INL/CON-12-27664
PREPRINT

Estimating Software Vulnerabilities: A Case Study Based on the Misclassification of Bugs in MySQL Server

8th International Conference on Availability, Reliability, and Security (ARES 2013)

Jason L. Wright
Jason W. Larsen
Miles A. McQueen

September 2013

Estimating Software Vulnerabilities A Case Study Based on the Misclassification of Bugs in MySQL Server

Jason L. Wright, Jason W. Larsen, Miles A. McQueen
Cyber Security R&D
Idaho National Laboratory
Idaho Falls, Idaho, USA
jlwright@ieee.org, {jason.larsen,miles.mcqueen}@inl.gov

***Abstract*—Software vulnerabilities are an important part of the modern software economy. Being able to accurately classify software defects as a vulnerability, or not, allows developers and end users to expend appropriately more effort on fixing those defects which have security implications. However, we demonstrate in this paper that the expected number of misclassified bugs (those not marked as also being vulnerabilities) may be quite high and thus human efforts to classify bug reports as vulnerabilities appears to be quite ineffective.**

We conducted an experiment using the MySQL bug report database to estimate the number of misclassified bugs yet to be identified as vulnerabilities. The MySQL database server versions we evaluated currently have 76 publicly reported vulnerabilities. Yet our experimental results show, with 95% confidence, that the MySQL bug database has between 499 and 587 misclassified bugs for the same software. This is an estimated increase of vulnerabilities between 657% and 772% over the number currently identified and publicly reported in the National Vulnerability Database and the Open Source Vulnerability Database.

***Index Terms*—software vulnerabilities, vulnerability estimation, bug reports, misclassification**

I. Introduction

Software vulnerabilities are an important aspect of managing current and future information technology infrastructures. New vulnerabilities are discovered in software products every day and publicly reported through a variety of publicly accessible vulnerability databases. What is not known, however, is whether the number of publicly reported vulnerabilities for a software product can reasonably be used as a stand-in for the security of the product. Without knowing this information, risk estimates for individual products are problematic.

The definition of software vulnerability for this paper comes from Krsul [1] and Ozment [2]: "an instance of [a mistake] in the specification, development, or configuration of software such that its execution can violate the [explicit or implicit] security policy." All software defects are not vulnerabilities by this definition, but vulnerabilities are a subset of software defects; the important factor is whether the security policy may be violated. As discussed in Section II-E, we further constrain this definition to err on the side of conservative estimates of the yet to be identified vulnerabilities.

This paper describes an experiment, performed on a popular open source product, to estimate the number of bugs misclassified as not being vulnerabilities. A subset of the product's bug reports were selected for detailed scrutiny. The selected bug reports and the associated source code were then carefully analyzed to determine what portion of the selected bugs had been misclassified as not vulnerabilities. Based on these results, we extrapolated to the full population of bug reports in order to provide an estimate of the total population of misclassifed bugs (those which should have been identified as vulnerabilities). Based on these estimates, we find that there are between 657% and 772% more vulnerabilities than the 76 that are currently documented for the product in the National Vulnerability Database (NVD)[1] and the Open Source Vulnerability Database (OSVDB)[2]. (Note: We found that the few MySQL vulnerabilities listed in the OSVDB all mapped to vulnerabilities found in the NVD, so for the rest of this paper we will simply refer to vulnerabilities reported through NVD).

As a consequence, we conclude that there is reason to believe that the number of publicly reported vulnerabilities in a software product is far less than those that have been discovered as bugs but misclassified as not vulnerabilities. This leads us to the conclusion that human efforts in identifying bugs as vulnerabilities may not be very effective, and that using the number of reported vulnerabilities as a measure of a software product's relative risk is highly questionable.

[1]National Vulnerability Database, http://nvd.nist.gov/
[2]Open Source Vulnerability Database, http://www.osvdb.org/

The remainder of this paper is organized as follows. Section II describes the experimental goals and setup including a description of the product we evaluated, its bug database, and the evaluation process for each bug. Section III describes the results of the experiment, and Section IV includes further analyses and discussion. Related work is described in Section V, and Section VI provides the conclusions.

II. Experimental Goals and Setup

The goal of our experiment was to determine if the total number of discovered vulnerabilities for individual software products (including those bugs misclassified as not being vulnerabilities) is much larger than the currently reported number of vulnerabilities in the NVD.

If the number of misclassified bugs is large then many analysis based on reported vulnerabilities, such as the half life of vulnerabilities [13] or rate of vulnerability discovery [15], are called into question. The experimental results may also suggest a need for improved bug classification in order to provide much stronger vulnerability data sets for vulnerability research; reduce vulnerability attack time windows; and allow more effective risk comparisons between software products.

Further, the results may also indicate whether or not the largely human process of identifying which bugs are also vulnerabilities is effective, and perhaps suggest a need for improved bug triage tools to aid more effective identification of vulnerabilities by software developers. In addition, a significantly large number of misclassified bugs could potentially provide new insight into vulnerability attributes and an associated opportunity for improvement in vulnerability identification tools such as static analyzers.

Each of the sub-sections below describes a part of the process used to conduct the experiment: product selection (II-A), MySQL bug database overview (II-B), selected subset of MySQL server software (II-C), MySQL bug scoring process and results(II-D), and determining the number of misclassified bugs (II-E).

A. Product Selection

The first step in this experiment was to select a software product for evaluation. The ideal product would satisfy a number of properties. These properties consist of:

- 1 – product pervasiveness,
- 2 – large number of announced vulnerabilities,
- 3 – source code availability,
- 4 – publicly accessible bug reports.

Pervasiveness was desired so that we could be confident that the product's code base, and its associated bugs, had received significant security attention. A fairly large number of publicly announced vulnerabilities was needed to both confirm that the code base had received security scrutiny, and to allow credible comparisons relative to the number of new vulnerabilities we identified (if any). The product's source code needed to be available so that we could reproduce the bug and effectively evaluate its security impact. And, of course, the bug reports were required to be publicly accessible so that we could reassess those bugs classified as not vulnerabilities.

TABLE I
Number of first published vulnerabilities and bug reports by year for MySQL

Year	Vulnerabilities	Bugs
1998	1	—
1999	0	—
2000	3	—
2001	6	—
2002	8	15
2003	5	2225
2004	9	5354
2005	11	8282
2006	14	8295
2007	14	7558
2008	7	7596
2009	7	7392
2010	6	8241
2011	16	—
Sub Total	107	54958
No such bug		9
No access to bug		4480
Total bug reports		59447

As discussed below, the MySQL product reasonably meets each of the four desired properties.

First, MySQL is widely deployed. It is part of the "LAMP" configuration of Linux (Linux, Apache, MySQL, and PHP) which is used on a large number of deployed web sites.

Second, at the time this experiment was conducted, the National Vulnerability Database listed 107 vulnerabilities for MySQL announced from 1998 through 2011. The number of vulnerabilities published per year is shown in the second column of Table I.

Third, the source code for all versions of the MySQL software was available for analysis. Although MySQL does not maintain old software source code on their servers, it was possible to reconstruct the state of the code at each release using the available source code management system (Bazaar).

Fourth, although we would have preferred to have all bug reports available for evaluation this did not appear feasible after a quick evaluation of a number of products. Fortunately, over 92% of MySQL bug reports were publicly accessible and this seemed to reasonably meet the desired property.

Consequently, we selected the MySQL software product for evaluation.

B. MySQL Bug Database Overview

A public bug database was setup for MySQL on September 12, 2002, and contains 59447 unique bug identifiers as of the end of 2010. The distribution of bug reports by year is shown in the third column of Table I. Of the 59447 bug reports, 4480 (7.5%) are not accessible to the public, and 9 (0.02%) simply do not exist.

The non-public bug reports perhaps contain either private customer data or information that may contain information regarding vulnerabilities. Since we have no information regarding these bugs, we discarded their bug identifiers from our study. This left 54958 valid bug identifiers for consideration.

Unfortunately, no small team can examine that many bug reports with sufficient diligence to credibly identify those bugs which are also vulnerabilities. The analysis required is too detailed and the rate of vulnerability occurrence is too small. For instance, during the years 2002 to 2010 (Table I), the ratio of reported vulnerabilities to bug reports is approximately 0.0018 (about $1 : 556$). For this and a few other reasons, as explained below, we decided to further narrow our evaluation focus to a subset of MySQL server software.

C. Selected Subset of MySQL Server Software

We decided to focus on bugs affecting the MySQL server itself (not client programs, support scripts, or third party applications) based on matching the category field in the bug database. The decision to exclude client bug reports was made because in a given installation, a wide variety of client applications may be used to access the server and we wanted to evaluate bugs in the most commonly deployed aspects of the package (the part of the package with the most potential impact on end-users).

We discarded bugs relating only to release 3.X of the database server software because the bug database barely covers the development of this branch; the final release of the 3.X branch was version 3.23.58 on September 11, 2003 (slightly less than a year after the bug database was created).

We also discarded the bugs only affecting the "telco" and "Falcon" branches of MySQL server software. Neither of these branches were released officially and thus vulnerabilities in these branches would not have wide spread implications and the software may not have undergone a rigorous security vetting.

The *telco* branch was a fork of MySQL 5.1, integrating the MySQL Cluster technology of highly available distributed operation. At least some of the code has since been integrated into official releases so only bugs in the early development branch are excluded from this study.

The Falcon storage engine was a part of the MySQL 6.0 branch which had only one "official" release. Since the merger with Oracle, this development branch has been abandoned [3]. Thus all bugs in this branch are excluded from this study.

These reasonable exclusions allowed us to focus on the four major development branches of MySQL server (4.X, 5.0, 5.1, and 5.5) all of which were formally released for general use; not on development branches which may have been publicly available but not intended for production environments (so called alpha and beta releases); and overlap the time period covered by the bug database.

Applying the above exclusions to the MySQL bug database eliminated 24230 (44.1%) of the 54958 bugs leaving 30728 bugs still under consideration (second column of Table II). We also applied the same exclusions to the MySQL vulnerabilities listed in the NVD and additionally discarded the few disputed vulnerabilities. This process resulted in the elimination of 31 of the 107 vulnerabilities in NVD, which left 76 reported vulnerabilities in the MySQL server software under evaluation (third column of Table II). Note that with these exclusions, the ratio of reported vulnerabilities to bug reports has increased to approximately 0.0025 (about $1 : 404$).

TABLE II
APPLYING RULES TO BUGS/NVD CVES.

	Bugs	CVEs
Starting	54958	107
Non-server	20893	13
Wrong version	3337	15
Disputed	—	3
Remaining	30728	76

D. MySQL Bug Scoring Process

In the remaining set of 30728 bug reports under consideration we did not know how many, if any, were unidentified vulnerabilities. Since we still had too many bugs to carefully evaluate, the decision was made to create a scoring system intended to indicate the likelihood that a bug might be misclassified and actually be a vulnerability. The scoring system was created by two expert vulnerability researchers and involved a two step process.

In the first step, our experts specified a set of text strings which, if found in a bug report, might indicate the bug was more or less likely to be a vulnerability (e.g. a bug involving illegal instruction exceptions). The set of text strings can be found in column one of Table III.

In the second step, each bug's score was initialized to 0 and the bug report was then searched for the specified text strings. When a text string was matched the associated weight of the text string was added to the bugs overall score. A given bug report could have matches with zero or more text strings. The full listing of the text strings assigned weights can be found in the second column of Table III. The table also shows in column three how many of the 30728 bug reports were found to have each of the associated text strings.

Before applying the scoring system to the 30728 bugs under consideration it was tested to assess potential effectiveness. The set of bug reports associated with publicly identified MySQL server vulnerabilities were scored, and then the set of remaining bug reports which were not identified as vulnerabilities were scored. These two sets of scores were then compared with the expectation that the scores of the publicly identified vulnerabilities would be higher than the scores of the bugs which had not been identified as vulnerabilities.

1) Scoring System Test: To assess the effectiveness of our scoring system, we examined the references of the 107 known CVEs for MySQL server (described in Section II-A). Specifically, we examined each known vulnerability for references to specific MySQL bug reports. We also followed references from the NVD to other vulnerability databases (OSVDB, ISS Xforce, etc.) looking for references to bug reports. We also examined the MySQL bug database for references to CVEs. In all, we found 74 mappings from bug report to CVE and/or vice versa (summarized in the top half of Table IV).

We then proceeded to exclude some of the 74 bugs for a variety of reasons. We excluded bugs that were unrelated to

TABLE III
TEXT STRING WEIGHTING AND OCCURRENCE COUNTS

Text Strings	Weight	How Many	Description
submitted $<$ Jan 1, 2003	-100	10	Bugs submitted before Jan 1, 2003
'%signal 11%', '%sig=11%', '%egfault%', '%handle_segfault%'	+100	1755	Reports showing signal 11 (SEGV)
'%signal%', '%sig=%'	+50	2433	Reports showing POSIX signal aborts
'%pthread_kill%'	+50	342	Calls to `pthread_kill()`
'%signal 6%', '%sig=6%'	-50	524	Calls to `abort()`
'%__assert_fail%'	-20	326	Calls to `__assert_fail()`
'%-I../../..%'	-50	33	Compiler errors
'%write_core%'	+100	312	Calls to `write_core()`
'%corruption%'	+100	967	Reports mentioning "corruption"
'%deref%'	+100	82	Reports mentioning "deref"
'%double free%'	+120	45	Double free assertion failures
'%Error::%'	+100	9	Calls to error routines
'%exploit%'	+120	73	Reports mentioning "exploit"
'%gcc%'	-20	1134	GCC errors
'%GDB is free software%'	+100	46	GDB backtraces
'%signal 4%', '%sig=4%'	+200	9	Illegal instruction traps
'%mysqldump%'	-50	1171	Uses of "mysqldump"
'%0x000000%'	+20	977	NULL addresses in backtrace
'%pointer%'	+20	2041	Mentions "pointer"
'%raise%'	-50	747	Raises exception
'%main_security_ctx%'	+100	7	Uses of `main_security_ctx()`

TABLE IV
BREAK DOWN OF CVE/BUG IDENTIFIER MAPPINGS

Mappings	Number
CVE $\Rightarrow$ bug	26
bug $\Rightarrow$ CVE	9
CVE $\Leftrightarrow$ bug	39
Total References	74
Exclusion	
not MySQL related	3
duplicate (CVE $\Rightarrow$ bug)	2
third party	4
disputed in NVD	3
client application	4
no access to bug	5
wrong version	3
Total Excluded	26
Remaining	48

TABLE V
TEST OF SCORING SYSTEM PREFERENCE FOR VULNERABILITIES.

		score	
population	N	mean	median
mapped	48	75.83	0
not mapped	30680	12.23	10
Total	30728		
Wilcoxon W	1006742		
p-value	2.145×10^{-10} (one tail)		

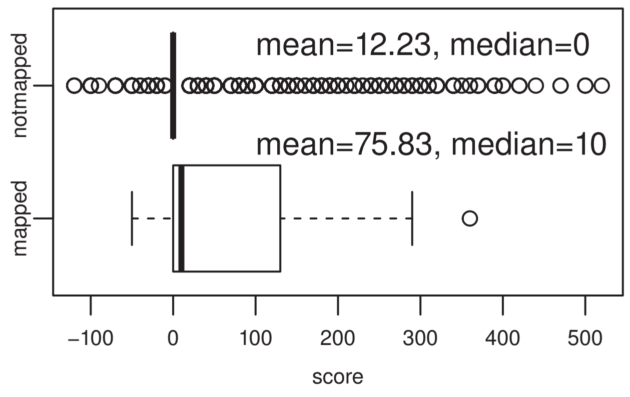

Fig. 1. Distribution of bugs mapped to vulnerabilities and otherwise.

MySQL (e.g. a CVE identifier appears in a Perl version string in bug 19532). Multiple CVEs (CVE-2008-4097, CVE-2008-4098, and CVE-2009-4030) point to MySQL bug 32167 so we excluded the latter two mappings. We also excluded CVEs that reference bugs for which we do not have access (e.g. CVE-2010-3838 maps to bug 54461 which is not accessible to the public). All of these exclusions are summarized in the bottom half of Table IV.

The scoring system as described in Section II-D was then applied separately to the 48 bug reports for identified vulnerabilities, and the bug reports for which there is no associated vulnerability. A one-sided Wilcoxon Rank Sum Test of the two populations was performed. We are able to reject the hypothesis that the medians of the two distributions are equal (Table V). Our scoring system scored bug reports significantly higher for known vulnerabilities than for other bugs. This means that on the surface at least, the scoring system shows some preference for vulnerabilities. Figure 1 shows the distribution, mean, and median for both populations.

We proceeded to apply the scoring system to the MySQL server bug reports under consideration.

2) Scoring System Results for MySQL Server: The application of the bug report scoring system produced the distribution of scores shown in Figure 2. The vast majority of scores fall in the interval $(-50, 0]$. Figure 3 zooms in on the distribution of the 4252 bugs with scores greater than 0.

Unfortunately, 4252 (13.8% of the 30728 server bugs under consideration) was still too many to evaluate. So we divided

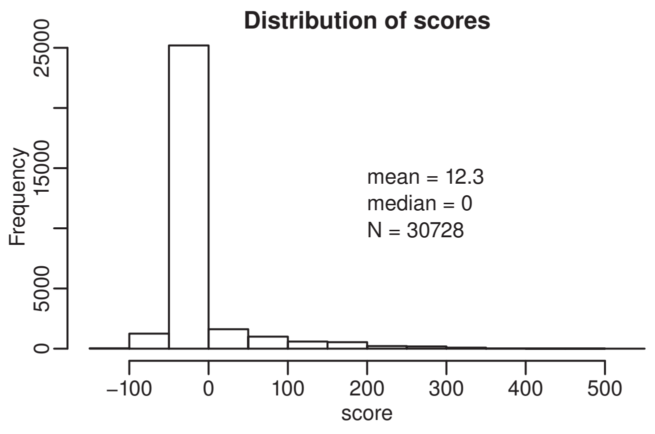

Fig. 2. Distribution of MySQL server bug scores

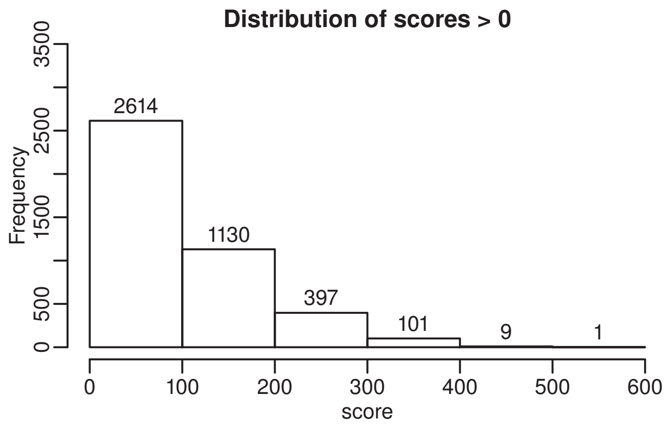

Fig. 3. Distribution of MySQL server bug scores greater than 0

the bugs into four disjoint groups based on score (second column of Table VI). We then estimated the percent of misclassified bugs (i.e. vulnerabilities) within each group.

E. Determining Number of Misclassified Bugs

We separately sampled the population of bugs in each of the four groups mentioned above in Section II-D2 except for Group 3 where the population was small enough to examine all bugs in the Group. The population size and sample size for each group is shown in the third and fourth columns of Table VI respectively. Each of the sampled bugs was carefully analyzed to determine if it was also a vulnerability.

TABLE VI
EXAMINED BUG REPORT TOTALS

Group	Score	Total Size	Sample Size
0	≤ 0	26476	160
1	$(0, 100]$	2614	46
2	$(100, 400]$	1628	55
3	> 400	10	10
Total		30728	271

In our analyses we assumed that the adversary would be able to execute arbitrary queries as an authenticated user. The reason for this assumption is the fact that a typical deployment of MySQL involves creation of database and user, and then web software is used to provide access to the database. The created user often has full access to the database.

Creation of a reliable exploit for a vulnerability requires a significant amount of work. For the purposes of this experiment, we decided to stop the evaluation short of this point. Instead, using terminology similar to Microsoft's Exploitability Index (EI) [4], we stopped at the point of determining whether an exploit was likely to be possible or not. The Microsoft's EI can be one of three values[3]:

- 1 – Consistent exploit code likely,
- 2 – Inconsistent exploit code likely, and
- 3 – Functioning exploit code unlikely.

To error on the side of conservative estimates, we did not include in our vulnerability counting and estimations those potential vulnerabilities for which we believed exploit code was unlikely (EI 3).

Similarly, to error on the conservative side and also proactively dampen potential criticisms that we primarily found uninteresting vulnerabilities, we decided to not include Denial-of-Service vulnerabilities in our counting and estimation. For instance, a basic NULL pointer dereference is, without additional resource manipulation (e.g. memory), not exploitable [5]. Dereferencing a NULL pointer will simply cause the program to crash. Such bugs, by themselves, can be used to do little else than create a Denial-of-Service. In brief, we only counted those vulnerabilities which are likely to allow violations of security policy and/or arbitrary code execution.

To summarize the overall conservative approach we took for determining whether a bug was a vulnerability, the bug:

- must affect the MySQL server software (client programs out of scope),
- must be in a released version of the software,
- must likely be susceptible to exploit code,
- must not be a denial-of-service.

III. Experimental Results

After careful evaluation of the selected bug reports from each scoring group, we were able to identify 12 bug reports that had been misclassified as not vulnerabilities. Without even extrapolating to the entire bug population, this represents a 15.8% increase over the current publicly reported vulnerabilities for the MySQL server code under evaluation. After extrapolation to the total set of MySQL server bug reports, we estimate that there are between 499 and 587 bugs which have been misclassified. This represents an estimated increase in vulnerabilities ranging from 657% to 672%.

In Section III-A we describe the evaluation steps for the bugs sampled from each scoring group. Each step represents the identification of one or more bug attributes. In Section III-B the empirical results from examining the sampled bug reports to determine the number of vulnerabilities present

[3]The Exploitability Index was "clarified" by Microsoft on December 13, 2011 and the definitions used here are the original definitions.

are provided. In order to be sensitive to the concerns of end users, we do not describe the newly identified vulnerabilities. However, the bug identifiers for the sampled bugs within each scoring group are provided in the Appendix in the hope that other research groups will be motivated to duplicate and extend our research. In Section III-C we then extrapolate from these empirical results to determine the statistical bounds on the number of misclassified bugs (i.e. vulnerabilities yet to be identified as such) in MySQL server code.

A. Vulnerability Evaluation of Sampled Bugs

The number of sampled bugs from each bug report scoring group are presented in the third column of Table VII. We instituted a maximum four step process for evaluating each of the sampled bugs.

TABLE VII
NEWLY IDENTIFIED VULNERABILITIES BY SCORING GROUP

Group	Score	Sampled	Vulnerabilities
0	≤ 0	160	1 (0.6%)
1	$(0, 100]$	46	4 (8.7%)
2	$(100, 400]$	55	5 (9.1%)
3	> 400	10	2 (20.0%)
Total		271	12 (4.4%)

The first evaluation step was to read each sampled bug report and determine if the bug was either a feature request or assessed to really not be a bug. A ”feature request” attribute, column two of Table VIII, is simply a request from a user to modify software that does not currently cause any unintended behavior of the server (e.g. Bug #26602 where a user suggests using a single lock variable instead of several). Outside of Group 0 (lowest scored vulnerabilities) feature requests were not found.

The “not a bug” attribute, column three of Table VIII, indicates that while the user reporting the bug believes the observed behavior is abnormal, it is not considered to be so by the MySQL developers. For example, bug #29033 describes a user expectation problem; the database was behaving as intended but not as the user hoped. This bug attribute was only found in the sampled bugs of the lowest two scoring groups.

All sampled bugs identified as a feature request or not a bug were set aside with no further analysis devoted to them. This first step was relatively quick and easy.

The second evaluation step was to determine if we could reproduce the bugs which had not been excluded in step 1. The “not reproducible” attribute, column four of Table VIII, describes bugs that could not be reproduced by the authors. In the case of bug #10918, a machine running SCO UNIX was not available to the authors, and for bug #24429, there was simply not enough information provided in the bug report to reproduce the problem described by the submitter.

Those sampled bugs we could not reproduce received no further analysis. This second step varied in difficulty with some bugs easy to reproduce; some bugs difficult to reproduce; and some which we were unable to reproduce.

The third evaluation step was to review the sampled bugs remaining after the first two steps and determine if they represented NULL pointer dereferences, column five of Table VIII. As described in Section II-E, these bugs are generally not exploitable without additional resource manipulation. So if the bug was a NULL pointer dereference then further analysis was done to determine if the bug could be exploited through some manipulation of memory such as is done in heap spraying [6]. This third step generally required a significant investment of time.

In the fourth evaluation step all remaining sampled bugs, column five of Table VIII, received in depth vulnerability analyses. Each required an investment of time and expertise roughly similar to, or exceeding, that used in analysis of NULL pointer dereferences.

TABLE VIII
TYPES OF BUGS WITHIN EACH GROUP.

Group	Feature Request	Not A Bug	Not Reproducible	NULL Pointer	Deeper Analyses
0	8	12	10	3	127
1	0	1	8	1	36
2	0	0	23	8	24
3	0	0	6	1	3
Total	8	13	47	13	190

B. Newly Identified Vulnerabilities

In all, we identified 12 previously unidentified vulnerabilities in the MySQL bug database by first searching through the sampled bug reports and then examining the affected code. The process for doing this was laborious, requiring duplication of the relevant portion of the environment in which the bug was reported and source code analysis by an experienced vulnerability researcher.

Table VII shows the break down of how the newly identified vulnerabilities fell within each of the scoring groups described in Section II-D2. Most of the vulnerabilities fell in groups 1 through 3 (further evidence that the scoring system was effective); however, one vulnerability was found in Group 0 (score ≤ 0) after analyzing 160 bug reports in that Group.

C. Extrapolation to Entire Bug Population

To extrapolate to the entire population of MySQL server bugs under consideration, we ran a simulation of 500,000 experiments using a population size of 30,728 bugs stratified as defined in Table VI. Within each strata, random Bernoulli trials were conducted with a probability of success equal to the probabilities in Table VII. The expected number of bugs that are also vulnerabilities ($E(y)$) can be calculated in a straightforward manner given $N = 30728$, $x_i = (1, 4, 5, 2)$, $n_i = (160, 46, 55, 10)$, and $N_i = (26476, 2614, 1628, 10)$ and Equation 1.

$$E(y) = N \sum_{i=0}^{3} \frac{N_i x_i}{N n_i} \approx 543 \qquad (1)$$

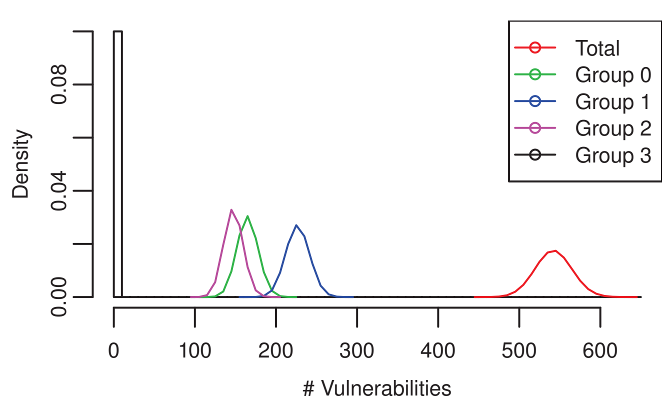

Fig. 4. Simulated probability density functions

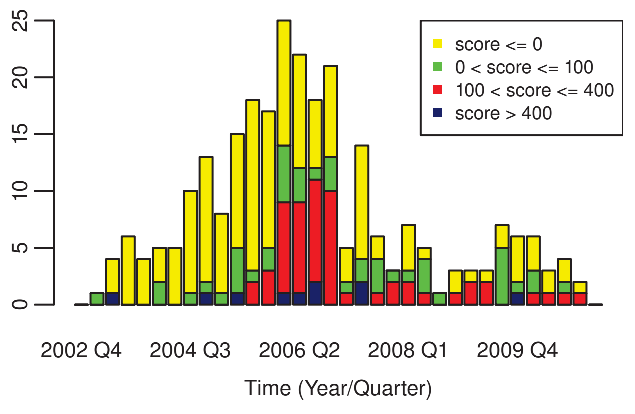

Fig. 5. Time distribution of examined bugs

However, the simulation allows us to combine the mixture of stratified binomial experiments into a single experiment and to obtain a 95% confidence interval on the number of remaining vulnerabilities. Our simulation resulted in a mean number of vulnerabilities of 543 and a 95% confidence interval $(499, 587)$. Figure 4 shows the PDFs obtained from the simulation for each group and the aggregate.

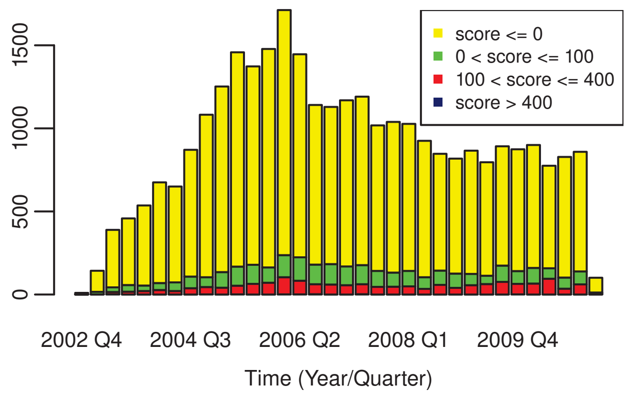

Fig. 6. Time distribution of all server bugs

IV. General Analyses and Discussion

In this section, the newly identified MySQL vulnerabilities (in addition to those previously reported) are analyzed from two different perspectives: distribution across time and by CVSS score (Section IV-A). We then present an estimate of the number of machines on the Internet which potentially could be affected by the new vulnerabilities we identified (Section IV-B).

A. Data Analysis

To check our sampling mechanism, the time distribution of the sampled bugs is shown in Figure 5, and the distribution of all MySQL server bugs is shown in Figure 6. The two distributions share a similar shape until Q2 of 2006, after which the sampled bugs have a smaller relative population. This change in overall shape is due to the stratified bug sampling based on each bug scoring group.

We also applied CVSS (Common Vulnerability Scoring System) scores to our identified vulnerabilities and their time distribution is shown in Figure 7. Most of our newly identified vulnerabilities were scored at 6.0 (medium severity). While scoring the new vulnerabilities, we gave them partial scores for the confidentiality, integrity, and availability dimensions (CIA dimensions). This reflects a conservative approach and is justified because we did not attempt to generate “weaponized” exploits for the vulnerabilities found.

Figure 8 shows the distribution of CVSS scores for known vulnerabilities from the NVD. The new vulnerabilities we identified as a result of this study fall above the median CVSS score, yet below the maximum score of 10. If we ranked the CIA dimensions as ’complete’ instead of ’partial’ compromise, as we would expect them to be for weaponized exploits, then even the lowest CVSS score for our vulnerabilities would be 7.1 (this is despite requiring authenticated access and having medium access complexity).

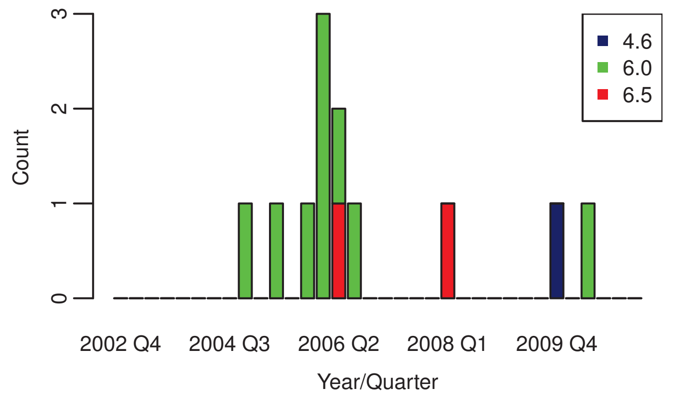

Fig. 7. Time distribution of examined bugs and CVSS score

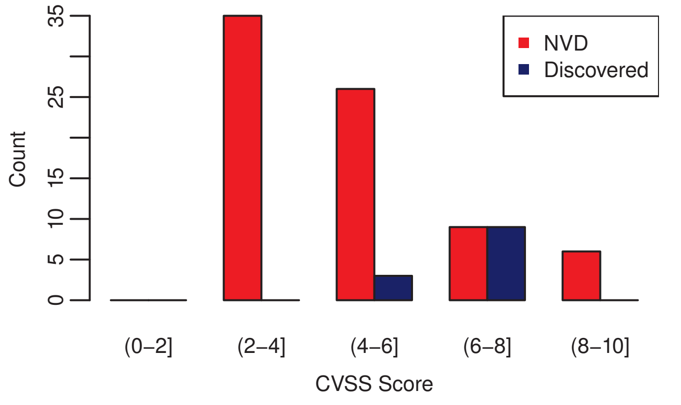

Fig. 8. Known vulnerabilities and new vulnerabilities distribution by CVSS

B. Affected Machines

One might argue that the newly identified vulnerabilities primarily occur in older versions of the software and thus the number of affected machines is likely to be small. The assumption is that most MySQL server installations will have been upgraded to the newest version of the server software. To assess this argument, we used the Shodan computer search engine[4] which regularly scans the Internet collecting banner strings from applications. To ensure freshness of the results, the query was limited to machines discovered after Jan 1, 2012. Fortunately, MySQL reports its version string immediately after a client connects. The version string takes the following form: `5.0.16-standard-log` where 5.0.16 is the version number and the remaining portion are compiled in options.

In all, Shodan found 1.5 million hosts that report a MySQL version string. Combining that information with the versions affected by our newly identified vulnerabilities, we were able to count the number of affected machines identified by Shodan. The results are shown in Table IX for 2 of the 12 vulnerabilities identified as a result of this study.

These 2 vulnerabilities represent the range of values for the number of machines affected by each of the 12 vulnerabilities. Interestingly, we are unaware of any credible reason to expose a MySQL server directly to the Internet; the database server provides the ability to restrict, by IP address, which machines can connect and any modern firewall (host based or hardware based) could be used to prevent similar connections. So it seems reasonable to assume that the 1.5 million hosts is a limited subset of machines actually running the MySQL server versions of interest in this study.

This analysis suggests that even vulnerabilities in older version of the MySQL server software may still be of potential impact to a large number of machines.

C. Study Limitations

The primary threat to validity comes in generalizing the experimental results from MySQL server software to a wider population of software products. This work examined the one software package and thus we cannot be confident that such high estimates for the number of misclassified bugs holds true for all software products. There is some evidence however to suggest that at least one other product has a similar incidence of misclassification. In [7], Arnold, et al. examined the Linux kernel for hidden impact bugs (bugs reported and fixed long before they were identified as vulnerabilities). They found a surprising number of bugs misclassified in this fashion. [8] found similar results for newer bugs reported in the Linux kernel and also for MySQL bugs. This later work also indicates that the incidence of these hidden impact bugs appear to be increasing with time. Consequently, there is at least some preliminary evidence that the problem of misclassified bugs may actually be getting worse, rather than better, with time.

To improve our misclassified bug estimates, increasing sample size and number of newly identified vulnerabilities would be beneficial. This could, in principle, be accomplished by increasing the number and diversity of expert vulnerability researchers devoted to the vulnerability identification effort. We primarily used one researcher extensively and a second one intermittently to identify and validate new vulnerabilities. Neither researcher was expert in database vulnerability discovery. Further, even these two experts were strongly constrained by time and availability. Consequently, there is some reason to suspect that the estimates we made in this case study for the total number of misclassifed bugs have errored on the conservative side.

In addition, the vulnerability liklihood scoring process for bugs described in Section II-D could likely be improved in several ways. Examination of Table III reveals a bias towards UNIX-like systems and this is due to the professional expertise of the authors: we simply know more about vulnerabilities affecting UNIX-like systems than, for example, the Microsoft Windows family of operating systems. Getting an accurate count of the number of bugs affecting UNIX-like operating systems versus other operating systems referenced in the MySQL bug database is complicated by the fact that the field capturing affected operating system is free-form and user-specified, so it is difficult to measure the effect of this sampling bias.

The text strings in Table III are used for static lexical matching and this has the possibility of lexically divergent but semantically equivalent phrases for the same idea. Because of its static nature, the bug scoring process is not resilient to the introduction of new phrases for the same idea and the retirement of old phrases (neologism). Note however, that we attempted to choose phrases commonly used to describe

TABLE IX
NUMBER OF AFFECTED MACHINES BY VULNERABILITY.

Vuln.	# affected	%
V1	17790	1.16%
V2	413065	26.92%
Total	1534188	

[4]SHODAN--ComputerSearchEngine,http://www.shodanhq.com

vulnerability precursors over the time period of the study.

V. Related Work

Our approach is unique primarily because we examined bug report databases to narrow the search for bugs not (yet) classified as vulnerabilities. Most previous work in the area of software vulnerability estimation has generally focused on the publicly reported vulnerabilities in software: known through online vulnerability databases such as the NVD or OSVDB.

For instance, Rescorla attempted unsuccessfully to fit the various statistical models (linear and Goel-Okumoto software reliability model) to vulnerability data from what is now the NVD [9]. He examined four operating systems: Windows NT, Solaris 2.5.1, RedHat Linux 7.0, and FreeBSD 4.0.

Likewise, using data including publicly reported vulnerabilities, Alhazmi, et al. proposed several statistical models for predicting vulnerability discovery: an S-shaped model, time-based logistical model, and an effort-based model [10]–[12]. Their primary work focuses on application software, specifically Microsoft Internet Information Server (IIS) and the Apache HTTP server. In this one way, our work is similar (we also focus on application software not on the underlying operating system).

Ozment and Schechter [13] examined the OpenBSD operating system and defined the term foundational vulnerabilities to describe those vulnerabilities present in the code base that did not change over the course of the study. They found that the rate of vulnerability reports in foundational code was decreasing over time (a potential contradiction with Resorla [9]).

In other work, Ozment compiled a database of vulnerabilities in OpenBSD from various sources (NVD, Bugtraq, OSVDB, etc.) and attempted to apply software reliability models to software security questions and had modest success in predictive accuracy [14]. However, his models were based on the rate of vulnerabilities identified and reported publicly. Given the predicted number of misclassified bugs presented in this paper, it is not reasonable to assume that Ozment's models reasonably describe the actual number of vulnerabilities which have been discovered. Many may have been misclassified as not being vulnerabilities.

In [15], it is argued that vulnerabilities have different properties than software defects (at least during the first phase of a product's existence). However, other research suggests that many vulnerabilities do, in fact, show up as bug reports and are not classified as vulnerabilities until some time later [7], [8]. Further, in this paper we demonstrate that a very significant portion of bug reports remain misclassified as not also being vulnerabilities. Thus it may be that the software properties of many vulnerabilities are, in general, similar to that of other software defects.

VI. Conclusions

The estimation of the number of vulnerabilities in a software product is an important factor for assessing its quality. Many previous attempts at estimation have relied on publicly available vulnerability databases or software reliability models derived from these databases. Our work calls into question the quality of estimates based on this type of information.

We conducted an experiment to determine if the number of publicly identified vulnerabilities was a good estimate of the total number of discovered vulnerabilities (including those discovered bugs which have been misclassified as not being vulnerabilities). Our experimental results provide some indication that even with the conservative approach outlined in this paper, the quantity of discovered vulnerabilities in software is considerably higher than what previous work would estimate for the same code base. Consequently, previous counts and estimates of vulnerabilities in a software product may be seriously flawed, and may be inappropriate for use in risk comparisons between competing software products.

Additionally, the results presented in this paper provide evidence that classifying bugs properly is a difficult task. Thus there is a need for both improved automated bug classifiers and a potential need for a deeper understanding of the software properties of vulnerabilities.

Future work will be to apply the approach described in this paper to one or more additional code bases to determine if the results hold. We encourage other research groups to join in this endeavor. Further, we have begun work to apply computational intelligence techniques to build a more robust bug classifier [8]. Eventually we hope to identify a large number of misclassified bugs, determine reasons for the misclassifications, and to identify new software attributes which can be used for vulnerability identification. The ultimate goal is to improve the security, and estimates of security, for software products as early in their life cycles as possible.

Acknowledgment

The authors would like to thank John Matherly of Shodan for helping to gather the number of affected machines.

References

[1] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, Purdue, May 1998. [Online]. Available: http://ftp.cerias.purdue.edu/pub/papers/ivan-krsul/krsul-phd-thesis.pdf

[2] A. Ozment, "Vulnerability discovery and software security," Ph.D. dissertation, University of Cambridge Computer Laboratory, 2007.

[3] S. M. Kerner. (2010, April) Oracle commits to mysql with innodb. Database Journal. [Online]. Available: http://www.databasejournal.com/features/mysql/article.php/3876206/Oracle-Commits-to-MySQL-with-InnoDB.htm

[4] (2008, October) Microsoft exploitability index. Microsoft. Last Updated: December 13, 2011. [Online]. Available: http://technet.microsoft.com/en-us/security/cc998259

[5] K. Johnson and M. Miller, "Exploiting the otherwise non-exploitable on Windows," Leviathan Security Group, Tech. Rep., May 2006. [Online]. Available: http://uninformed.org/?v=4&a=5&t=sumry

[6] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap taichi: Exploiting memory allocation granularity in heap-spraying attacks," in *Annual Computer Security Applications Conference (ASSAC '10)*. ACM, 2010, pp. 327–336.

[7] J. Arnold, T. Abbott, N. Elhage, G. Thomas, and A. Kaseorg, "Security impact ratings considered harmful," in *12th workshop on Hot Topics in Operating Systems*. USENIX, May 2009.

[8] D. Wijayasekara, M. Manic, J. L. Wright, and M. A. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *International Conference on Human System Interactions (HSI)*, Perth, Austrailia, June 2012.

[9] E. Rescorla, “Is finding security holes a good idea?” *IEEE Security and Privacy*, vol. 3, no. 1, pp. 14–19, January 2005.

[10] O. H. Alhazmi and Y. K. Malaiya, “Modeling the vulnerability discovery process,” in *International Symposium on Software Reliability Engineering*. IEEE, December 2005.

[11] ——, “Quantitative vulnerability assessment of systems software,” in *Reliability and Maintainability Symposium*, January 2005, pp. 615–620.

[12] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, “Measuring, analyzing and predicting security vulnerabilities in software systems,” *Computers Security*, vol. 26, no. 3, pp. 219–228, 2007.

[13] A. Ozment and S. E. Schechter, “Milk or wine: Does software security improve with age?” in *15th USENIX Security Symposium*. USENIX, July 2006, pp. 93–104.

[14] A. Ozment, “Software security growth modeling: Examining vulnerabilities with reliability growth models,” in *Quality of Protection: Security Measurements and Metrics*, ser. Advances in Information Security, D. Gollmann, F. Massacci, and A. Yautsiukhin, Eds. Springer, 2006, vol. 23.

[15] S. Clark, S. Frei, M. Blaze, and J. M. Smith, “Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities,” in *Annual Computer Security Applications Conference ACSAC*, 2010, pp. 251–260.

APPENDIX
IDENTIFIERS OF SAMPLED BUGS FOR EACH SCORING GROUP

For each of the four MySQL server bug scoring groups, we sampled a subset of the bug reports to undergo detailed analysis in order to determine if they were also vulnerabilities. In this Appendix we provide in Table X, Table XI, Table XII, and Table XIII the bug identifiers for all of the bugs we sampled and carefully evaluated from each scoring group. The bug identifiers provided are those used by the MySQL software team to uniquely identify each bug. Each of the 12 vulnerabilities we identified are highlighted in their respective Table.

We have begun the process of reporting our vulnerabilities to CERT/CC and Oracle (current owner of the MySQL codebase). As of this writing they have not been accepted or rejected as vulnerabilities by either party. However, it has been observed that as vulnerabilities are discovered, Oracle will likely disallow access to the corresponding bug report. If this should happen, the authors website[5] contains a mirror of the bug database as it existed during this research and will be made available to other researchers on a case by case basis.

We encourage other research teams to carefully evaluate each of these bug samples and validate, or invalidate, our results We further suggest that other research teams perform the same scoring process we developed, do their own sampling, followed by vulnerability identification. We will of course be interested in the results of these other efforts, including case studies of other software products, and would hope to find mechanisms for collaboration and sharing of intermediate data in addition to final results.

TABLE X
GROUP 3 (SCORE > 400) BUGS EXAMINED.

bugid	score	bugid	score	bugid	score	bugid	score
639	470	7289	500	10918	420	16550	500
19885	420	21658	420	21913	440	27752	470
28975	520	**48319**	420				

TABLE XI
GROUP 2 ($100 <$ SCORE ≤ 400) BUGS EXAMINED.

bugid	score	bugid	score	bugid	score	bugid	score
13090	200	13627	200	14086	270	15268	200
15924	200	**16218**	270	16298	180	16805	270
17001	270	17419	270	17535	270	17672	200
18649	180	19155	200	19210	180	19382	180
19727	270	20151	200	20258	270	20595	200
20979	200	21135	270	21651	170	21709	170
21851	200	**21927**	170	22244	170	22413	170
22440	170	22879	170	23075	170	23368	170
23506	170	23542	170	23944	170	24199	170
24211	170	24480	170	24502	170	27296	180
30562	180	31569	240	32431	240	33844	180
35272	180	36656	180	41733	240	42438	240
43827	320	44040	320	44886	320	51136	190
52711	190	55627	190	59111	190		

TABLE XII
GROUP 1 ($0 <$ SCORE ≤ 100) BUGS EXAMINED.

bugid	score	bugid	score	bugid	score	bugid	score
146	100	2490	100	2674	40	5435	70
6885	20	7981	100	9916	20	9949	100
10058	20	11226	20	13626	100	**13673**	100
15338	50	**16470**	20	16738	20	16739	20
17123	20	18037	100	19311	100	19626	50
20076	50	20823	100	23165	100	24482	100
25332	20	25908	100	28073	50	28421	100
29801	100	31214	50	31242	100	31916	20
34534	20	36156	20	36583	100	37408	20
38848	100	46321	20	46592	50	46782	20
47144	100	47280	20	47883	100	50457	20
50632	100	56281	100				

[5]http://www.thought.net/

TABLE XIII
GROUP 0 (SCORE ≤ 0) BUGS EXAMINED.

bugid	score	bugid	score	bugid	score	bugid	score
249	0	429	0	708	0	785	0
856	0	952	0	1010	0	1073	0
1076	0	1484	0	1879	0	1977	0
1993	0	2539	0	3089	0	3218	0
3446	0	3667	0	3852	0	3863	0
3925	0	4451	-20	4609	0	4752	0
4801	0	5059	0	5145	0	5248	0
5712	0	5811	0	5926	0	5985	0
6557	0	6600	0	6690	0	6932	0
7047	0	7170	-20	7223	0	7329	0
7443	0	7687	0	7873	0	7957	0
7996	0	8134	0	8357	0	8435	0
9650	0	9965	0	10083	0	10313	0
10537	0	10818	-50	10935	0	11017	0
11072	0	11549	0	11731	0	11774	0
11953	0	11997	0	12457	0	12501	0
12657	0	12702	0	12702	0	12778	0
12902	0	13059	0	13202	0	13370	-50
13499	0	13642	0	13845	0	14216	0
14543	0	14900	0	14906	-50	14953	-50
15068	0	15072	-50	15471	0	15599	0
15716	0	15745	0	16172	0	16179	0
16205	0	16215	0	16331	0	17020	0
17319	0	17894	0	18276	-20	18289	0
18347	0	18729	0	18859	0	19326	0
19421	0	19747	0	19861	0	19893	0
20074	0	20140	0	20214	0	20381	-50
21020	0	21088	0	21332	0	21487	0
21566	0	22457	0	23208	0	24002	0
24062	0	24302	0	24429	0	24532	-50
24663	0	24988	0	26287	0	26602	0
26613	0	27600	0	28166	0	28194	0
28571	0	28578	0	28788	0	28872	0
28940	0	29033	-50	29214	0	29579	0
29822	0	33613	0	34454	0	35146	0
35343	0	37133	0	40263	0	40737	0
41854	0	44923	-50	47318	0	47582	0
47902	0	48661	0	48691	0	49450	0
50412	-70	51468	0	52207	0	54645	0
54699	0	55686	0	56708	0	57564	0