B.Sc. in Computer Science and Engineering Thesis

Analysing the Effectiveness of Peer Code Review in Ensuring Software Security: An Empirical Study

Submitted by

Sheikh Nasif Imtiaz
201105019

Raisul Arefin Nahid
201105091

Supervised by

Dr. Anindya Iqbal

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

17 February 2017

CANDIDATES’ DECLARATION

This is to certify that the work presented in this thesis, titled, “Analysing the Effectiveness of Peer Code Review in Ensuring Software Security: An Empirical Study”, is the outcome of the investigation and research carried out by us under the supervision of Dr. Anindya Iqbal.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Sheikh Nasif Imtiaz
201105019

Raisul Arefin Nahid
201105091

CERTIFICATION

This thesis titled, **"Analysing the Effectiveness of Peer Code Review in Ensuring Software Security: An Empirical Study"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in 17 February 2017.

Group Members:

Sheikh Nasif Imtiaz

Raisul Arefin Nahid

Supervisor:

Dr. Anindya Iqbal
Assistant Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

ACKNOWLEDGEMENT

We are greatly thankful to our supervisor, Dr.Anindya Iqbal, Assistant Professor, Dept. of CSE, BUET, Dhaka, Bangladesh, for his guidance, encouragement and constant supervision which motivated us to take the challenge and complete the thesis. We are also greatly thankful to Dr.Amiangshu S. Bosu, Assistant Professor, Department of Computer Science at Southern Illinois University, USA. Our work was essentially an extension to his previous study, and him being continuously directing us essentially made it possible to shape up a comprehensive work.

Dhaka
17 February 2017

Sheikh Nasif Imtiaz

Raisul Arefin Nahid

Contents

List of Figures

List of Tables

ABSTRACT

Ensuring the highest security is a major challenge in today's software development.Hence, the quest for the optimum methodology which produces the best result in expense of the least resources. Peer code review is already a popular strategy which has a potential to fit our needs. The goal of this study is to analyze how effective peer code review could be in ensuring the minimal security threats in the final product, and what parameters should be considered while assigning reviewers for a code change. We analyzed 235,202 code review requests from 9 open source projects and identified 523 Vulnerable Code Changes(VCC). With our scientific approach, we proved that common security defects can be identified during peer code review and be fixed very quickly. Also by analyzing the large data set of VCCs that we developed, we found that the reviewers who identify the VCCs are generally more experienced than the authors. It gives us a key insight in how to efficiently handle the code review team. Also from our thorough auditing of the review requests, we noticed that the authors make a lot of trivial mistakes and a detailed discussion during the review helps them understand the issue in depth. So we also recommend the training of developers on common vulnerabilities and the practice of writing detailed explanations in review discussions.

Chapter 1

Introduction

1.1 Motivation

Ensuring a secure product is one of the major challenges for the Open Source Community. The code being accessible to all, anyone could analyze and figure out vulnerabilities in order to exploit. So ensuring defect-free code is the utmost priority in the community. But humans are error prone, and even the most trivial mistakes could arise from the most experienced developer. What's more unique to the case of security is that the vulnerable codes which can be exploited to compromise the integrity, authentication and availability of the program, do not generally hamper in the normal work flow of the program. Which means a separate vigorous testing is often necessary with the presence of domain experts. But such experts are scarce [1], and an extra step in the development cycle heavily increases the time and cost needed for the finished product. Hence the project manager has to efficiently allocate his resources and take smart decisions in order to optimize this process. Early detection of the vulnerabilities, choosing the best development method and allocating the right persons in the right duties could save a lot of time and money. This essentially motivates us to study the impact of *"Peer Code Review"* in identifying the vulnerabilities and how to assign the code reviewers to make this process the most efficient.

1.2 Objective

The objective of this study is *to use data from Open Source Software(OSS) projects to verify if peer code review is really effective in detecting the most common security vulnerabilities and how does the reviewers' experience co-relates with the possibility of identification of those mistakes. Hence, to help project managers take their decisions regarding to adopt peer code review in the development process, and to efficiently conduct it to get a robust, secured product* *in result.*

To facilitate this exploration, we coin a term “Vulnerable Code Change” (VCC) as “the change in the code that contain exploitable vulnerabilities, security wise”. There have been numerous approach to characterize and study such vulnerabilities [2, 3]. In our approach, we use the data gathered during the peer code review process, characterize them along with the type of discussion, and match them with the traditional classifications to prove the ability of peer code review in identifying the top vulnerabilities. In addition, we also try to estimate the experience of both the authors and identifiers of such VCCs and compare them to gain insights.

One major objective to study peer code review is that *code reviews happen soon after the code change is made [4]. Thus, vulnerabilities are identified in the earliest phase of the development cycle. Previous studies show that, the longer it takes to detect and fix a security vulnerability, the more that vulnerability will cost [5]. Therefore, eliminating vulnerabilities early via careful coding and reviewing should help reduce the cost of developing secure software.*

In our study, we take advantage of the documentation that code review tools have for the discussions between the developers and the reviewers upon various aspects of the code including potential security defects. When we could identify any such defect, the discussion, location and other related information that are available with the data, give us the possibility of gaining deeper insights about this vulnerabilities.

Another advantage of our approach is that we could gather vulnerabilities that were identified before the release unlike many data sets which contain vulnerabilities that were detected after the release. This gives us the opportunity to study them in the context of the initial development cycle, even before testing, and gain insights about how to make the overall process more efficient.

To achieve our goals, we followed a scientific method for analyzing 235,202 peer code review requests mined from 9 popular Open Source Software (OSS) projects. Each of these projects use the *Gerrit* code review tool, which captures the data necessary for our analysis. Based on an empirically built and validated set of keywords, we followed a three-stage systematic analysis process to identify 524 VCCs from our mined data.

1.3 Research Questions

The goal of our effort is to empirically analyze the types of vulnerabilities detected in code reviews, the characteristics of such VCCs , and the characteristics of the developers who are more likely to author these VCCs and also the reviewers who are more likely to identify them. From those data, we primarily focus on answering two Research Questions.

1.3.1 Identification of Vulnerabilities through Peer Code Review

RQ1: Do peer code review successfully identify the top security vulnerabilities?

This is a foundational question of our study. This information will help project managers understand what effects the addition of peer code review can have on the overall security of their products. Hence, giving insights to goals that mainly motivate us for this study.

1.3.2 The experience of Code Reviewers

Once we recognize the importance of the peer code review process, the next question that immediately arises is to how to improve it. Here we study if we need to assign code reviewers who are more experienced than the developers who wrote the code and requested the review. Or such assignment have no improvement over our performance. Therefore, we ask,

RQ2: Are the reviewers who identify the vulnerable code changes (VCCs) more experienced than than the authors who write it?

1.4 Thesis Organization

in Chapter 2, we discuss the major concepts related to our study. Then, we describe the methods and the way we generated our data set of vulnerabilities and discuss its authenticity in Chapter 3.

Chapter 4, illustrates the answers to our research questions based on our findings. While in Chapter 5, we discuss our contributions, implications of our findings, comparison with previous studies and future works that should be explored.

Chapter 6 briefly discusses some threats to validity of our findings. Finally, we conclude our study by discussing about possible future work in Chapter 7

Chapter 2

Background Study

2.1 Security Vulnerabilities

Vulnerability is a weakness in the program that allows attacks on the integrity, authentication, availability and information security of the system. As the software industry is booming, along with it the people trying to exploit those software are also increasing. Attackers could exploit this code for money, fun, fame or for other leverages. Although, sometimes higher security costs system performance, especially in the open source community, as we've mentioned, security is always the highest priority.

There are many types of vulnerabilities. Classifying them into separate groups is often not possible. Still, we closely follow the vulnerability databases and previous studies, as described in Section 2.2, and select the most common vulnerabilities to concentrate on in our study. They are-

Race-

The definition of a race condition is when two different execution contexts, whether they are threads or processes, are able to change a resource and interfere with one another. The typical flaw is to think that a short sequence of instructions or system calls will execute atomically, and that theres no way another thread or process can interfere. Even when theyre presented with clear evidence that such a bug exists, many developers underestimate its severity. In reality, most system calls end up executing many thousands (sometimes millions) of instructions, and often they wont complete before another process or thread gets a time slice.

Buffer Overflow-

Strictly speaking, a buffer overrun occurs when a program allows input to write beyond the end of the allocated buffer. But there are several implications to this definition. Basically if there's any way an attacker go beyond the buffer which is allocated by the program, and do manipulations (or even just read the unauthorized data), it is called buffer overflow.

Integer Overflow -

This phenomenon occurs when a result of an arithmetic operation such as addition or multiplication exceeds the maximum size of the integer type used to store it. The overflow could occur due to faulty type casting.

Improper Access-

Improper access is one of the most important vulnerability. The attackers often target the system this way as this is common in almost all the systems. Improper access occurs due to faults in access control. Access control consists of Authentication, Authorization, Accounting. This is often refereed as AAA security model.

Cross Site Scripting (XSS) -

Cross site scripting is an attack which is done by injecting malicious scripts into otherwise benign and trusted web sites. This attack may occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to another user. This may get viral and spread very quickly. The flaws that allow such attacks are well known and quite widespread. This might occur when user input is taken without using proper encryption.

Denial of Service (DoS) /Crash -

When an application crashes it takes time to reboot. So if an attacker finds a way to do some action which may lead to repeated restarting of the system, leads to a DoS attack. Any server may loose money, popularity or reputation due to such attacks. DoS attack is often carried out by flooding the target with traffic, or sending it information that triggers a crash. The main intention of the attack is to deprive the real users from using the service they intend to use.

Deadlock -

Deadlock is a situation where more than one programs try to use the same resource at the same time which leads to blocking each other from using the resource. This leads to a halting of the programs.

SQL Injection -

SQL injection often targets the naive coders. If a programmer takes input from the user and directly uses the input in his queries then this code is sql vulnerable. Because a tricky attacker can manipulate the input word into a sql statement which might give him more information than the programmer intended to. This is one of the most common attack for beginner hackers and preventing this is also quite easy.

Format String -

A Format String attack is an exploit which occur when the submitted data of an input string is evaluated as a command by the application. The attacker might execute code to read/write something he should not. This is a major security risk.

Cross-site request forgery -

Cross-site request attack, is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the website thinks trust-able. This attack is also known as one-click attack or session riding and abbreviated as CSRF (sometimes pronounced sea-surf) or XSRF,

Memory Leak -

This is an issue which occurs when the program does not releases the memory properly which it allocates. This phenomenon may occur when some object is created but not accessible by the running code. This is mainly dangerous for programs in kernel space, as kernel memory is normally limited and sensitive. Even in the user space, long-lived applications may continuously waste memory and at one point use up all the resources and get blocked or crash.

2.2 Formal List of Vulnerabilities

In order to serve the developers and the security practitioners, there exist many formal list, directories and database of software security threats. We discuss few in short in here.

CWE

Common Weakness Enumeration is a community developed list of software weakness types. It currently has a total 707 software weaknesses with a ranking and score attached to all of them.

OWSAP

The Open Web Application Security Project is an on-line community which also have its own listing of vulnerabilities, although more dedicated to web based applications.

NVD

NVD is the U.S. government repository of standards based vulnerability management data. Their database also reports history of many security threats [6].

Top Vulnerabilities

Based on the directories mentioned above, similar studies under similar domain [7], and our own evaluation on the selected projects, *we have enlisted some most common vulnerabilities to heavily concentrate on to investigate if peer code review can actually identify them and we also took some generic measures to have an idea about all other vulnerabilities overall as shown in*

Table 3.1.

2.3 Peer Code Review

The process in which developers review each other's code is called Peer Code Review. The benefit of peer code review is better code quality, efficiency and security. There is no coder whose coding skills can not be improved. By peer code review the knowledge of multiple codes come together resulting better code. Though this might also cause dispute among the codes. This is why often certain rules are set before peer code review for the best output.

After the code is written, it goes through a vetting process by the adapt collaborators before it is incorporated in the collaborator-maintained codebases. The idea of teamwork is also enforced by peer code review. Though it increases the work of the coders, the output is worth it. Such as:

- It enables to find better ways to solve a problem by providing multiple pairs of eyes.
- Ensures someone other than the coder is familiar with the code.
- The learning process is improved when the developers read others code.
- A good platform for knowledge sharing, because it enables users to discuss the goodness or badness of coding in a particular way.
- Peer code review encourages the coder to write better code because it's embarrassing to be caught in a silly mistake.

Besides, it reduces risk. Because coding always gives birth to more bugs. Any coder can be unaware about a small bug which might lead to a big issue in the future. But a peer code review reduces such risks. A developer cannot test all his codes. But a second eye is always helpful to find and fix bug as early as possible. For example if a bug is caught by the developer team a single more commit would be sufficient. But if the code is not tested and integrated with the bug, that may lead to hours of more works which could be avoided. Another question might be asked that why should the developers read others code carefully? It is because he knows that others will do the same for him. Peer code review also helps the coders to evaluate each others skills. It inspires them to write better code by learning from others. So, code review is not just about the process of review itself, but about learning and teaching.

2.4 Measuring Inter Rater Agreement

In statistics, inter-rater agreement, or concordance is the degree of agreement among raters. It gives a score of how much homogeneity, or consensus, there is in the ratings given by judges. It is useful in refining the tools given to human judges, for example by determining if a particular scale is appropriate for measuring a particular variable. If various raters do not agree, either the scale is defective or the raters need to be re-trained.

One popular method to measure agreement between exactly two raters is Cohen's kappa.

2.5 Statistical Tests

Shapiro-Wilk Test

To make a choice between t-test and Mann-Whitney U test, we need to know whether the data set follows normal distribution. The The Shapiro-Wilk test, proposed in 1965, calculates a W statistic that tests whether a random sample comes from (specifically) a normal distribution. We can use the R language to perform this test, and if the resultant *p-value* is under a threshold significance, we assume the sample don't follow a normal distribution.

Wilcoxon signed-rank test

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (i.e. it is a paired difference test). When performing the test, the sample drawn should be random, independent and ordinal in measure. Similarly like Shapiro-Wilk, We can use the R language to perform this test, and if the resultant *p-value* is under a threshold significance, we assume the two population are not equal and has some difference. Then to identify the sign of the difference, we measure the median of the two population.

Chapter 3

Data Set Generation

The Data Set Generation was the heaviest work of our study and should be regarded as a great contribution to the field. We followed the approach taken in a previous study [7] to get a final set of vulnerabilities from our chosen projects. The rationale behind our approach mainly depends on two points-

- OSS communities often provide detailed review comments and suggestions about potential defects [4].
- Searching through a validated set of keywords is effective in pointing out those review comments indicating vulnerabilities [8].

The stages of our search method is discussed in the following sections-

3.1 Stage 1 - Mining Code Review Repositories

We used the Gerrit-Miner tool developed in a previous similar study [7]. We used this tool to mine the Gerrit Repository of 9 Open Source projects. Each of these projects requires all code changes to be posted to Gerrit for review prior to inclusion in the project. We conducted a more detailed analysis on those nine projects (listed in Table 3.2).

3.2 Stage 2 - Building Keyword Set

The review comments were searched with a set of keywords as we want to collect the comments most likely indicating to vulnerable code changes. A classification of vulnerability was made and different words which may have related meaning to that specific class of vulnerability was

Table 3.1: Keywords to Search Common Vulnerability Types

Vulnerability Type	Keywords
Buffer Overflow	buffer, overflow, stack, strcpy, strcat, strtok, gets, makepath, splitpath, heap, strlen
Format String	format, string, printf, scanf
Integer Overflow	integer, overflow, signedness, widthness, underflow, wrap, truncate,
Cross Site Scripting	cross site, CSS, XSS, htmlspecialchar (PHP only)
SQL Injection	SQL, SQLI, injection
Race Condition / Deadlock	race, racy, deadlock
Improper Access Control	improper, unauthenticated, gain access, permission
Denial of Service / Crash	denial service, DOS, crash
Cross Site Request Forgery	cross site, request forgery, CSRF, XSRF, forged
Memory Leak	Memory Leak, Leak, Leaky, Memory, Disallocate
Common	security, vulnerability, vulnerable, hole, exploit, attack, bypass, backdoor, threat, expose, breach, violate, blacklist, overrun, insecure, fatal, danger, risk, steal, danglin, unsafe, leak, heap

identified. We took the keyword list used in [7], and revised it based on upon our own knowledge and experience. We had some keywords targeting to some specific security defect which we are primarily conctrating on our thesis, and common generic keywords to tackle general issues which we list in the *"Others"* category. The set of keywords is shown in Table 3.1.

3.3 Stage 3 - Building a Data Set of VCCs

This stage consists of 3 sub stages also based on a previous study [7]. First step is database search, which is discussed earlier.

3.3.1 Step 1: Database Search through Keywords

In this first step, we queried all the review comments and extracted the ones which contain at least one keyword at least once. Based on our observations, we consciously avoided some problematic words which have some of our keywords as a part of it but are irrelevant. Like "stackoverflow" is a word that contains both the keyword "stack" and "overflow" but is an irrelevant word to our objective, as it's only a refernce link to the stackoverflow.com site. Similarly "stackexchange", "grace", "trace" and such words were avoided which occur frequently in the comments, contains keyword as part of it, but are irrelevant. Finally from the 23502 comments that were mined from the 9 projects, we extracted 8360 for the next auditing step through this keyword search Table 3.2.

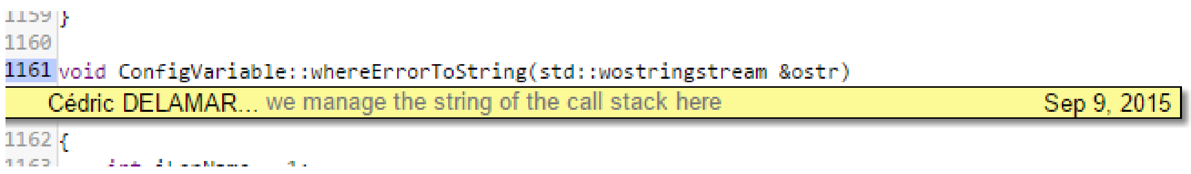
Projects	**Request Mined**	**Comments Extracted**
couchbase	48226	1293
gerrit go	10569	1735
gerrit review	15575	615
kitware	20104	186
libreoffice	20104	311
ovirt	50824	3212
scilab	17307	110
typo3	37609	552
wireshark	14361	346
TOTAL	235202	8360

Table 3.2: Keyword Search through Review Comments

3.3.2 Step 2: Auditing Comments

In this step two persons independently reviewed each file for manually selecting which comments actually point to vulnerability in code. So if there is any comment which might contain any of the keywords but clearly does not pose a threat and irrelevant is rejected. We excluded a review request when both of the auditors think they should be rejected. In this step we found 3305 reviews. Table 3.3

Figure 3.1: Irrelevant Comment



```
}

void ConfigVariable::whereErrorToString(std::wostringstream &ostr)
```

Cédric DELAMAR... we manage the string of the call stack here Sep 9, 2015

```
{
```

Figure 3.1 shows an example where the “String” keyword is used but clearly not in the context. One observation is that the keywords “buffer”, “stack”, “heap”, “integer”, “gets”, “wrap” are the words which constructed the most irrelevant comments as they are common terms in coding contexts and used in numerous scenarios. Other keywords are more specific to the vulnerability, and were most likely to have some relation with potential defects.

Projects	**Comments Audited**	**Thoroughly Audited**
couchbase	1293	516
gerrit go	1735	644
gerrit review	615	294
kitware	186	65
libreoffice	311	118
ovirt	3212	1200
scilab	110	41
typo3	552	212
wireshark	346	215
TOTAL	8360	3305

Table 3.3: Thoroughly Audited Review Requests

3.3.3 Step 3: Verification

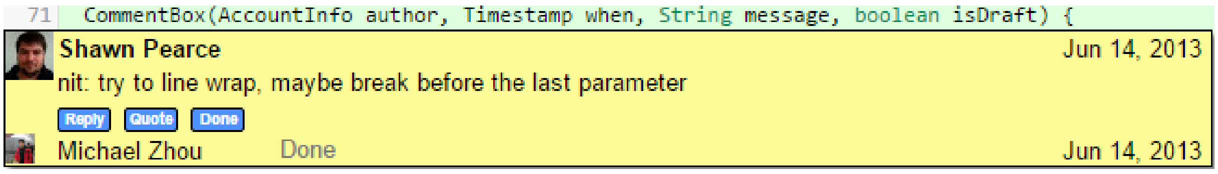
In the final step we did a thorough audit. We looked into the whole review discussions and corresponding codes in the associated patches for the 3,305 review requests. Two authors independently audited the review request in detail to determine whether the reviewer identified a vulnerability. We only accepted the review if and only if both auditors find it vulnerable. And the process of this final verification is very thorough. We looked into both code and patch sets. We looked into next patch sets for determining whether there was actually any vulnerable code. When a reviewer suggests a potential defect, we look if the editor of code response on the review request in the positive and changes the code accordingly. If so, we considered that a vulnerable code. We also considered a code vulnerable if the patch version on which the request was made was the last and the file was abandoned assuming the fixing might be too difficult which resulted in abandoning the file altogether. Also by looking at the discussion made by the review requester and author, we can verify the risk present in code. Conversely, if the editor gives no response to a reviewer suggesting threat, and the code was still merged in the code base, we conclude that it wasn't an actual vulnerability, as such vulnerabilities don't get merged even after some reviewer has pointed it out.

For example in Figure 3.2, we can see that Shawn Pearce, the reviewer posted a review request and the code author responded with "done". So we can conclude that there was some kind of vulnerable code which was changed by the code author due to the review request. In Figure 3.3 we can also find a similar case. But in Figure 3.4 the reviewer asked the author a question which has the intention of asking the coder to change his code into some thing else. But the author replies that the task the reviewer is asking him to do is not possible. So to understand if the code is really vulnerable we can look into next patch sets of the file. But in this case we find no change were made, the file was eventually merged, and so this is not vulnerable.

In Figure 3.5, we see a reviewer pointing out to an application crash threat which the author doesn't reply. We notice it's the last patch, and the file was abandoned. So we assume that it was a vulnerability; may be the fixing seemed too difficult at the moment so the whole file was abandoned altogether.

In Figure 3.6, we see a reviewer suggesting an issue which can lead to "buffer overflow" attack. The author recognizes it, and fixes it in the next patch. So, we marked it as a vulnerable code.

Figure 3.2: Vulnerable code 1

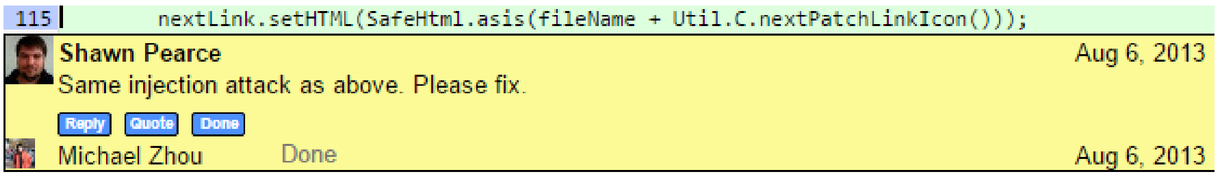


```
CommentBox(AccountInfo author, Timestamp when, String message, boolean isDraft) {
```

Shawn Pearce — Jun 14, 2013
nit: try to line wrap, maybe break before the last parameter

Michael Zhou — Done — Jun 14, 2013

Figure 3.3: Vulnerable code 2

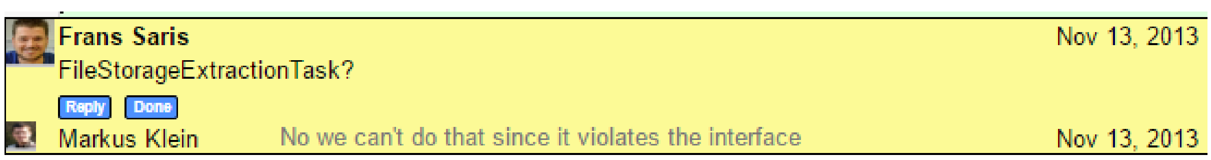


```
        nextLink.setHTML(SafeHtml.asis(fileName + Util.C.nextPatchLinkIcon()));
```

Shawn Pearce — Aug 6, 2013
Same injection attack as above. Please fix.

Michael Zhou — Done — Aug 6, 2013

Figure 3.4: Not Vulnerable code



Frans Saris — Nov 13, 2013
FileStorageExtractionTask?

Markus Klein — No we can't do that since it violates the interface — Nov 13, 2013

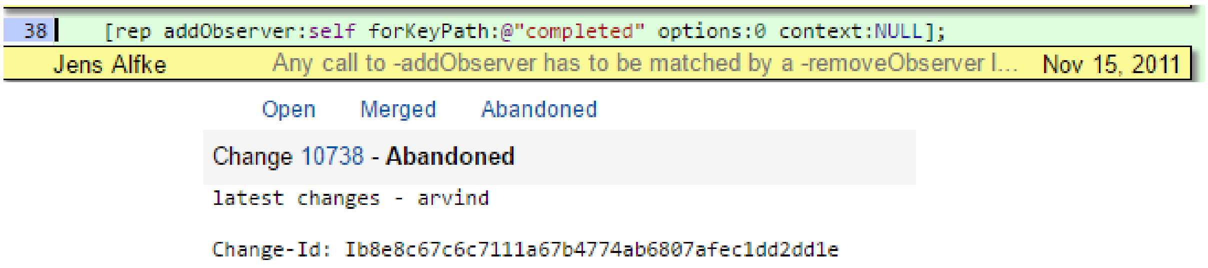
3.3.4 Step 4: Classification

After the final verification phase, the selected review request were classified into the categories given in Table 3.2. The result of this classification is given in Table 4.1.

3.4 Agreement Rate

From the data shown in Table 3.4, we measure the Cohen's kappa agreement rate between the raters. We find, k = .375 which indicates moderate agreement. This rating is more dependable in our domain, as kappa tends to give lower rates in scenarios like diagnosing rare diseases. As

Figure 3.5: Vulnerable due to being Abandoned



```
38    [rep addObserver:self forKeyPath:@"completed" options:0 context:NULL];
```

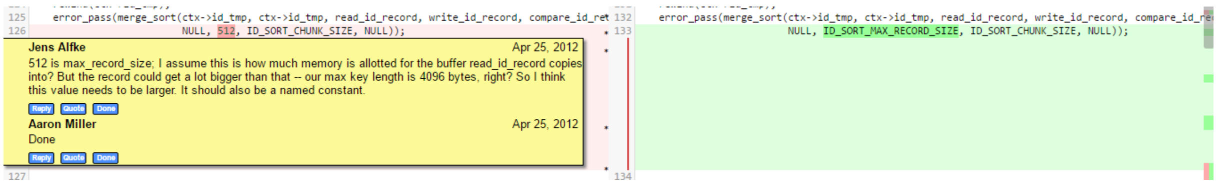
Jens Alfke — Any call to -addObserver has to be matched by a -removeObserver l... — Nov 15, 2011

Change 10738 - Abandoned

```
latest changes - arvind

Change-Id: Ib8e8c67c6c7111a67b4774ab6807afec1dd2dd1e
```

Figure 3.6: Vulnerable as the suggested fixing was made



```
error_pass(merge_sort(ctx->id_tmp, ctx->id_tmp, read_id_record, write_id_record, compare_id_ret
                      NULL, 512, ID_SORT_CHUNK_SIZE, NULL));
```

Jens Alfke — Apr 25, 2012
512 is max_record_size; I assume this is how much memory is allotted for the buffer read_id_record copies into? But the record could get a lot bigger than that -- our max key length is 4096 bytes, right? So I think this value needs to be larger. It should also be a named constant.

Aaron Miller — Apr 25, 2012
Done

```
error_pass(merge_sort(ctx->id_tmp, ctx->id_tmp, read_id_record, write_id_record, compare_id_re
                      NULL, ID_SORT_MAX_RECORD_SIZE, ID_SORT_CHUNK_SIZE, NULL));
```

security defects are analogous to rare diseases, this result is not problematic at all and indicates a good review from the raters.

Table 3.4: Result of thoroughly audited review requests

FILE	NAHID	NASIF	INTERSECTION
gerrit_go	146	118	50
gerrit_review	66	45	19
LIBREOFFICE	69	50	29
COUCHBASE	238	260	128
OVIRT	256	220	134
TYPO	64	60	37
WIRESHARK	89	73	73
SCILAB	35	32	32
KITWARE	32	23	21
	995	881	523

Chapter 4

Results

From the data set generation stage, we identified total 523 VCCs across 9 projects. For each VCCs, we kept track of it's project, review request id, final file status to the code base (Merged or Abandoned), vulnerability type, author, author's coding experience (along with percentile rank) in the project, reviewer, reviewer's experience(along with percentile rank) based on both his code and reviews in the project, patch set number, no. of files in the patch and number of lines changed. From these metrics, we try to answer the questions we asked in the Introduction, Section 1.3.

4.1 RQ1

4.1.1 Identification of Vulnerabilities through Peer Code Review

RQ1: Do peer code review successfully identify the top security vulnerabilities?

Table Table 4.1 shows the vulnerability distribution across their types and projects. Below, we discuss about our findings as we notice that peer code review is successful at identifying all the major threats and a lot of miscellaneous types of security defects too.

Memory Leak

The most common ($\approx 21\%$) vulnerability is memory leak. The severity of impact of this vulnerability depends on the program type. While short-lived user processes may pose very little threat, long-lived user processes or kernel processes may have severe implications. Although such classification is beyond our current study, our findings show that peer code review is very effective in identifying such leaks. Having the highest ratio among all vulnerability types, as native languages like C/C++ are very error prone to such leaks. Even for the managed & dy-

Table 4.1: Vulnerability Distribution

Project	Race Condition	Buffer Overflow	Integer Overflow	Improper Access	Cross Site Scripting (XSS)	Denial of Service (DoS) /Crash	Deadlock	SQL Injection	Format String	Memory Leak	Other	Total
Typo3	3	0	0	5	0	4	0	2	14	0	9	37
LIBRE OFFICE	4	2	4	0	0	4	0	0	0	12	3	29
Kitware	0	1	0	0	0	1	0	0	3	9	7	21
Gerrit RE-VIEW	7	0	0	0	0	1	2	4	0	4	1	19
Wireshark	2	4	9	0	0	6	0	1	21	21	9	73
Gerrit Go	12	4	5	0	0	15	3	0	4	2	5	50
Scilab	0	0	0	0	0	6	1	0	8	7	10	32
Ovirt	27	3	5	22	1	8	4	0	10	15	39	134
Couchbase	23	14	4	6	0	19	5	0	5	38	14	128
Total	78	28	27	33	1	64	15	7	65	108	97	523
% of Total	14.9	5.35	5.16	6.31	0.19	12.24	2.87	1.33	12.43	20.65	18.55	100%

namic languages like Java/Python, shared memory access could lead to such leaks on confusion over which part of the program is responsible for freeing the memory. It's also pretty eminent from our results. The majority of the leaks ($\approx 81\%$) happened in projects that used C/C++ files like Couchbase, Wireshark, LibreOffice, Kitware, SciLab projects. The Project Ovirt, which used python had 15 leaks detected; GerritReview, which used Java had 4, and GerritGo using Go language had 2 such leaks detected. A future study with equal share of projects from each language can reflect how each programming language is susceptible to such memory leaks, and if they match with insight of our findings which tells C/C++ to be the most vulnerable ones.

Table 4.2: Platforms

Project	Core Platform
GERRIT_GO	Go
GERRIT_REVIEW	Java
LIBREOFFICE	C++
COUCHBASE	C,C++,Matlab , Erlang
OVIRT	Java, Python
TYPO	php
WIRESHARK	C
SCILAB	C,C++
KITWARE	C++

Race Condition

The next common issue was the Race Condition ($\approx$ 15%). This is because we cover projects of large sizes, and like most modern programs, they rely on managing shared resources across threads. We cover all Race Conditions as a further classification is beyond this study. We also notice that among all the projects, Couchbase and Ovirt are the more susceptible ones to this vulnerability. The reason is not immediately apparent from our data, and we assume the underlying design of those projects should be the major cause here.

Buffer Overflow

Buffer Overflow($\approx$ 5.5 %), not unsurprisingly was another major finding. It's one of the most well-known software security threat and very easy for developers to make such mistakes no matter how experienced they are. Also 5 of our 9 projects use C/C++ which have the least protection against this threat.

Integer Overflow

Similarly Integer Overflow had almost similar share ($\approx$ 5 %) which is as well-known and ancient as Buffer Overflow. Unsafe type casting, integer arithmetic and failure to account for the differences in integers between 32-bit and 64-bit systems were the common reasons behind this issue. One point to notice, not only C/C++ projects, but Ovirt and GerritGo also showed a significant amount of integer overflow.

Format String Attack

Format String was a also a major threat ($\approx$12.5%) in our findings. It's no surprise that the C project, wireshark containing the lion's share of them ($\approx$ 33%) as C functions like printf, scanf are the major culprits behind this issue. Typo3 written in php is also found susceptible to it, and the Ovirt project too. The languages like Java, php are actually descendants of C that carry on various conventions. Format parameters like %x still exists in php and make functions like printf vulnerable to unvalidated inputs. However, our study shows peer code review is very effective in identifying such mistakes in the code.

Denial of Service(DoS)/Crash

The Denial of Service (DoS) and application crash were also found in good numbers, 64 to be specific and standing at over 12%. This is a major threat to commercial products, as even seconds of unavailability could cause a major revenue loss. Though Couchbase and Gerrit_Go had these issues in higher number, no immediate reason could be deduced. However, this threat is present in all platforms due to design issues and silly coding mistakes, Our results show peer code re-view's efficacy in identifying them early in the development cycle.

Improper Access

We also found Improper Access to be occurring in large amounts ($\approx 6\%$). It is a very broad term to coin, and we haven't done further classification. But the major causes were unvalidated input, unauthorized memory access, broken access control, broken authentication and session management and so on. It's not unlikely to have found these in large, as it's a high impact vulnerability, and mostly fixed in reviewing and testing period. We notice the project Ovirt having the most number of such threats. Tthe reason behind this is not immediately clear from our data. We also found a similar type of threat which is Cross-Site Scripting(XSS) in the project Ovirt.

Deadlock and SQLI

Though small in number, we also found issues of Deadlock and SQL Injection. It also proves the ability of peer code review in identifying them.

Others

Lastly, a vast majority of numerous securities, which we listed in the "Others" category were also found. In fact they were the second largest amount at $\approx$19%. These issues were found using common keywords that were not specific to any particular threat. We concentrated on some traditional top vulnerabilities to measure our method's efficacy, but the large number of miscellaneous threats being identified through the process further strengthens our assumption. It is a strong measurement of the performance of peer code review on improving overall security of the projects.

Implication

This results indicate that *peer code review is highly effective in detecting different vulnerability types very early in the development cycle. Therefore projects that are concerned with security, should adopt this process in order to achieve higher optimization of resources.*

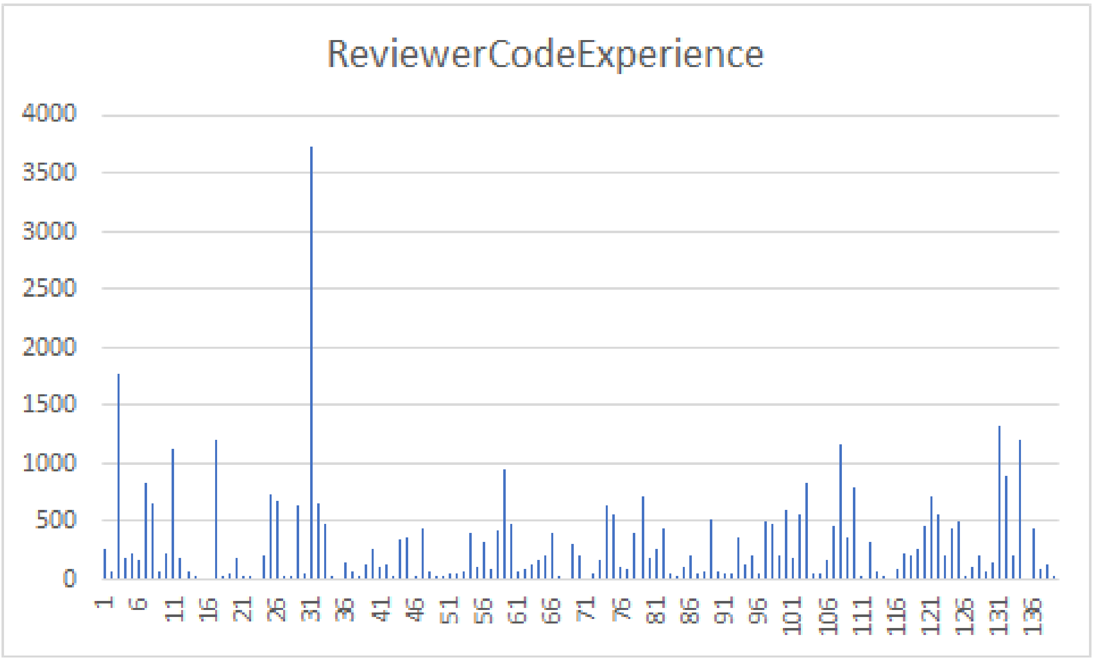


Figure 4.1: Reviewers' Coding Experience

4.2 RQ2

4.2.1 Experience

RQ2: Are the reviewers who identify the vulnerable code changes (VCCs) more experienced than than the authors who write it?

We got a data set of 523 vulnerabilities that were written by 214 different authors. Similarly these were reviewed by 139 different reviewers. We have a data set of considerable size to compare and test characteristics of the author and the identifier of a vulnerability.

We try to investigate if the project specific experience does any role in separating the author & identifiers' population. We define **Experience** of someone at the time of authoring/reviewing as, **"the number of posted code review requests (CRR) in the current project at the time the VCC was posted"**

The rationale behind our definition, is that more a developer contributes code change to the project, the more experienced he is on that particular project. This measure of experience is inspired from previous software engineering studies. Also, as our data set consists of 9 different projects, which have varying number of contributors, we calculated percentile ranks of each contributor based on such coding experience at the time of the author posting the VCC.

Normally, every code change is reviewed very soon after the change was requested. [4] So it is safe to assume that the reviewer doesn't significantly improve his experience within the time the VCC is posted and when he reviews it. So we are also measuring identifiers' experience for a vulnerability upto the time the VCC is posted, not reviewed.

Generally in Open Source Projects, coders and reviewers come from the same team. [9] That means, the same person writes some code and reviews some written by the others. It is also evident from the 139 reviewers who identified the vulnerabilities in our dataset. From Fig Figure 4.1 we can see, that 124 reviewers out of 139 has requested at least 15 code changes in the repository which matches our prior knowledge on how OSS projects are run.

From this point of View, We want to investigate the experience of author and identifier for a vulnerability. We hypothesize that, the identifier is generally more experienced than the author and experience is a major factor in being able to identify Vulnerability in the coding. In next section, we test this hypothesis in order to answer our second research question.

4.2.2 Hypothesis Testing

Normality Distribution

As we have a sample of moderate size(526), we first determine if our authors' and identifiers' experience follow a Normality distribution. We used the open source language R, for our statistical computation and visualization. We found that we can confidently assume authors' experience and reviewers' both have a non-normal distribution (Shapiro-Wilk, p-value $< 2.2e-16$; lower than significance,.05 for both the case). This indicates we have to use some non-parametric test to compare them. We choose the popular Wilcoxon signed-rank test in the next section to compare them.

Wilcoxon signed-rank test

Table 4.3: Statistical Tests

Step 1	Normality Test of Authors' Percentile Rank in Experience	Shapiro-Wilk, p < 2.2e-16	significance=.05	Non-Normal
Step 2	Normality Test of Reviwers' Percentile Rank in Experience	Shapiro-Wilk, p < 2.2e-16	significance=.05	Non-Normal
Step 3	Compare Difference between Authors' & Reviewers' Rank	Wilcoxon, p < 2.2e-17	significance=.05	Different
Step 4	Median of Author's Rank	80	Reviewers are generally more experienced than authors	
Step 5	Median of Reviwers' Rank	94		

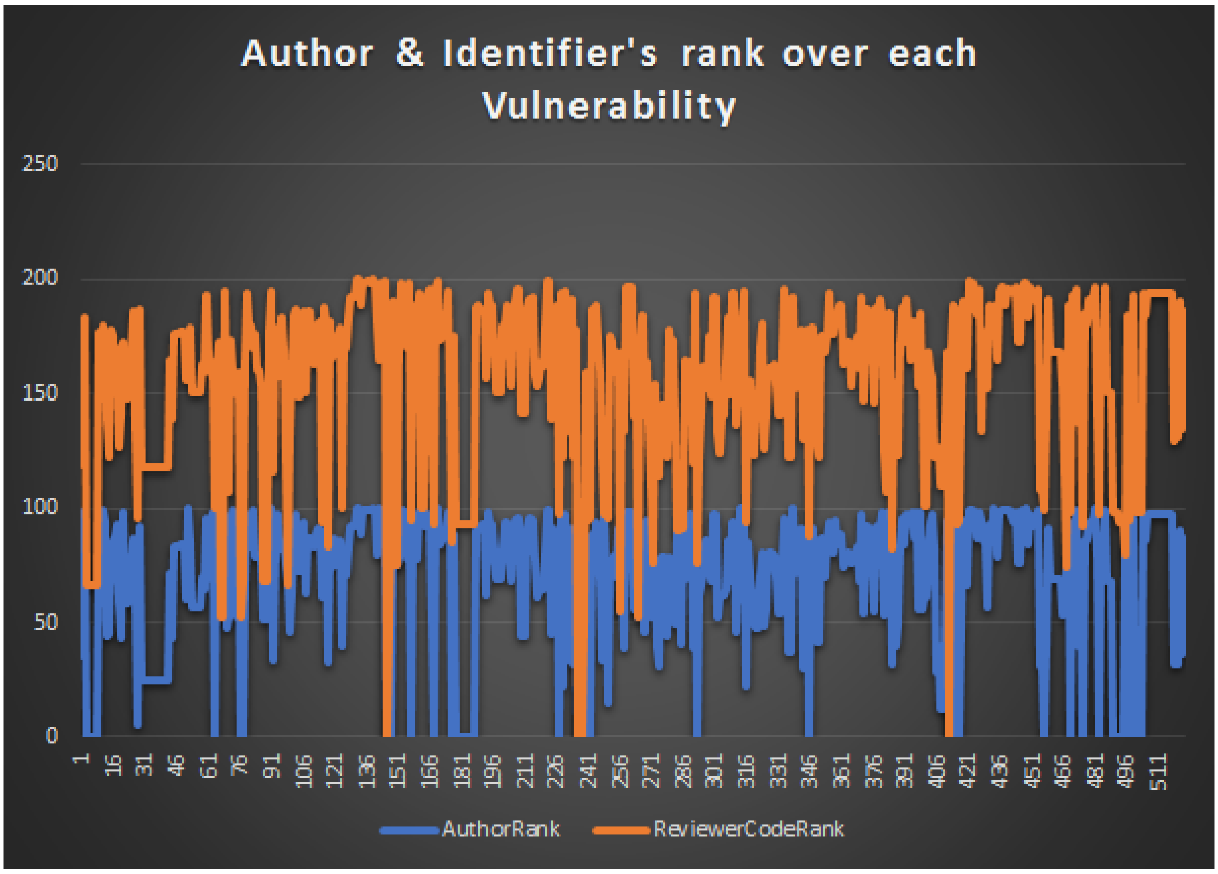


Figure 4.2: Reviewers' Coding Experience

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ. Our data fulfil the 3 major assumption of this test, that are-

- Data Samples are paired. We have an author and an identifier for each vulnerability.
- Data Samples are independent. The vulnerabilities we gathered are no co-related.
- Data Samples are ordinal as we use percentile ranks of experience.

Again we use R to perform wilcox test. We notice that two population are not equal (p-value $<$ 2.2e-16, lower than the significance, .05).

We then calculate their median. We find that the median of authors' percentile rank in experience is 80% while the reviewers' percentile rank in experience is 94%. This proves our hypothesis and gives a definite answer to our second research question, that the identifiers are more experienced than the authors.

4.2.3 Implication

Our results show that, *The ability to identify vulnerable code changes co-relates to experience as identifiers are more experienced than the authors in our data. So, if we assign reviewers who are more experienced than the author of a code change, we can hope for more success in correctly identifying the threats.*

Chapter 5

Discussions

5.1 The Uniqueness of Data Set

One of our major contributions is to prepare a data set of real life vulnerable codes and mistakes made in the development cycle of popular Open Source Projects, which we extracted through the review comments from their code repositories as they use the peer code review methodology. Most vulnerability databases contain vulnerabilities that were reported after the release which should be of graver standard. While we prepare a data set from the development cycle, where many trivial mistakes are done by the developers. So from our data set, we gain a new insight on what mistakes normally the coders do when they write the code initially. *Training the team based on this knowledge would help authors to write less such weak codes and also will boost the reviewers and overall security team's ability to successfully identify them.*

5.2 Implications of our Result

We shed light on another advantageous aspect of peer code review in addition to numerous well-known benefits, e.g., detecting bugs [10,11], improving relationship among the team [12], ensuring code integrity [13], spreading knowledge within the team [14, 15] and so on. It gives us one more solid reason in adopting this methodology in development. Specially, for the open source projects, where the code is open to extreme scrutiny by the potential attackers, security is the utmost priority and our result shows peer code review is very effective in detecting security threats.

This result is especially intriguing and exciting to us, as code review happens very soon in peer code review [4]. So, peer code review has the ability to detect VCCs in the earliest of times. And as we know, the cost of vulnerability goes up with how much it lingers in the code base

[5]; we can adopt peer code review as the most cost-effective solution to our security threats. Also, in such peer review techniques, coders and reviewers come from the same team. [9]. And as we have identified in our study, that most trivial mistake is genrally could be fixed by the more experienced coder from the team; the duty load on the separate security team for the final testing reduces in a great deal even if we want to assign one. As security experts are scarce [1], a small security team with smaller range of work would reduce the cost in a great deal.

The Effectiveness of peer code review, which we have shown in our study, is highly exciting mainly due to-

- *vulnerabilities are identified in quickest time after it has been written*
- *vulnerabilities are identified from within a team, not from external security experts*

Our next results show that, Reviewers who identify the VCCs are generally more experienced than the authors, where experience was measured on the acquaintance with that specific project. There could be multiple reasons behind these issues.

- More acquaintance with the code of the project gives the coder more knowledge about the overall system design, thread related and resource management issues which in truns make him capable in identifying subtle mistakes.
- The ones who contribute more to the code base could be the leading developers overall. Project experience indicates to overall lifetime experience, which in turn indicates to security knowledge.
- The other parameters which were taken into account in assigning reviewers in these projects co-related with their security knowledge, and thus the outcome of our study.

We can use our findings in the process of choosing reviewers in order to achieve better secured products. Effectively choosing code reviewer is a challenge in peer code review, and studies were made on how to improve it [16, 17]. *Our result adds another parameter (experience) for the project manager or the automated reviewer recommendation tools in choosing reviewers while dealing with security in priority.*

5.3 Comparison with Previous Study

Our works is inspired from a previous study [7] where a similar kind of data set were prepared and characterizations were made.

The results of our study is much comparable. In the previous paper, 10 similar open source projects were chosen, from which 413 vulnerabilities were found.

The main difference was that we included the keyword “leak” to concentrate on the “memory leak” issue as an addition. It created a major change in our result, as our major findings were of this issue ($\approx 21\%$). But in the previous paper, “memory leak” was listed in “Others” which constituted around 9% in their results.

In our study, “Denial of Service/Crash” and “Format String Attack”, also had a higher percentage of presence. The majority of projects being C/C++ projects, addition of keywords that deal with unsafe string functions (e.g. strlen, strcat), addition of keyword “heap” which dealt with some memory related issues were some of the reasons for this increase. We also seen an increase in the “Others” category. The revision of keywords and project selection should be the reason here.

Conversely, “Cross Site Scripting(XSS)” and “Bufef Overflow” have seen a decrease in presence. XSS was found only once as the project domains were not primarily web-based. While the case of “Buffer Overflow” is an interesting one. No immediate reason could be found, and most probably it’s just due to project selection.

Otherwise, our study pretty much supports the previous one and strengthens the idea of peer code re-view’s effectiveness in identifying security related vulnerabilities.

5.4 Recommendations

- We tracked down many trivial and common mistakes that were committed during the development cycle. We recommend training the coders on this common issues beforehand, so that such VCCs get reduced. Also they become more able in identifying VCCs made by others.
- We encourage detailed discussions from both the reviewer and author about such security defects as only then it will extract the truest values of peer code review such as dissemination of knowledge, avoidance of future similar mistakes or more data to deal with for reaserchers like ours.
- As we have seen similar type of mistakes occur again and again, it reminds us of the necessity of automatic code reviewer tools to automate the process.
- We recommend a dynamic selection of code reviewers. Initially the most reputed one could start reviewing. As soon as the novice authors submit more codes, and gets more experienced, he/she should be starting reviewing others codes.

Chapter 6

Threats to Validity

6.1 Internal Validity

6.1.1 Code Review Tool Selection

As we extracted our data from all the projects that use Gerrit Code Review Tool, there could be a possibility that projects that use other code review tool could behave differently. But this isn't a threat if we consider-

- all code review tools performs more or less the same functionalities
- no code review tool has a special focus on security. So there's no reason whya different tool would behave differently in the study that we did.

6.2 External Validity

6.2.1 Project Selection

Although Project selection was random, they still cover a set of multiple languages [see Table 4.2].And the kind of security threats we concentrate on our study, the projects' domain and platform are susceptible to them, except threats that are specific to web interaction (e.g. XSS) which we don't havily concentrate on. From that point of view, project selection poses minimal risk to the validity of our final results in the study.

6.3 Construct Validity

6.3.1 Completeness of Keyword Set

This is an obvious question. As we search for the vulnerabilities through a set of keywords; the completeness of our search depends on the completeness of our keyword list.

To validate our keyword set, we prepared a secondary set of keywords. We prepare this secondary set based on our knowledge and observation, as they are too related to the vulnerabilities we discuss in this study. However, we excluded them from our initial list because they point to vulnerabilities in limited scenarios. And if they do, it's highly probable that one of our primary keywords are also in the comment. Table 6.1 shows the list of our secondary set.

Using these keywords, we again do a database search looking for the comments which does not have any keyword from our primary list, but contain any one from this secondary keyword list. The result of the review requests that are extracted in this search is show in Table 6.2.

Table 6.1: Secondary Keyword List

Keyword	Likely Reason
stale	stale memory/stale pointer
hijack	session hijacking
validat	input validation
safe	not ensuring safety
overread	synonymous to overrrun/overflow
bound	for getting out of boundary
stack	stack smashing
synch	for race situations
NPE	null pointer exceptions
magic	magic strings
wild	wild pointer

Table 6.2: Review Requests extracted in Secondary Search

Project	Review Requests
Couchbase	67
Gerrit_review	48
Gerrit_go	112
kitware	1
libreoffice	18
ovirt	167
scilab	0
typo3	21
wireshark	13
Total	447

As we can see, only 447 new review requests are found in this search, while we got a overwhelming size of 8360 review request with our primary keyword set. If we take the probability of a comment pointing to actual VCCs from first search, which is (523/8360) approximately .06; such low probability indicates this new set of 447 review requests doesn't add much detail to our initial result.

Also, even if there are some vulnerabilities that are missed, this doesn't affect our final result. As we're more concerned in peer code re-view's ability to detect VCCs, missing VCCs that may have been identified only adds to our cause.

Also, for our second research question, we already have a set of 523 VCCs to run a scientific statistical test to come to a conclusion. As we have shown by our secondary keyword search, very few VCCs that we might have missed, they would not alter the population significantly to change our final result.

6.3.2 Manual Auditing

As we depended on manual auditing in identifying VCCs, there's always a threat to the result due to accidental error, experimenter's' bias & limitations and lack of proper attention.

To counter this problems and reduce the threat to minimum, in Section 3.3.2, two authors independently reviewed the comments and discarded them as not VCCs only if both of them agreed. This reduces the chance of any VCCs being missed to a great extent.

In , however we became more restrictive and identified a review request as VCC only if both the author agreed. The rational behind these are -

- It is more probable to identify a non-VCC as a VCC than the opposite. As many general bugs could be wrongly counted as a security threat.
- In our final data set, we emphasize more on every sample being truly a VCC than an actual VCC being missed out. As we have previously discussed, a small addition of VCCs don't alter our results in a great manner. That's why if one author considered it not being a VCC, we didn't count it in the final data set.

The Cohen Kappa agreement rate in the Section 6.3.2 is .375, which is acceptable considering the type of the domain.

Furthermore, to ensure our final data set contains all true VCC, Dr. Amiangshu Bosu verified them again manually.

Thus we reduce the threat to the minimum in auditing.

6.3.3 Measurement of Experience

In answering , we define experience in a particular way although this is a multi-factorial, complex and not entirely a measurable attribute. But we believe our measurement should be a considerable representation of ones experience and familiarity with the project.

And the motivation of our study is to recommend a guideline while assigning code reviewer to a code change. And because our result is completely based on our definition of “experience”, if a project manager or an automated tool assigns code reviewers under the same definition of “experience”, he should expect the same result that we concluded.

6.4 Conclusion Validity

6.4.1 Conclusion from RQ1

Here, we concentrate on top security threats, and some generic keywords which extend to all defects; and from the result we get based on them, we extend our conclusion to all top security vulnerabilities. However, the reasoning is that we noticed all the security threats that we concentrated can be identified by peer code review. Also our “Others” list constitute of multiple types of threats. So it is safe to conclude from this approach that, peer code review should be able to identify most of the security threats that is well-known among the programmers.

6.4.2 Conclusion from RQ2

One major threat to validity is that we didn’t consider the difference between authors’ rank and reviewers’ rank over all the review requests. If generally the experience gap for all the review requests is same as it is for the VCCs, then our conclusion that experience plays a role in identifying vulnerabilities would be a hastily drawn one.

Chapter 7

Future Work

- The frequent occurrence of trivial mistakes reminded us about the necessity of automatic code reviewing tool. The way we built this data set, could be of major use if we want an automatic VCC detection tool. We have the corresponding code base, related discussion, and all other information that the code review tool stores for a patch. We could process these data, feed them to a proper learning model and may generate good results. These could bring ground breaking change in modern software development. So, how to effectively use these data and build an auto-detector is a future challenge.
- More type(of vulnerability) specific studies could be made upon ensuring a comprehensive data set to better analyze the authors' and reviewers' response to them. Like if authors do repeat similar type of mistakes, if some specific type of VCCs come more from some specific type of authors, or could be identified by similar type of reviewers are all questions that can be asked from a larger data set.
- The data set could be used in better understanding of the security defects also. We could use the corresponding codes and the discussions for establishing good training system.

We hope the data set we built, will be helpful to do these future works, as larger the data set, greater the efficiency of the study.

References

[1] G. McGraw, *Software security: building security in*, vol. 1. Addison-Wesley Professional, 2006.

[2] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*, pp. 489–498, IEEE Computer Society, 2007.

[3] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*, pp. 31–38, ACM, 2008.

[4] A. Bosu and J. C. Carver, "Peer code review in open source communitiesusing reviewboard," in *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, pp. 17–24, ACM, 2012.

[5] G. McGraw, "Automated code review tools for security," *Computer*, vol. 41, no. 12, 2008.

[6] S. Christey and R. A. Martin, "Vulnerability type distributions in cve," *Mitre report, May*, 2007.

[7] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 257–268, ACM, 2014.

[8] A. Bosu and J. C. Carver, "Peer code review to prevent security vulnerabilities: An empirical evaluation," in *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*, pp. 229–230, IEEE, 2013.

[9] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," tech. rep., Technical Report DCS-305-IR, University of Victoria, 2006.

[10] M. Fagan, "Reviews and inspections," *Software Pioneers–Contributions to Software Engineering*, pp. 562–573, 2002.

[11] K. E. Wiegers, *Peer reviews in software: A practical guide*. Addison-Wesley Boston, 2002.

[12] A. Bosu and J. C. Carver, "Impact of peer code review on peer impression formation: A survey," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pp. 133–142, IEEE, 2013.

[13] J. Cohen, E. Brown, B. DuRette, and S. Teleki, *Best kept secrets of peer code review*. Smart Bear, 2006.

[14] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 202–212, ACM, 2013.

[15] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering*, pp. 712–721, IEEE Press, 2013.

[16] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pp. 141–150, IEEE, 2015.

[17] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving code review effectiveness through reviewer recommendations," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 119–122, ACM, 2014.