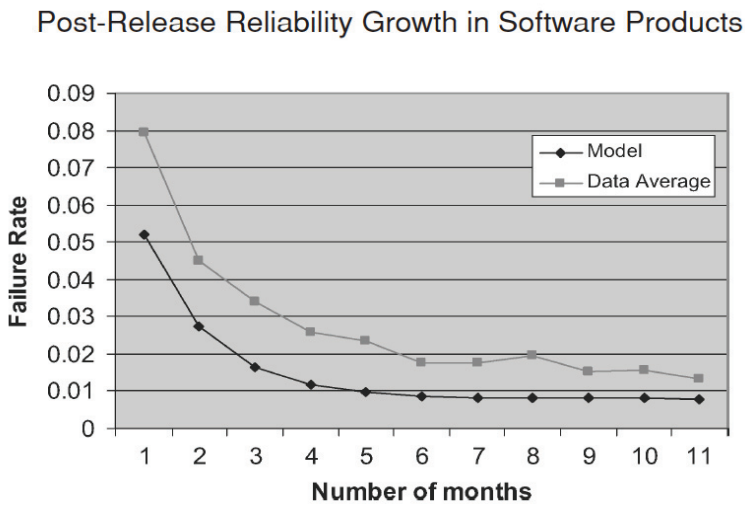

Figure 2: Current Software Engineering literature supports the Brooks life-cycle model - image taken from "Post-release reliability growth in software products", *ACM Trans. Softw. Eng Methodol. 2008* see references

Table 1: Percentages of Honeymoons by Year

Year	Honeymoons
1999	56%
2000	62%
2001	50%
2002	71%
2003	53%
2004	49%
2005	66%
2007	58%

ware releases based on "new" code have longer honeymoons than those that re-use old code. We discuss this in detail in the following sections.

3.2 Honeymoons in Different Software Environments

The number of days in the honeymoon period varies widely from software release to software release, and ranged from a single day to over three years in our dataset. The length of the honeymoon presumably varies due to many factors, including the intrinsic quality of the software and extrinsic factors such as attacker interest, familiarity with the system, and so on.

To "normalize" the length of the honeymoon for these factors to enable meaningful comparisons between different software packages, the honeymoon ratio – the ratio of the time between release and the discovery of the first exploit and the time between the discovery of the first and the second – may be more revealing, since time to the second vulnerability discovery occurs in exactly the same software.

The median number of days in the honeymoon period across all 700 releases in our dataset was 110. The median honeymoon ratio across all releases is 1.54.

The honeymoon ratio remained positive in virtually all software packages and types. The effect is weaker, but also occurred, between the primal and second and second and third reported vulnerabilities, depending on the particular software package.

Figure 4 shows the median honeymoon ratio (and the median ratios for the intervals between the second, third and fourth vulnerabilities) for each operating system in the dataset. Figure 5 shows the median honeymoon ratio of servers, and Figure 6 shows end-user applications.

3.3 Open vs. Closed Source

The honeymoon effect is strong in both open- and closed-source software, but it manifests itself somewhat differently.

Of the 38 software systems we analyzed, 13 are open-source and 25 are closed-source. But of the 700 software releases in our dataset 171 were for closed-source systems and 508 were for open source. Open-source packages in our dataset issued new release versions at a much more rapid rate than their closed source counterparts.

Table 2: Median Honeymoon Ratio for Open and Closed Source Code

Type	Honeymoon Days	Ratios
Open Source	115	1.23
Closed Source	98	1.48

Yet in spite of its more rapid pace of new releases, open source software releases enjoyed a significantly longer median honeymoon before the first publicly exploitable vulnerability was discovered: 115 days, vs. 98 days for closed-source releases.(see Table 2)

The median honeymoon ratio, however, is shorter in open-source than in closed. The median ratio for all open-source releases was 1.23, but for closed source it was 1.48. Figure 7 shows the median honeymoon ratios for various open-source systems, and Figure 8 shows the median ratios for closed-source systems.

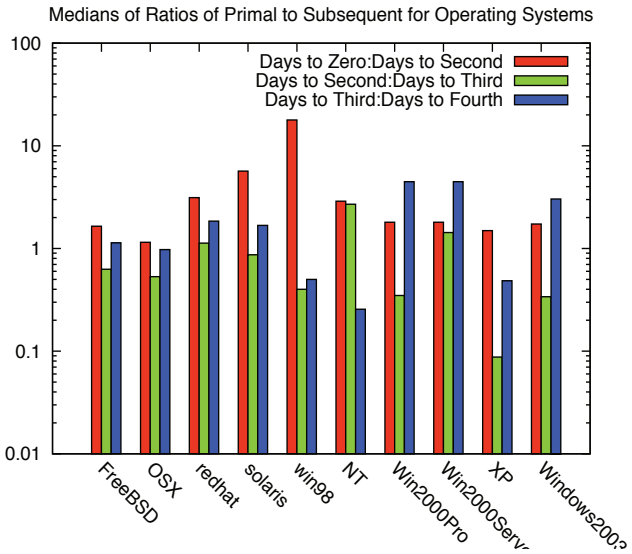

Figure 4: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for major operating systems. (Log scale. Note that a figure over 1.0 indicates a positive honeymoon).

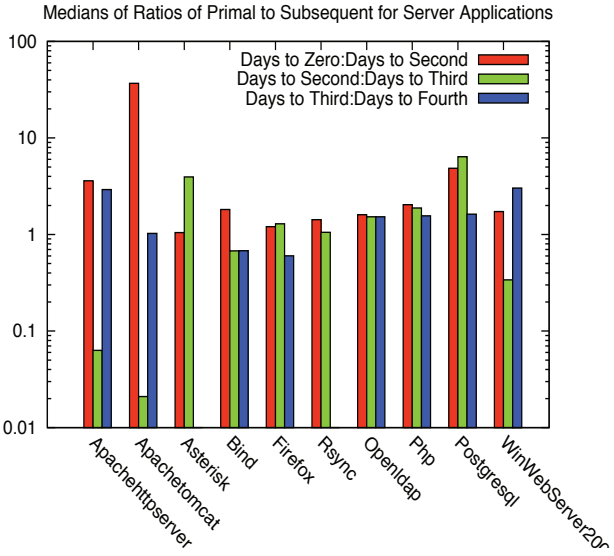

Figure 5: Honeymoon ratio of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common server applications

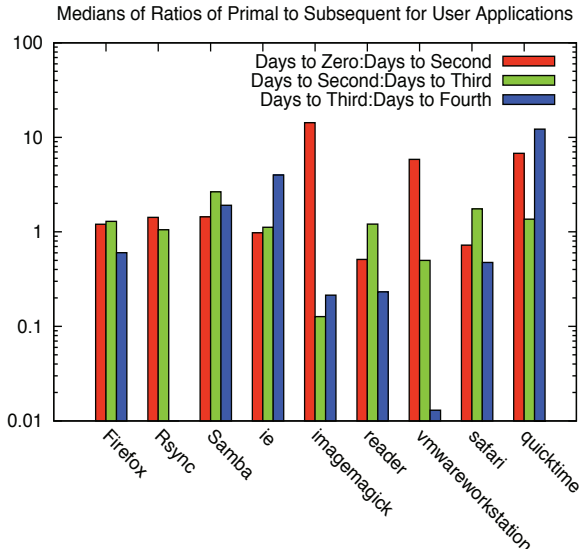

Figure 6: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common user applications

The longer honeymoon period with a shorter honeymoon ratio for open-source software suggests that it not only takes longer for attackers to find the initial bugs in open-source software, but that the rate at which they "climb the learning curve" does not accelerate as much over time as it does in closed-source systems. This may be a surprising result, given that attackers do not have the opportunity to study the source code in closed-source systems, and suggests that familiarity with the system is related to properties *extrinsic to the system* and not simply access to source code.

4. THE HONEYMOON EFFECT AND PRIMAL VULNERABILITIES

To more fully understand the factors responsible for the honeymoon effect, we examined the attributes of a particular set of *primal* vulnerabilities. In this section we compare the honeymoon periods of this set and show that primal vulnerabilities are not a result of "low-hanging fruit", and that other extrinsic properties must apply.

It is well known that as complex software evolves from one version to the next, new features are added, old ones deprecated and changes are made, but throughout its evolution much of the standard code base of a piece of software remains the same. One reason for this is to maintain backward compatibility, but an even more prevalent reason is that code re-use is a primary principle of software engineering [18, 5].

In Milk or Wine [25] Ozment *et al* measured the portion of legacy code in several versions of OpenBSD and found that 61% of legacy (their term is 'foundational') code was still present 15 releases (and 7.5 years) later. This legacy code accounted for 62% of the total vulnerabilities found. While it is not possible to measure the amounts of legacy code from version to version in closed source products as it is for open source, it is well known that the major ven-

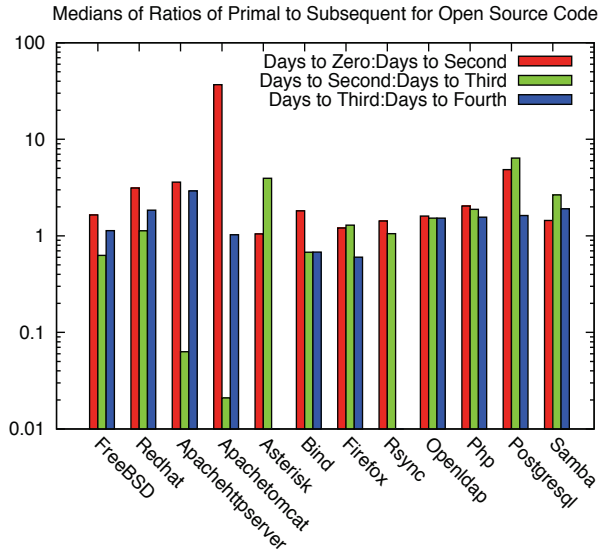

Figure 7: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for open source applications

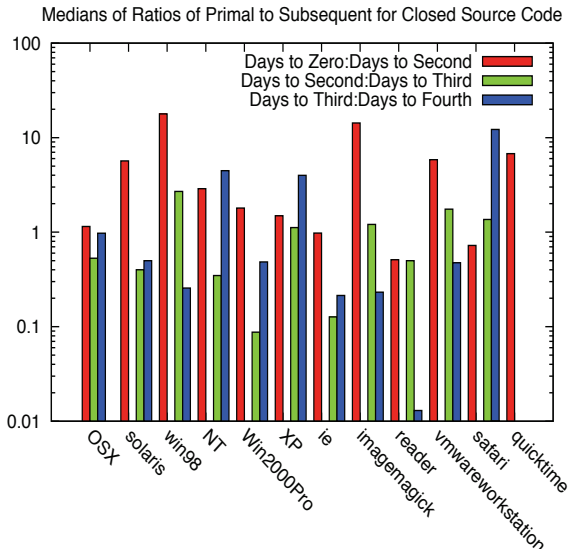

Figure 8: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for closed source applications

dors strongly encourage code re-use among their collaborating developers [19], and more importantly, it *is* possible to measure the numbers of legacy vulnerabilities. By comparing the disclosure date of a vulnerability with the release dates and product version affected, it is possible to determine which vulnerabilities discovered in the current release result from earlier versions. For example, if a vulnerability V affects versions (k,...N) (0<k<N) of a product, but not versions (1,...,k-1) and was disclosed *after* the release date of version N, we know that the vulnerability was introduced into the product with version k, and that it stayed hidden until its discovery after the release of version N. We call these *regressive* vulnerabilities as they are those vulnerabilities which are not found through normal regression testing and may lie dormant through more than one version release(sometimes for years).[1] For the purposes of this paper, we define a *regressive* vulnerability as a primal vulnerability that was discovered to affect not only version N in which it was found, but also affect one or more earlier versions (versions N-1, N-2,...,1.0)

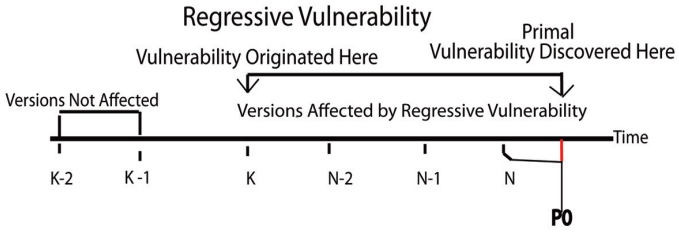

Figure 9: Regressive Vulnerability timeline

On the other hand, a *progressive* vulnerability is primal vulnerability which is discovered in version N and does not affect version N-1 or any earlier versions. A progressive vulnerability indicates that the vulnerability was introduced with the new version N. (see Figure 9)

Figure 10 shows that legacy vulnerabilities[2] make up a significant percentage of vulnerabilities across all products, e.g. 61% of the Windows Vista vulnerabilities originate in earlier versions of the OS, 40% of which originate in Windows 2000 released seven years earlier. This analysis shows that vulnerabilities are typically long-lived and can survive over many years and many product versions until discovered.

In order to ascertain whether regressive vulnerabilities could be the result of code reuse rather than configuration or implementation errors, we manually checked the NVD database description and the original disclosure sources for information regarding the type of vulnerability. We found that 92% of the regressive vulnerabilities were the result of code errors (buffer overflows, input validation errors, exception handling errors) which strongly indicates that a vulnerability that affects more than one version of a product is most likely a result of legacy code shared between versions. We removed the vulnerabilities which are the result of implementation or configuration errors from our dataset and focused exclusively on code errors.

4.1 Regressive Vulnerabilities

[1]In OpenBSD, Ozment *et al* states "It took more than two and a half years for the first half of these ... vulnerabilities to be reported." [25].

[2]including both regressives and progressives

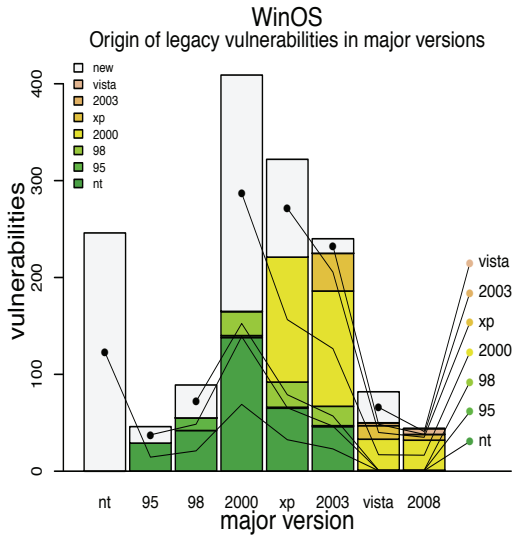

Figure 10: Proportion of legacy vulnerabilities in Windows OS

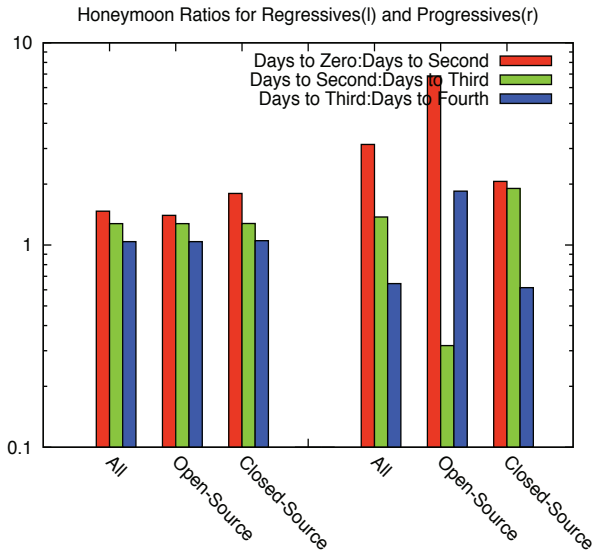

Figure 11: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common user applications

If code reuse and an attacker's familiarity with the system has an effect on the rate of vulnerability discovery, then when one examines the *primal* vulnerabilities, one should expect to see that regressive vulnerabilities make up a significant percentage of them. And indeed, after examining all the primal vulnerabilities in our data set, we find that 77% of them are regressive. (ie, 77% of the primals were found to also affect earlier versions). Table 3 lists the percentages of regressives for all, open source, closed source primals. Table 3 also shows that the percentage of regressives is even higher for open source primals (rising up to 83%), and lower for closed source (59%). The high percentage of regressive vulnerabilities is surprising, because it shows that the majority of primal vulnerabilities, (the first vulnerability found after a product is released), are not the easy to find "low-hanging fruit" one would expect from conventional software engineering defects, instead these regressives lay dormant throughout the life-time of their originating release (and possibly several subsequent releases). If these regressives had been easy to find, then presumably, *they would have been found in the version in which they originated.*

Table 3: Percentages of Regressives and Regressive Honeymoons for all Primal Vulnerabilities

Type	Total Regressives	Total Regr. Honeymoons
ALL	77%	62%
Open Source	83%	62%
Closed Source	59%	66%

4.2 The Honeymoon Effect and Regressive Vulnerabilities

Another unexpected finding is that regressive vulnerabilities also experience the honeymoon effect. Because regressive vulnerabilities have been lying dormant in the code for more than one release, and because the attackers have had more time to familiarize themselves with the product, it seems reasonable to presume that the first of these vulnerabilities would be found in a shorter amount of time than time to find the second vulnerability (whether regressive or progressive). But, our analysis shows this isn't the case. The second column of Table 3 lists the percentages of regressives that were also honeymoons. In each case whether we looked at all regressives combined, only open source or only closed source, the percentages of honeymoons is in the low to mid 60th percentile - almost the same as the total honeymoon effect for all regressives and progressives combined. Closed source does exhibit a slightly longer honeymoon effect, but not significantly so. The existence of regressive honeymoons, especially in such high proportions indicates that properties extrinsic to the quality of the code, in particular an attacker's familiarity with the system play a much greater role early on in the life-cycle of a release than previously expected.

4.3 Regressives vs. Progressives

The strong presence of the honeymoon effect even among regressive vulnerabilities leads us to wonder what if any effect regressives might have on the length of the honeymoon period. Yes, regressive vulnerabilities experience a honeymoon, but is it longer or shorter than the honeymoon for progressive vulnerabilities? The honeymoon ratio provides insight into the length of the honeymoon period. Figure 11 shows the median honeymoon ratios for regressives (all, open and closed), progressives (all, open and closed), for the vulnerabilities p_0/p_{0+1}, through p_{0+2}/p_{0+3}. The median honeymoon ratio for regressive vulnerabilities is lower than that for progressives. In fact, the honeymoon ratio for progressive vulnerabilities is almost twice as long. This strongly suggests that familiarity with the system is a major con-

tributor to the time to first vulnerability discovery. Interestingly, it doesn't seem to have a significant effect on open source code, but closed source does seem to have a longer honeymoon period, even for regressives. In other words, *familiarity shortens the honeymoon.*

4.4 Less than Zero Days

Table 4: Percentages of Primals that are Less-than-Zero (released vulnerable to an already existing exploit) and the new expected median time to first exploit, for all products, Open source and Closed Source

Type	Percentages	Median Honeymoon Period
ALL	21%	83
Open Source	18%	89
Closed Source	34%	60

Dormant vulnerabilities are not the only cause of 0-days. Legacy vulnerabilities result in a second category of regressive 0-days for which there can be no honeymoon period. These *Less-than-Zero* days occur when a new version of a product is released vulnerable to a previously disclosed vulnerability. For example, the day Windows 7 was officially released, it was discovered that it was vulnerable to several current prominent viruses which had originally been crafted for Windows XP [35] Our research shows that less-than-zero days account for approximately 21% of the total legacy vulnerabilities found, with closed source code containing the most (34%)(see Table 4). In all cases the median number of days to first exploit is reduced by approximately 1/3 and the median honeymoon ratio drops from 1.54 to 1.0. From this we conclude that not patching vulnerabilities has a significant negative effect on the honeymoon period. Of course there is no way to measure exactly when an attacker is likely to test an existing exploit against a newly released product however, the Sophoslabs [35] tests are indicative of how quickly a vendor might expect attackers to act.

5. RELATED WORK

As noted in the Introduction, both the scale of modern software systems and the scale of their deployment have made software design and engineering the focus of significant attention from scientists and engineers.

Brook's "The Mythical Man-Month" [5] is a bedrock reference for both the problems that the software engineering discipline is intended to address and its collected data (albeit from the 1960s) in support of its cogent observations. As Brooks addresses the issues in successfully engineering large software systems his focus is software defects ("bugs") rather than software security vulnerabilities. His analyses of the management issues in software engineering, particularly factors to account for in scheduling, still hold true. For example, the discussion of "Regenerative Schedule Disaster" (particularly Fig. 2.8, illustrating the added cost for training time) lends support to our observations about the time required to gain familiarity with a software system. Brook's Figure 11.2, "Bug occurrence as a function of release age", reproduced here on the left of Figure 1, shows an interval of decrease in bugs found, slowing to some minimum rate, followed by a slow rise in the rate of bugs found. This shows the effects of increased familiarity with a software system. As do many software engineering scholars, Brooks emphasizes the positive aspects of reusable software components without discussion of the potential risks from malicious actors.

Software reliability analysis is crucial to commercial firms which must deliver reliable software in a timely manner. A number of software reliability [21, 12, 27, 22] models have been developed, with a focus on bug rates and their implications for software maturity and releasability. The models, testing [26] and data collections do not address malicious actors.

Arbaugh, *et al.* [2] initiated the study of the more specialized software vulnerability life-cycle, with a particular focus on the intervals of time between when a vulnerability is known and when a software system is updated to remove the vulnerability. It is important to note, that these works focused on rate of exploitation, while this paper focuses on rate of vulnerability discovery.

Work by Jonsson, *et al.* [17] provides observations on a user population of students with quantitative evaluation of behavioral hypotheses, of which the most interesting to us is the ability to find bugs rapidly once the price is paid (in time) of learning the software system.

Alhamzi, et al [1] studied Windows 98 and Windows NT 4.0 and proposed a 3-phase S-shaped model (AIM) to describe the rate of change of cumulative vulnerabilities over time where the first phase includes time spent learning, but Ozment's analysis [24] of this and other vulnerability discovery models showed that its predictive accuracy assumed a static code-base and therefore was never tested against software spanning multiple versions. Our analysis supports an S-shaped curve model, but shows that the three phases in the AIM model do not accurately describe the data we have collected. Additionally, we are not concerned with the total number of vulnerabilities found over a product's lifetime, but with the first vulnerability found per version, as well as with a comparison of the cumulative number of days between vulnerabilities, particularly those closest to the product's release date.

Recent studies of bugs or vulnerabilities in large open source software systems [6, 25] did analyze vulnerability density across several versions and provide some data and observations that we believe support our hypothesis. First, since the software systems under study are open source software (e.g., Linux and OpenBSD) and readily available, they are learn-able by an attacker with an appropriate expenditure of time. Second, an analysis of bugs that persisted from version to version showed that such bugs were often a consequence of "cut and paste" software engineering, a crude yet effective form of software reuse. The majority of the existing vulnerability life-cycle and VDM research which makes use of the NVD dataset focused primarily on a small number of operating systems or a few server applications and in all but a few cases [25] only looked at one particular version of each (e.g. Windows NT, Solaris 2.5.1, FreeBSD 4.0 and Redhat 6.2, or IIS and Apache). In particular, Ozment and Schecter [25] found that 62% of the vulnerabilities in OpenBSD v.2.3-3.7 came from legacy code, and concluded that the original version of the source code may constitute the bulk of the later version's code base.

One large scale attempt to positively alter the rate of vulnerability discovery early on is Microsoft's Security Devel-

opment Lifecycle (SDL) which claims to have reduced the numbers of vulnerabilities found in Windows Vista's first year compared with those found in Windows XP, which does not use the SDL, (66 vs. 119) a 45% improvement. However, while Vista was in its first year, XP had been out for 6 years. We believe this also supports our hypothesis, especially since, in its first year, XP had only 28 vulnerabilities [20], a difference of 58%. [23]

Code reuse continues to be considered an important part of secure, efficient software development in both open and closed products [13, 10, 4]. However, Coverity's analysis of the lessons learned after years of using their static code analysis tool provides some possible explanations of the role legacy code plays in the honeymoon effect. For example, the authors list the most common response from software developers after the discovery of 1000+ bugs: "...The baseline is to record the current bugs, don't fix them, but do fix any new bugs... A reasonable conservative heuristic is if you haven't touched the code in years, don't modify it (even for a bug fix) to avoid causing any breakage." [3] This suggests that an attacker familiar with the legacy code that has been carried over into a newly released version would have an edge in finding new vulnerabilities in it (the legacy code), and this might have a negative effect on the honeymoon period.

In a recently published paper [28] the author analyzed the risk of first exploitation attempt using a Cox proportional model and concludes "that the exploitation process is accelerated for open source products". The focus of the paper is on measuring the rate of exploitation attempts, not on the rate of vulnerability discovery and is therefore not relevant to our paper.

6. DISCUSSION AND CONCLUSIONS

The software lifecycle has been repeatedly examined, with the intent of understanding the dynamics of software production processes, most particularly the arrival rate of software faults and failures. These rates decrease with time as updates gradually repair the errors as they are found, until an acceptable error rate is achieved.

The software vulnerability lifecycle has been less extensively studied, with most attention paid to the period after an exploit has been discovered. In attempting to understand the properties of vulnerability discovery, there are two approaches we might have taken. One approach would have been to study a single software system in depth, over an extended period, draw detailed conclusions, and perhaps generalize from them. Indeed, several of the related works mentioned above try to do just that for the middle and end phases of the lifecycle. But, another approach is to examine a large set of software systems and try to find properties that are true over the entire set and over an extended period.

We chose the latter approach for an number of reasons, which include the following: This approach allowed us to incorporate both open and closed source systems in our analysis, this approach also allowed us to analyze several different classes of software (Operating Systems, Web Browsers User applications, Server applications, etc), and this approach allowed us to discover general vulnerability properties, e.g. the honeymoon period, independent of the type of software, and without requiring a detailed analysis of the properties of each specific, individual vulnerability.

It might appear that given so many changes in tools, utilities, methodologies and goals used by both attackers and defenders over the last decade, a long term analysis would be inconsistent. To mitigate this we broke down each analysis by year and from version-to-version which are much shorter time intervals, and we demonstrated the consistency of this approach over time.

We also analyzed the role of legacy code in vulnerability discovery and found surprisingly, based on a detailed study of a large database of software vulnerabilities, that software reuse may be a significant source of new vulnerabilities. We determined that the standard practice of reusing code offers unexpected security challenges. The very fact that this software is mature means that there has been ample opportunity to study it in sufficient detail to turn vulnerabilities into exploits.

There are multiple potential causal mechanisms that might explain the existence of the honeymoon effect and the role played by familiarity. One possibility is that a second vulnerability might be of similar type to the first, so that finding it is fascilitated by knowledge derived from finding the first one. A second possibility is that the methodology or tools developed to find the first vulnerability lowers the effort required to find a subsequent ones. A third possible cause might be that a discovered vulnerability would signal weakness to other attackers (ie, blood in the water), causing them to focus more attention on that area. [7]

The first two possible causes require familiarity with the system, while the third is an example of properties *extrinsic* to the quality of the source code that might affect the length of the honeymoon period. An examination of these possible causes will appear in future work.

The period between when the error rate is low enough for release and attacker familiarity becomes high enough for an initial 0-day vulnerability we have called the *honeymoon* and its dynamics have been demonstrated in this paper to apply to the majority of popular software systems for which we had data.

The dynamics of the honeymoon effect suggest an interesting tradeoff between decreasing error rate and increasing familiarity with the software by attackers. This basic result has important implications for the arms race between defenders and attackers.

First, it suggests that a new release of a software system can enjoy a substantial *honeymoon* period without discovered vulnerabilities once it is stable, *independent of security practices*. Second, this honeymoon period appears to be a strong predictor of the approximate upper bound of the vulnerability arrival rate. Third, it suggests (as hinted at by the paper title) that attacker familiarity is a key element of the software process dynamics, and this is a contraindication for software reuse, as the greater the fraction of software reuse, the smaller the amount of study required by an attacker. Fourth, it suggests the need for more alternative approaches to security software systems than simply trying to create bug-free code.

In particular, research into alternative architectures or execution models which focuses on properties extrinsic to software, such as automated diversity, redundant execution, software design diversity [8] might be used to extend the honeymoon period of newly released software, or even give old software a *second honeymoon*.

6.1 Acknowledgments

Professors Blaze and Smith's work was supported by the

Office of Naval Research under N00014-07-1-907, Foundational and Systems Support for Quantitative Trust Management; Professor Smith received additional support from the Office of Naval Research under the Networks Opposing Botnets effort N00014-09-1-0770, and from the National Science Foundation under CCD-0810947, Blue Chip: Security Defenses for Misbehaving Hardware. Professor Blaze received additional support from the National Science Foundation under CNS-0905434 TC: Medium: Collaborative: Security Services in Open Telecommunications

References

[1] O.H. Alhamzi and Y.K. Malaiya. Modeling the vulnerability discovery process. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering(ISSRE'05)*, Washington, DC, USA, 2005.

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

[3] 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.

[4] BlackDuck. Koders.com. `http://corp.koders.com/about/`, April 2010.

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

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings, 18th ACM Symposium on Operating Systems Principles*, pages 73–82, October 2001.

[7] Sandy Clark, Matt Blaze, and Jonathan Smith. Blood in the water: Are there honeymoon effects outside software? In *In Proceedings of the 18th Cambridge International Security Protocols Workshop -pending publication*. Springer, 2010.

[8] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *In Proceedings of the 15th USENIX Security Symposium*, pages 105–120, 2006.

[9] CVE. Common vulnerabilities and exposures, 2008.

[10] Dr Dobbs Journal. Open Source Study Reveals High Level of Code Reuse. `http://www.drdobbs.com/open-source/216401796`, March 2009.

[11] Stefan Frei. *Security Econometrics - The Dynamics of (In)Security*. Eth zurich, dissertation 18197, ETH Zurich, 2009. ISBN 1-4392-5409-5, ISBN-13: 9781439254097.

[12] A.L. Goel and K. Okumoto. A time dependent error detection model for software reliability and other performance measures. *IEEE Transactions on Reliability*, R-28:206–211, August 1979.

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

[14] IBM Internet Security Systems - X-Force. X-Force Advisory. http://www.iss.net.

[15] iDefense. Vulnerability Contributor Program. http://labs.idefense.com/vcp.

[16] 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.

[17] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Trans. Softw. Eng.*, 23(4):235–245, 1997.

[18] M.C. McIlroy. Mass producted software components. *Report to Scientific Affairs Division, NATO*, October 1968.

[19] Microsoft. Internet explorer architecture. `http://msdn.microsoft.com/en-us/library/aa741312(VS.85).aspx`, 2010.

[20] Microsoft Corporation. Microsoft security development lifecycle. `http://www.microsoft.com/security/sdl/benefits/measurable.aspx`, September 2008.

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

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

[23] NIST. National Vulnerability Database, 2008.

[24] 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.

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

[26] R.E. Prather. Theory of program testing - an overview. *Bell System Technical Journal*, 72(10):3073–3105, December 1983.

[27] C.V. Ramamoorthy and F.B. Bastani. Software reliability - status and perspectives. *IEEE Transactions on Software Engineering*, SE-8(4):354–371, July 1982.

[28] Sam Ransbotham. An Empirical Analysis of Exploitation Attempts based on Vulnerabilities in Open Source Software. In *Workshop on the Economics of Information Security (WEIS)*, June 2010.

[29] Secunia. http://www.secunia.com. Vulnerability Intelligence Provider.

[30] Security Focus. Vulnerabilities Database, 2008.

[31] SecurityTracker. `http://www.SecurityTracker.com`. SecurityTracker.

[32] TippingPoint. Zero day initiative (zdi). `http://www.zerodayinitiative.com/`.

[33] US-CERT. Vulnerability statistics. `http://www.cert.org/stats/vulnerability\_remediation.html`.

[34] Vupen. Vupen security. `http://www.vupen.com`.

[35] Chester Wisniewski. Windows 7 vulnerable to 8 out of 10 viruses, 2009. `http://www.sophos.com/blogs/chetw/g/2009/11/03/windows-7-vulnerable-8-10-viruses/`.